Model Checking and Model-Based Testing

Improving Their Feasibility by Lazy Techniques, Parallelization, and Other Optimizations

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

David Faragó

aus Bretten

Tag der mündlichen Prüfung: 29.01.2016

Erster Gutachter: Prof. Dr. Peter H. Schmitt,
Karlsruher Institut für Technologie

Zweiter Gutachter: Prof. Dr. Shmuel Tyszberowicz,
The Academic College of Tel Aviv Yaffo

This dissertation is available in several versions (see `http://bit.do/diss`):

- a free pdf at KIT's open access
- a kindle version at Amazon
- a print-on-demand back matter at Amazon
- a print-on-demand front matter and main matter at Amazon: either colored or in black and white.

The following 24 figures and 11 listings contain color that facilitates (but is not required for) understanding: Fig. 1.1 (p. 6), Fig. 3.1 (p. 37), Fig. 4.1 (p. 66), Fig. 4.3 (p. 85), Fig. 6.2 (p. 120), Listing 6.3 (p. 123), Listing 6.4 (p. 124), Listing 6.5 (p. 125), Fig. 6.5 (p. 132), Fig. 6.6 (p. 133), Fig. 6.7 (p. 134), Listing 6.7 (p. 139), Fig. 6.9 (p. 148), Listing 8.2 (p. 211), Listing 8.3 (p. 215), Fig. 9.1 (p. 232), Listing 10.1 (p. 238), Listing 11.1 (p. 254), Fig. 11.1 (p. 255), Listing 12.1 (p. 298), Listing 12.2 (p. 300), Fig. 12.4 (p. 322), Fig. 13.2 (p. 332), Listing 13.1 (p. 337), Fig. 14.1 (p. 342), Fig. 14.2 (p. 346), Fig. 14.3 (p. 356), Fig. 14.4 (p. 359), Fig. 14.5 (p. 360), Fig. 14.6 (p. 362), Fig. 14.7 (p. 365), Fig. 14.8 (p. 367), Fig. B.2 (p. 387), Fig. B.3 (p. 388), and Fig. B.7 (p. 392).

The cover of this book was designed by Laura Sauer and Karin Mahler and illustrates that checking whether we have correct parallelism is difficult.

Acknowledgements

Firstly, I express my gratitude to my Doktorvater Prof. Dr. Peter H. Schmitt for a great work environment, the incentive for continuous improvement, and the trust he placed in me. I am thankful to Prof. Dr. Shmuel Tyszberowicz for agreeing to be the second reviewer of this thesis, and for iteratively giving thorough feedback on this thesis. Thanks also go to Prof. Dr. Ralf Reussner and to Prof. Dr. Bernhard Beckert for being examiners in the thesis defense.

I am highly grateful to my current and former colleagues, Dr. Christian Engel, Dr. Frank Werner, Dr. Benjamin Weiß, Dr. Mattias Ulbrich, Dr. Christoph Scheben, Dr. Carsten Sinz, Florian Merz, Reimo Schaupp, Markus Iser, and Dr. Tomás Balyo for the pleasant working atmosphere, teamwork, discussions, helpfulness and friendship. I also thank the current and former members of the other verification-related research groups at the Karlsruhe Institute of Technology: Dr. Thorsten Bormer, Dr. Stephan Falke, Dr. Aboubakr Achraf El Ghazi, Dr. Christoph Gladisch, Dr. Daniel Grahl, Sarah Grebing, Simon Greiner, Mihai Herda, Michael Kirsten, Dr. Vladimir Klebanov, Tianhai Liu, Dr. Hendrik Post, Dr. Mana Taghdiri, Dr. Olga Tveretina, and Alexander Weigl. Further thanks go to our administrators Lilian Beckert, Simone Meinhart, and Ralf Koelmel, and to Bernhard Klar for his statistical consulting.

The work in this thesis was strongly enhanced by a lot of teamwork, also outside the institute: During our work on $\text{PDFS}_{\text{FIFO}}$, Dr. Alfons Laarman, from the Formal Methods & Tools Group at University of Twente gave references and thorough answers to all of my questions, which was very helpful in diving into LTSmin and concurrency. Thanks for the great collaboration and your help. Lars Frantzen, from Radboud University Nijmegen, gave many references about formal testing, especially the ioco theory and the testing hypothesis. Thanks for our regular discussions. Dr. Axel Belinfante from the Formal Methods & Tools Group at University of Twente provided JTorX, related help and support in integrating my work. Thanks a lot. I am grateful to all other editors of the model-based testing community [URL:MBT-C], Dr. Baris Güldali, Dr. Michael Mlynarski, Dr. Sebastian Oster, Arne Michael Törsel and Dr. Stephan Weißleder, for the discussions, references and advice, especial related to practical aspects of model-based testing and testing in general. The same applies to all active members of the Gesellschaft für Informatik, Fachgruppe "Test, Analyse und Verifikation von Software", Arbeitskreis "Testen objektorientierter Programme / Model-Based Testing" [URL:GI-TAV TOOP]. I am very thankful to Dr. Carsten Sinz and Dr. Frank Werner for their expertise and guidance on software bounded model checking and related tools and technologies, as well as for offering a computer cluster so I was able to handle the huge amount of experiments. Dr. Roland Glantz from the Parallel Computing Group at the Karlsruhe Institute of Technology gave helpful consulting in algorithm engineering, especially for my experiments on distributed LazyOTF. Thank you. I thank Stefan Nikolaus from WIBU SYSTEMS AG for his help in the case study with CodeMeter License Central.

I thank Leif Frenzel from andrena objects AG for his proficient feedback and advice in relation to agile software development, especially agile testing. Many thanks go to my long-term student research assistant Felix Kutzner for the proficient dialogs and his clean implementations with great design. I thank all the anonymous reviewers of my publications (see Subsec. 1.2.4) for their valuable comments. Special thanks go to my friends and family. Finally, I am thankful for the grants I received from the Exzellenzinitiative (project FYS on formal verification, which lead to DFS_{FIFO}) and the Deutsche Forschungsgemeinschaft (project MOCHA, which produced LazyOTF).

Karlsruhe, December 2015
David Faragó

Modellprüfung und modellbasiertes Testen

Höhere Durchführbarkeit durch träge Techniken, Parallelisierung und weitere Optimierungen

Zusammenfassung (German Summary)

Diese Arbeit verbessert die Qualitätssicherung von (sicherheitskritischer) Software, indem sie die Durchführbarkeit von Modellprüfung (engl.: model checking) und modellbasiertem Testen steigert. Dadurch können größere Systeme analysiert werden, die Analyse bedarf weniger Aufwand und Expertise, und das Risiko, Fehler zu übersehen, wird reduziert. Hiermit wird die Wirtschaftlichkeit der Qualitätssicherung erhöht - ein Muss für die Akzeptanz Formaler Methoden in der Industrie [Hunt, 2011; Weißleder et al., 2011; Faragó et al., 2013].

Motivation

Software ist heute allgegenwärtig, ihre Komplexität und Sicherheitsanforderungen steigen. Da die Sicherheit für komplexe Software schwierig zu gewährleisten ist, führt dies zu einem Dilemma zwischen Softwarequalität und Softwarekomplexität und somit zu einem Problem in der Qualitätssicherung: Mittlerweile werden 50% bis 70% der Software-Entwicklungskosten in die Qualitätssicherung gesteckt, trotzdem entstehen pro Jahr weltweit mehrere Zehn-Milliarden Euro Kosten durch Softwarefehler.

Einer der wichtigsten Lösungsansätze sind Formale Methoden: Diese operieren auf formalen Beschreibungen der Anforderungen an die Software, um die Korrektheit der Software zu überprüfen, d.h. deren Übereinstimmung mit ihren Anforderungen. Diese Arbeit beschäftigt sich mit automatisierbaren Formalen Methoden, denn weniger Aufwand an Zeit und Expertise durch Automatisierung führt zu höherer Durchführbarkeit in der Industrie. Die betrachteten automatischen Methoden sind:

- Modellprüfung (MC), z.B. mit SPIN oder LTSmin (siehe Kapitel 5), die auf Systemen mit endlichem Zustandsraum operiert und durch explizite oder implizite Aufzählung des Zustandsraumes temporale Eigenschaften verifiziert. MC ist korrekt und vollständig, aber der Benutzer muss meist das System abstrahieren, um eine Zustandsraumexplosion (engl.: state space explosion) zu vermeiden;
- beschränkte Modellprüfung (BMC) ist eine spezielle Variante der Modellprüfung, welche den Zustandsraum nur bis zu einer vorgegebenen Tiefe aufzählt. Dies steigert die Durchführbarkeit zu einem gewissen Grad, ist aber nicht immer vollständig;
- modellbasiertes Testen (MBT) generiert Testfälle aus einer Spezifikation. Vollständigkeit ist zwar theoretisch erreichbar, wird aber für praktische Anwendungen zu

Gunsten der Durchführbarkeit aufgegeben. Zusätzlich testet MBT das tatsächliche System.

Abb. 15.1 zeigt eine genauere Positionierung dieser Arbeit.

Trotz ihrer Vorteile werden Formale Methoden in der Industrie nur wenig eingesetzt, da sie noch nicht hinreichend durchführbar sind [Bienmüller et al., 2000; Kreiker et al., 2011; Hunt, 2011; Gnesi and Margaria, 2012]. Wird die Durchführbarkeit erhöht, können Formale Methoden die oben beschriebenen Anliegen der Entwicklung (sicherheitskritischer) Software lösen [DO178C Plenary, 2011; Dross et al., 2011].

Überblick

Das Ziel dieser Arbeit ist die Steigerung der Durchführbarkeit Formaler Methoden: Die Zeit- und Speicher-Anforderungen sollen gesenkt werden, damit mehr Probleme (neue, größere oder weniger abstrakte Probleme) geprüft werden können. Zusätzlich soll die Ausdrucksstärke und Benutzerfreundlichkeit erhöht werden.

Da die Zustandsraumexplosion für MC gravierend ist, haben wir den Algorithmus DFS_{FIFO} entwickelt; er erhöht die Durchführbarkeit, indem er träge (engl.: lazy) Techniken verwendet und sich auf eine bestimmte Lebendigkeitseigenschaft spezialisiert, die Abwesenheit von Livelocks. Die Durchführbarkeit wird weiter erhöht durch zusätzliche Optimierungen: verstärkte Halbordnungsreduktion (engl.: partial order reduction), Parallelisierung mit nahezu linearem Speedup und verbesserte Benutzerfreundlichkeit und Leistung, indem Fortschritt (engl.: progress) durch Transitionen statt Zuständen modelliert wird. Insgesamt wird eine Verbesserung um vier bis fünf Größenordnungen im Vergleich zu etablierten Livelock-Prüfungen erzielt. Die Prüfung noch größerer Probleminstanzen kann durch die Zustandsraumexplosion aber immer noch nicht durchgeführt werden. Deswegen untersucht diese Arbeit, ob BMC die Prüfung von Sicherheitseigenschaften eines großen C-Programms durchführen kann. Diese Fallstudie zeigt, dass auch für BMC die Zustandsraumexplosion auftritt, wenn die untersuchten Programme komplex sind. Die Durchführbarkeit wird mittels Abstraktion und einem leichtgewichtigeren Vorgehen über den Testansatz erzielt, was auch den Einsatz von MBT motiviert.

MBT integriert MC-Algorithmen mit dem Testansatz, um Sicherheitseigenschaften zu prüfen. Vollständigkeit, komplette Überdeckung, sowie bestimmte Ziele sind für große Programme nach wie vor nicht erreichbar. Die Situation wird etwas verbessert durch unsere Erweiterung der ioco-Theorie: ein deterministischer, beschränkter Testgenerierungs-Algorithmus und Vollständigkeitsschwellen (engl.: exhaustiveness thresholds), ähnlich zu BMC, verringern die Redundanz und Größe der generierten Testsuite. Die Vollständigkeitsschwelle ist oft zu Groß, um in der Praxis angewandt zu werden. Unsere ioco-Erweiterung wird aber auch in unserem neuen Ansatz für MBT, genannt LazyOTF, verwendet, das auch unvollständiges MBT anbietet. Mittels träger Techniken integriert LazyOTF offline MBT, welches zuerst alle Testschritte generiert und erst dann ausführt, und on-the-fly MBT, welches die Generierung und Ausführung von Testschritten streng verzahnt. LazyOTF führt teile der generierten Testfälle träge on-the-fly auf dem System aus, zu den sinnvollsten Zeitpunkten, beispielsweise wenn ein Ziel, eine Testtiefe oder unkontrollierbarer Nichtdeterminismus erreicht wird. Im Gegensatz zu offline MBT wird bei LazyOTF die Durchführbarkeit nicht durch die Zustandsraumexplosion verhindert. Im Vergleich zu on-the-fly MBT hat LazyOTF eine bessere Zielsuche, so dass eine bessere

Überdeckung sowie bestimmte Ziele erreichbar werden. Um die Durchführbarkeit weiter zu erhöhen, entwickelt diese Arbeit neue Heuristiken, welche durch die träge Technik sowohl Rücksetzverfahren (engl.: Backtracking) als auch dynamische Information nutzen kann. Wir führen zusätzlich Testobjekte ein, welche die Benutzerfreundlichkeit durch einfache Konfigurierbarkeit und Komposition verbessert. Wir erhöhen die Leistung von LazyOTF weiter mittels Parallelisierung mit nahezu linearem Speedup sowie weiteren Optimierungen. Insgesamt erreicht LazyOTF Überdeckungen sowie bestimmte Ziele mehrere Größenordnungen besser als bisherige MBT-Verfahren, was die Durchführbarkeit von großen und nichtdeterministischen Systeme ermöglicht.

Zusammenfassend hat diese Arbeit einen neuen MBT-Ansatz eingeführt, um die Prüfung von Sicherheitseigenschaften durchführbar zu machen. Da Lebendigkeitseigenschaften nicht durch dynamisches Testen geprüft werden können, wurde für eine wichtige Klasse von Lebendigkeitseigenschaften, den Livelocks, ein spezialisierter MC-Algorithmus entwickelt, der auch diese Prüfungen durchführbar macht.

Alle Verbesserungen im Vergleich zu den besten bisherigen Verfahren werden durch Messungen belegt:

Für unsere Experimente mit vier etablierten Protokollen ist der Speicherverbrauch von DFS_{FIFO} 3 bis 200 mal kleiner, die Laufzeit 3.4 bis 16 mal kleiner. Zusätzlich ist sein on-the-fly-Verhalten (engl.: on-the-flyness) über 150 mal stärker. Leider verschlechtert sich das on-the-fly-Verhalten vom parallelen DFS_{FIFO} ($\text{PDFS}_{\text{FIFO}}$) bei 48 Instanzen auf einen Faktor von 1.75, allerdings ist die Ursache die starke Verbesserung vom besten bisherigen Verfahren (CNDFS) von 1 auf 48 Instanzen. $\text{PDFS}_{\text{FIFO}}$ erziehlt nahezu lineares Speedup, und $\text{PDFS}_{\text{FIFO}}$ mit Halbordnungsreduktion kann vier bis fünf mal größere Probleme handhaben als die besten bisherigen Verfahren. Für relevante Livelocks in unseren Experimenten sind die Gegenbeispiele von $\text{PDFS}_{\text{FIFO}}$ bis zu 10 mal kürzer.

Für unsere Experimente mit einer akkumulierten Wallclock-Zeit von 14 Jahren sind LazyOTFs Aussagekraft (engl.: meaningfulness) und Laufzeit exponentiell besser. Zusätzlich erzielt verteiltes LazyOTF (engl.: distributed LazyOTF) (super-)lineares Speedup für die aussagekräftige Testausführung und insgesamt nahezu lineares Speedup. Leider verbessern unsere dynamischen Beschränkungs-Heuristiken das Verhältnis zwischen Aussagekraft und Generierungszeit der Testfälle nur für manche Situationen und manche Parameter. Deswegen sind weitere Untersuchungen und erweiterte Beschränkungs-Heuristiken mit noch mehr dynamischen Informationen relevante Arbeit für die Zukunft.

Die Beiträge im Einzelnen

Der wichtigste Beitrag zu MC ist der Entwurf, die Implementierung und Analyse des Algorithmus $\mathbf{DFS_{FIFO}}$, um Livelock-Prüfungen durchführbarer zu machen. DFS_{FIFO} nutzt on-the-fly MC, traversiert aber Fortschritt träge. Hiermit kann DFS_{FIFO} simultan in einem Durchlauf den Zustandsraum durchsuchen und auf Nichtfortschritts-Zyklen prüfen. Dies erhöht die Durchführbarkeit, sowohl direkt als auch indirekt durch bessere Optimierungsmöglichkeiten (siehe weitere Beiträge). Somit ist DFS_{FIFO} der neue Stand der Technik zur Livelock-Prüfung.

Der wichtigste Beitrag zu MBT ist der Entwurf, die Implementierung und Analyse des Verfahrens **lazy on-the-fly MBT** (**LazyOTF**), welches die Zielsuche von on-the-fly MBT verbessert: LazyOTF integriert offline und on-the-fly MBT synerge-

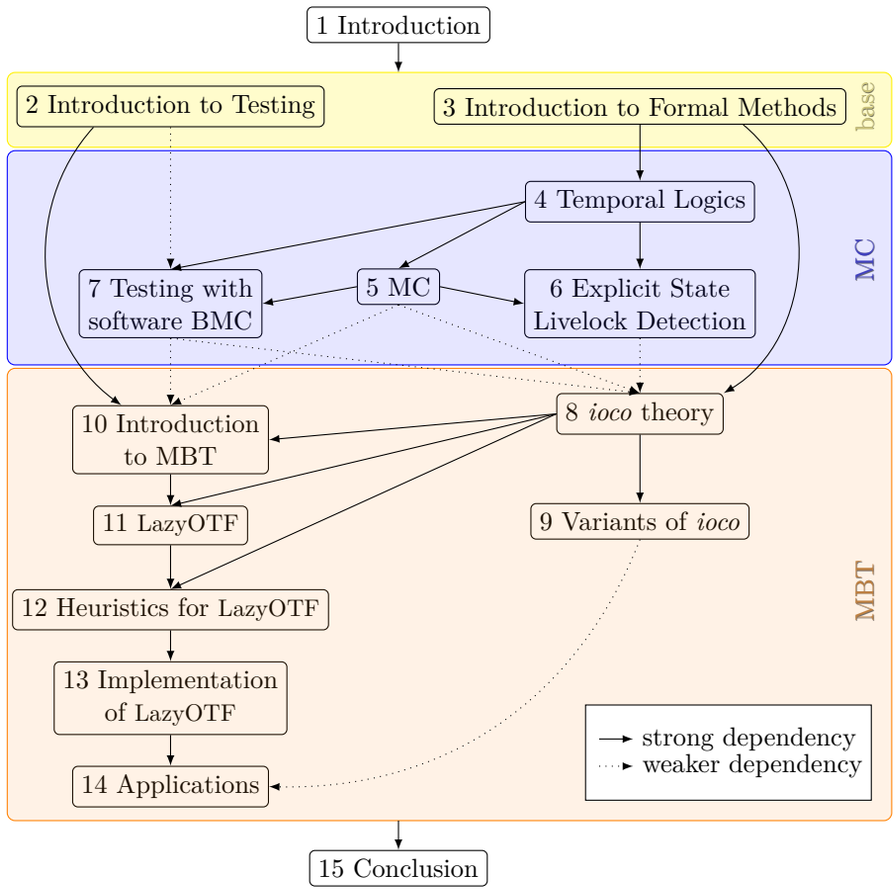
tisch, indem es zu den sinnvollsten Zeitpunkten zwischen Testgenerierungs-Phasen und Testausführungs-Phasen hin und her wechselt. Hierdurch kann LazyOTF mit verbesserter Zielsuche effizient aussagekräftige Testfälle selektieren und ausführen, auch für große und nichtdeterministische Systeme.

Auf der Metaebene bilden **träge Techniken** und deren Verwendung für Formale Methoden den wichtigsten Beitrag.

Die weiteren Beiträge dieser Arbeit (siehe Abb. 1.1) sind:

- eine verallgemeinerte automatentheoretische Basis für temporale Logiken, MC, ioco-Theorie, und MBT. Die ioco-Theorie wurde erweitert um eine ausführliche Test-Hypothese, eine Verfeinerungsrelation (engl.: refinement relation), mehrere Fairness-Bedingungen, Überdeckungskriterien, einen abstrakten deterministischen Testfallgenerierungs-Algorithmus, Vollständigkeitsschwellen und deren Verhältnis zu Fairness und Überdeckungen;
- Optimierungen und Anpassungen der Halbordnungsreduktion, die für DFS_{FIFO} verstärkt werden kann im Vergleich zu allgemeinem LTL MC;
- Optimierungen via Parallelisierungen, welche durch unsere trägen Techniken vereinfacht werden: unser mehrfädiges (engl.: multi-threaded) DFS_{FIFO} ($\text{PDFS}_{\text{FIFO}}$, in Kollaboration mit Alfons Laarman) erzielt nahezu lineares Speedup, unser verteiltes LazyOTF hat (super-)lineares Speedup für aussagekräftige Testausführung und insgesamt nahezu lineares Speedup;
- neue Heuristiken, die durch LazyOTF ermöglicht werden und sowohl Rücksetzverfahren als auch dynamische Information nutzen können: **Phasen-** und **Beschränkungs-Heuristiken** bieten Effizienz und dynamische Informationen; **Zielsuch-Heuristiken** und ein Rahmenwerk an Bedingungen bieten flexible Test-Selektion und Garantien für Vollständigkeit, Überdeckungen oder das Erreichen gewünschter Zielzustände. Gewichte und Komposition für die Umsetzung der Zielsuch-Heuristik werden eingeführt;
- die **Implementierung** und Integration in etablierte Werkzeuge: DFS_{FIFO} (in C implementiert, primär von Alfons Laarman) ist in LTSMIN integriert und verfügbar unter [URL:LTSmin]; LazyOTF (in Java implementiert, mit Hilfe von Felix Kutzner) ist in JTorX integriert und verfügbar unter [URL:JTorXwiki];
- Untersuchung des **praktischen Einsatzes**: In DFS_{FIFO} wurde die Benutzerfreundlichkeit durch die Modellierung von **Fortschritt mittels Transitionen** verbessert. Praktische **Experimente** (durchgeführt zusammen mit Alfons Laarman) belegen die theoretischen Vorteile von DFS_{FIFO} in der Durchführbarkeit. In LazyOTF wurde für bessere Benutzerfreundlichkeit das Konzept der **Testobjekte** entwickelt, sowie flexible Heuristiken. Auch hier belegen praktische **Experimente** (durchgeführt zusammen mit Felix Kutzner) die theoretischen Vorteile von LazyOTF in der Durchführbarkeit. Zusätzlich betrachten wir, wie LazyOTF und Verfeinerungen in der agilen Softwareentwicklung eingesetzt werden können;
- zwei weitere Einsichten auf der Metaebene: Mehrere Kapitel zeigen, dass der **passendste Abstraktionsgrad** zu effizienteren Algorithmen und Optimierungen führt. Zusätzlich zeigt eine Fallstudie zu **Testen von Software mit BMC**, wie der Testansatz Formale Methoden durchführbarer machen kann.

Short Contents With Dependencies



Contents

1. Introduction **1**
- 1.1. Motivation . . . 1
  - 1.1.1. Motivation for Software Quality . . . 1
  - 1.1.2. Motivation for Formal Methods . . . 2
  - 1.1.3. Motivation for More Feasible Formal Methods . . . 2
- 1.2. Overview . . . 4
  - 1.2.1. Goal of This Thesis . . . 4
  - 1.2.2. Achieving The Goal of Increased Feasibility . . . 4
  - 1.2.3. Contributions in Detail . . . 5
  - 1.2.4. Publications . . . 7
  - 1.2.5. Roadmap of This Thesis . . . 8
- 1.3. Conventions . . . 9

I. Introduction to Formal Verification and Testing **11**

2. Introduction to Testing **13**
- 2.1. Introduction . . . 13
- 2.2. Classification of Testing . . . 14
- 2.3. Artifacts . . . 15
- 2.4. Software Engineering . . . 17
- 2.5. Meaningfulness and Heuristics . . . 20
- 2.6. Summary . . . 24

3. Introduction to Formal Methods **25**
- 3.1. Introduction . . . 25
- 3.2. Propositional Logic . . . 26
  - 3.2.1. Syntax . . . 26
  - 3.2.2. Semantics . . . 26
  - 3.2.3. SAT Solvers . . . 27
  - 3.2.4. BDD-based Techniques . . . 27
- 3.3. First Order Logic . . . 28
  - 3.3.1. Syntax . . . 29
  - 3.3.2. Semantics . . . 30
  - 3.3.3. SMT Solvers . . . 30
- 3.4. Automata Theory . . . 32
  - 3.4.1. Transition Systems . . . 32
  - 3.4.2. Finite State Machines . . . 37
  - 3.4.3. System Specification Description Languages for Transition Systems 38

3.5. Improving Formal Methods by Parallelization 47
3.5.1. Introduction 47
3.5.2. Foundation of Parallel Computing 47
3.5.3. Roadmap 53
3.6. Improving Formal Methods by Lazy Techniques 54
3.6.1. Introduction 54
3.6.2. Current Lazy Techniques for Formal Methods 55
3.6.3. Advantages and Disadvantages of Lazy Techniques 57
3.6.4. Roadmap 58
3.7. Improving Formal Methods by Abstraction 59
3.7.1. Introduction 59
3.7.2. Roadmap 60
3.8. Conclusion 62
3.8.1. Summary 62
3.8.2. Contributions 62

II. Model Checking 63

4. Temporal Logics 65
4.1. Introduction 65
4.2. Behavioral Properties 65
4.2.1. Branching time properties 66
4.2.2. Linear time properties 67
4.3. Temporal Logics 68
4.3.1. Temporal Logics for Branching Time Properties 70
4.3.2. Temporal Logics for Linear Time Properties 74
4.4. Relationships 80
4.5. Conclusion 87
4.5.1. Summary 87
4.5.2. Contributions 87
4.5.3. Future 88

5. Model Checking 89
5.1. Introduction 89
5.2. Classifications of Model Checking 92
5.2.1. Explicit versus Implicit 92
5.2.2. On-the-fly versus Offline 93
5.2.3. Exhaustive versus Bounded 93
5.2.4. Finite versus Infinite 94
5.2.5. Practical Relationship between CTL and LTL MC 95
5.3. Model Checking Algorithms 97
5.3.1. CTL 97
5.3.2. On-the-fly LTL 97
5.3.3. CTL* 101

5.4. Reductions . . . 102
5.4.1. Partial Order Reduction . . . 103
5.4.2. Symbolic Techniques . . . 104
5.4.3. Other Reductions . . . 107
5.5. Tools . . . 108
5.5.1. SPIN . . . 108
5.5.2. LTSMIN . . . 109
5.5.3. DiVinE . . . 112
5.5.4. PRISM . . . 113
5.6. Conclusion . . . 114
5.6.1. Summary . . . 114
5.6.2. Contributions . . . 114
5.6.3. Future . . . 114

6. Explicit State Livelock Detection **115**
6.1. Introduction to Livelocks . . . 115
6.2. Introduction to Non-progress Cycle Checks . . . 117
6.3. Non-progress Cycle Checks via LTL . . . 118
6.3.1. Introduction . . . 118
6.3.2. Deficits . . . 118
6.4. A Better Non-progress Cycle Check . . . 120
6.4.1. $\text{DFS}_{\text{incremental}}$ . . . 122
6.4.2. DFS_{FIFO} . . . 125
6.4.3. Comparing LTL NPC Checks to DFS_{FIFO} . . . 129
6.5. Progress Transitions . . . 132
6.5.1. Semantics . . . 132
6.5.2. System Specification Descriptions . . . 133
6.5.3. DFS_{FIFO} . . . 134
6.6. Compatibility of DFS_{FIFO} with POR . . . 134
6.6.1. Motivation for Strong POR . . . 134
6.6.2. Correctness of DFS_{FIFO} with POR . . . 135
6.6.3. Comparing LTL NPC checks with POR to DFS_{FIFO} with POR . . . 136
6.7. Parallel DFS_{FIFO} . . . 137
6.7.1. The Algorithm $\text{PDFS}_{\text{FIFO}}$ . . . 138
6.7.2. Correctness of $\text{PDFS}_{\text{FIFO}}$ . . . 140
6.7.3. Implementation of $\text{PDFS}_{\text{FIFO}}$ . . . 141
6.7.4. Comparing Parallel LTL NPC Checks to $\text{PDFS}_{\text{FIFO}}$ . . . 144
6.8. Experiments . . . 144
6.8.1. DFS_{FIFO}'s Performance . . . 145
6.8.2. Strength of POR . . . 146
6.8.3. Parallel Runtime . . . 146
6.8.4. Parallel Memory Use . . . 147
6.8.5. Scalability of Parallelism Combined with POR . . . 149
6.8.6. On-the-flyness . . . 150
6.9. Conclusion . . . 151
6.9.1. Summary . . . 151

6.9.2. Contributions . . . 153
6.9.3. Future . . . 153

7. Testing with Software Bounded MC **155**
7.1. Introduction to Techniques for SBMC . . . 156
7.1.1. CBMC . . . 158
7.1.2. Complexities . . . 158
7.1.3. CBMC Optimization Heuristics . . . 159
7.2. Case Study . . . 160
7.2.1. Introduction . . . 161
7.2.2. The ESAWN Protocol . . . 161
7.2.3. TinyOS Platform and Model Abstraction . . . 163
7.2.4. Property Specification . . . 167
7.2.5. Verification Results . . . 169
7.3. Conclusion . . . 171
7.3.1. Summary . . . 171
7.3.2. Related Work . . . 172
7.3.3. Contributions . . . 177
7.3.4. Future Work . . . 177

III. Model-based Testing **181**

8. Input-output conformance theory **183**
8.1. Introduction . . . 183
8.1.1. Interface Abstraction . . . 184
8.1.2. Testing Hypothesis . . . 185
8.1.3. Overview . . . 186
8.2. Labeled Transition Systems for ioco . . . 187
8.2.1. Internal Transition . . . 188
8.2.2. Livelocks . . . 189
8.2.3. Quiescence . . . 190
8.2.4. Operations . . . 193
8.2.5. Nondeterminism . . . 194
8.2.6. Enabledness . . . 199
8.3. System Specifications . . . 200
8.4. *MOD* . . . 201
8.5. Implementation Relation ioco . . . 202
8.6. Test Cases . . . 203
8.7. Test Case Execution . . . 205
8.7.1. Test Case Execution on *MOD* . . . 205
8.7.2. Test Adapter . . . 207
8.8. Test Case Generation . . . 208
8.8.1. Introduction . . . 208
8.8.2. Nondeterministic Test Case Generation genTC . . . 210
8.8.3. Deterministic Test Suite Generation genTS . . . 214

8.8.4. Fairness and Coverage . . . 216
8.8.5. Exhaustiveness Threshold . . . 222
8.9. Conclusion . . . 224
8.9.1. Summary . . . 224
8.9.2. Contributions . . . 225
8.9.3. Future . . . 226

9. Variants of ioco **227**
9.1. Generalized *ioco* Relation and Derivates . . . 227
9.2. Underspecification and *uioco* . . . 228
9.3. Underspecification and *refines* . . . 229
9.3.1. Implementation . . . 233
9.3.2. Related Work . . . 234
9.4. Symbolic Transition Systems and *sioco* . . . 234
9.5. Conclusion . . . 235
9.5.1. Summary . . . 235
9.5.2. Contributions . . . 235
9.5.3. Future . . . 235

10. Introduction to Model-based Testing **237**
10.1. Introduction . . . 237
10.1.1. Motivation . . . 237
10.1.2. Model-based Testing . . . 237
10.1.3. Roadmap . . . 239
10.2. Classification of MBT . . . 239
10.2.1. The Kind of Properties . . . 240
10.2.2. The Kind of Test Selection . . . 240
10.2.3. Test Generation Technology . . . 240
10.2.4. The Kind of Test Cases . . . 241
10.2.5. Interplay between Test Generation And Test Execution . . . 242
10.2.6. The Kind of System Specification . . . 246
10.2.7. Level of Detail of The SUT's and Environment's Specification . . . 247
10.3. Tools . . . 247
10.3.1. TGV . . . 247
10.3.2. TorX . . . 248
10.3.3. JTorX . . . 249
10.3.4. Other Prominent Tools . . . 251
10.4. Summary . . . 252

11. Lazy On-the-fly MBT **253**
11.1. Introduction . . . 253
11.1.1. Motivation . . . 253
11.1.2. Lazy On-the-fly MBT . . . 253
11.1.3. Roadmap . . . 257
11.2. Classifications . . . 257
11.2.1. Test Generation Technology . . . 257

11.2.2. Properties . . . 257
11.2.3. Interplay Between Test Generation and Test Execution . . . 257
11.2.4. Test Selection . . . 259
11.2.5. Other Classifications . . . 262
11.3. Formalization . . . 263
11.3.1. Test Selection . . . 263
11.3.2. Interplay Between Test Generation And Test Execution . . . 264
11.3.3. LazyOTF Algorithm . . . 267
11.4. Related Work . . . 270
11.5. Parallelization . . . 272
11.5.1. Introduction . . . 272
11.5.2. Distributed LazyOTF . . . 275
11.6. Conclusion . . . 279
11.6.1. Summary . . . 279
11.6.2. Contributions . . . 280
11.6.3. Future . . . 280

12. Heuristics for Lazy On-the-fly MBT **283**
12.1. Introduction . . . 283
12.2. Phase Heuristics . . . 284
12.2.1. Inducing States . . . 285
12.2.2. Bound Settings and Heuristics . . . 285
12.3. Guidance Heuristics . . . 286
12.3.1. Introduction . . . 286
12.3.2. Exhaustiveness and Test Objectives . . . 291
12.3.3. Discharging Test Objectives . . . 295
12.3.4. Weight Heuristics . . . 297
12.3.5. Exhaustiveness and Coverage via Weight Heuristics . . . 302
12.3.6. Discharging Test Objectives Via Weight Heuristics . . . 304
12.3.7. Composition of Test Objectives . . . 310
12.3.8. Countermeasures for Unmet Provisos . . . 314
12.3.9. Related Work on Guidance via Weight Heuristics . . . 316
12.4. Optimizations . . . 317
12.4.1. Lazy Traversal Sub-phases . . . 318
12.4.2. Eager Micro-traversal Sub-phases . . . 318
12.4.3. Reproducibility . . . 319
12.4.4. More Dynamic Information for Bound Heuristics . . . 319
12.4.5. Quantifying Nondeterminism . . . 320
12.5. Conclusion . . . 321
12.5.1. Summary . . . 321
12.5.2. Contributions . . . 323
12.5.3. Future . . . 323

13. Implementation of Lazy On-the-fly MBT **325**
13.1. Introduction . . . 325

13.2. Core LazyOTF . . . 325
13.2.1. STSimulator . . . 325
13.2.2. Test Cases . . . 326
13.2.3. Heuristics . . . 326
13.2.4. Quality Assurance . . . 328
13.3. JTorX Integration . . . 329
13.3.1. Test Objective Management . . . 329
13.3.2. Configuration . . . 329
13.3.3. Coverage Criteria . . . 330
13.3.4. Test Execution . . . 331
13.3.5. Dynamic Feedback and Interaction . . . 331
13.4. Alternative Implementation with Symbolic Execution . . . 333
13.4.1. STSExplorer . . . 334
13.4.2. Related Work . . . 335
13.5. Optimizations . . . 335
13.5.1. Test Case Execution . . . 336
13.5.2. Test Case Generation . . . 336
13.6. Conclusion . . . 338
13.6.1. Summary . . . 338
13.6.2. Contributions . . . 338
13.6.3. Future . . . 339

14. Applications **341**
14.1. Introduction . . . 341
14.2. Agile Software Development and *refines* . . . 341
14.2.1. Introduction . . . 341
14.2.2. Agile Development . . . 342
14.2.3. Integrating MBT and Agile Development . . . 344
14.2.4. Summary . . . 348
14.3. Experiments . . . 348
14.3.1. Introduction . . . 348
14.3.2. Case Study and Its Specification . . . 349
14.3.3. Configuration . . . 350
14.3.4. Comparing MBT Approaches . . . 355
14.3.5. Comparing *aggWTCs* . . . 358
14.3.6. Comparing Bound Heuristics . . . 360
14.3.7. Distributed LazyOTF . . . 362
14.3.8. Real SUT . . . 366
14.3.9. Preliminary Further Experiments . . . 368
14.3.10. Conclusion . . . 369
14.4. Conclusion . . . 374
14.4.1. Summary . . . 374
14.4.2. Contributions . . . 375
14.4.3. Future . . . 375

15. Conclusion **377**
15.1. Summary . . . 377
15.1.1. Main Addressed Challenges . . . 377
15.1.2. Measurements . . . 378
15.2. Future . . . 379

A. Source Code **381**
A.1. Excerpt for SPIN . . . 381
A.2. Excerpt for CBMC . . . 381

B. Supplemental Figures And Tables **385**
B.1. Figures for the Application of MBT . . . 385
B.1.1. Abstract STS Specification for CodeMeter License Central . . . 385
B.1.2. General Performance Plots . . . 387
B.1.3. Exemplary Plot for CPU Time per Test Step . . . 389
B.1.4. Plots for Distributed LazyOTF . . . 390
B.1.5. Table Describing the Amount of Experiments . . . 391

Bibliography **395**

Index **463**

List of Figures

1.1. Most relevant contributions of this thesis 6

2.1. Artifacts of black-box testing 16
2.2. The V-model 19

3.1. Relationships between various transition systems 37
3.2. FSM 37
3.3. FSM $K_{fin}2FSM(\mathcal{S})$ of the unlabeled finite Kripke structure from Fig. 4.2 . 38

4.1. Classification of linear time properties 66
4.2. Kripke structure $\mathcal{S}$ showing LTL $\not\geq$ CTL 84
4.3. Classification of temporal logics 85
4.4. Kripke structure $\mathcal{S}_e$ showing Büchie $\not\geq$ ECTL* 87

5.1. Overview of on-the-fly LTL or Büchi MC 98
5.2. Overview of LTSmin 110

6.1. Büchi automaton for NPC checks via $\Diamond\,\Box$ np_ 118
6.2. NPC not traversed by the basic DFS 120
6.3. The found NPC does not contain s 122
6.4. Example for $\text{DFS}_{\text{incremental}}$ not traversing progress lazily 124
6.5. Fake progress cycle for $\mathcal{P} = S^{\mathcal{P}}$, but not for $\mathcal{P} = \mathfrak{T}^{\mathcal{P}}$ 132
6.6. Fake progress cycle by parallel composition 133
6.7. Progress state (left) and progress transition without changing PROMELA's semantics (right) 134
6.8. Early backtracking 144
6.9. Speedups of DFS, $\text{PDFS}_{\text{FIFO}}$ and CNDFS in LTSMIN(without giop due to oom), and piggyback in SPIN 148

7.1. ESAWN scenario of an aggregation tree with $w = 2$ witnesses. 162
7.2. Abstraction from TinyOS 166
7.3. Work-flow for our verification 166

8.1. Exemplary $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ and its transformations ($\delta_{\not\mu}$ for $\tau = \tau_{\not\mu}$, δ_u for $\tau = \tau_u$, δ for both) 192
8.2. Angelic and demonic completions of $\mathcal{S}$ from Subfig. 8.1a 200
8.3. assembleTC$(s, l2TC)$ (with L_U finite) 211
8.4. TCs generated by genTC for $\mathcal{S}$ from Subfig. 8.1a 214
8.5. Overview of the ioco theory and its artifacts 224

9.1. Exemplary refinement hierarchy for web services managing licenses 232

10.1. Architecture of JTorX 250

11.1. Exemplary phase of LazyOTF 255
11.2. Sequences describing the whole run of LazyOTF 266

12.1. assembleWTC($s, p2W, l2WTC$) (with L_U finite) 298
12.2. A TC (for $L_U = \{a, y\}, L_I = \{b\}$) and all $path2W(\pi^{full})$ 309
12.3. $\mathcal{S}$ with strictly monotonically increasing (path2W$_{o_1}$,path2W$_{o_2}$) towards the TOs and balanced aggWTCs resulting in no finitely monotonically increasing $w(\mathbb{W}_{p^{curr}})$ 315
12.4. Guarantees for exhaustiveness and TO discharge of LazyOTF by a framework of (TO-based and other) provisos 322

13.1. LazyOTF's main tab 330
13.2. LazyOTF's introspection window 332

14.1. Exemplary agile methods: XP and Scrum 342
14.2. MBT in our exemplary agile methods 346
14.3. Comparing the meaningfulness of the test cases generated by LazyOTF (Subfig. 14.3a and Subfig. 14.3b) and JTorX (Subfig. 14.3c and Subfig. 14.3d) for all our TOs, where Subfig. 14.3a is a lin-lin plot, all others are log-lin plots 356
14.4. Comparing the meaningfulness of the test cases generated by $\underline{aggWTCs}_{Max}$ (Subfig. 14.4a and Subfig. 14.4b) and $\underline{aggWTCs}_{MaxMax}$ (Subfig. 14.4c and Subfig. 14.4d) for our TOs (excluding o_{dec}) 359
14.5. Resource consumption for one traversalSubphase depending on a constant bound b 360
14.6. Comparing the meaningfulness and runtime of various (dynamic) bound heuristics of LazyOTF with $MAXPRT = 1$ (Subfig. 14.6a and Subfig. 14.6b) and $MAXPRT = 10$ (Subfig. 14.6c and Subfig. 14.6d) for $\ddot{o}_{S_{isol}}$ 362
14.7. Distributed LazyOTF for $\ddot{o}_{S_{isol}}$ 365
14.8. One LazyOTF on the real SUT $\mathbb{S}_{LC}$, version 2.0a, with fancy TO o_{RSML} (Subfig. 14.4a) and nonfancy TO o_{NFRSML} (Subfig. 14.4c) 367

15.1. Positioning of this thesis (according to our taxonomies) 377

B.1. Abstract specification of the License Central, described as STS $\mathcal{S}_{LC}$, with maximal portion $MAXPRT \in \mathbb{N}_{>0}$ 386
B.2. Memory requirements (of storing communication) for a long running experiment, depending on the bound (i.e., on the amount of communication) 387
B.3. Hardware Performance (CPU and WC time) for Concurrent Experiments 388
B.4. Scatter plot for parallel experiment with P=14 389
B.5. Mean CPU time per test step for o_{NFRSML} with bound $b = 5$ 390
B.6. Hazelcast's WC time for setup and cleanup 391
B.7. Distributed LazyOTF for o_{RSML}, $\ddot{o}_{cov}$, and their composition 392

List of Tables

3.1. Exemplary theories and their decidability 31

5.1. Comparison of CTL and LTL 96
5.2. Provisos on $ample(s)$ 105

6.1. Difference of DFS with POR and NPC checks via LTL MC with POR . . 135
6.2. Visibility provisos for DFS_{FIFO} 136
6.3. Visibility provisos for lazy full expansion 138
6.4. Number of states and runtimes (sec) of (sequential) DFS, DFS_{FIFO}, NDFS in SPIN and LTSMIN 146
6.5. POR (%) for $DFS^{\mathcal{T}}_{\text{FIFO}}$, $DFS^{\mathcal{S}}_{\text{FIFO}}$, DFS and NDFS in LTSMIN and SPIN . 147
6.6. Runtimes (sec) for the parallel algorithms: DFS, $\text{PDFS}_{\text{FIFO}}$ and CNDFS in LTSMIN, and PB in SPIN 147
6.7. Number of locally stored states for $\text{PDFS}_{\text{FIFO}}$ and CNDFS 149
6.8. POR and speedups for leader_{DKR} using $\text{PDFS}_{\text{FIFO}}$, OWCTY and CNDFS . 150
6.9. On-the-fly runtimes (sec) and counterexample lengths (states) for CNDFS and $\text{PDFS}_{\text{FIFO}}$ 151

7.1. Verification results for *STATUS* packets for a valid loop unwinding of 4. . 170

8.1. Upper estimates for $\mathcal{ET}$ depending on the fairness 225

9.1. Variants of *ioco* defined via $ioco_{\mathcal{F}}$ and $\mathcal{F}$ 227

12.1. Our *aggWTCs* for Fig. 12.2 with $L_U = \{a, y\}, L_I = \{b\}$ 309

13.1. Structure and size of the implementation of LazyOTF 338

B.1. The (min-max or sum) values for RSEM of t^{max}_{curr} (in %), sample size n, and total WC time t (in minutes or days) per experiment 394

List of Listings

5.1. Contract for model checking . . . 90
5.2. Contract for conditional model checking . . . 92
5.3. Nested DFS . . . 100

6.1. Contract for non-progress cycle checks . . . 117
6.2. Never claim for NPC checks via $\Diamond \Box$ np_ . . . 118
6.3. Typed $\text{DFS}_{\text{incremental}}$ . . . 123
6.4. Typed generic $\text{DFS}_{\text{prune,NPC}}$: DFS with pruning and NPC check . . . 124
6.5. Typed DFS_{FIFO} . . . 125
6.6. Contract for DFS_{FIFO}'s $\text{DFS}_{\text{prune,NPC}}$ . . . 128
6.7. Typed parallel DFS_{FIFO} ($\text{PDFS}_{\text{FIFO}}$) . . . 139
6.8. Work stealing for $\text{PDFS}_{\text{FIFO}}$ . . . 143

7.1. Exemplary loop unwinding for a bound of 2 . . . 156
7.2. Exemplary translation from C code to SSA . . . 157
7.3. Exemplary workarounds for CBMC . . . 165

8.1. Contract for test case generation . . . 209
8.2. Typed nondeterministic genTC($\mathcal{S}$, $\ddot{s}$) . . . 211
8.3. Typed deterministic genTS($\mathcal{S}$, $\ddot{s}$, b) . . . 215

10.1. Contracts for model-based testing for c . . . 238

11.1. Abstract LazyOTF($\mathcal{S}$, $\ddot{o}$, $\mathbb{S}$) algorithm . . . 254

12.1. Typed nondeterministic genWTC($\mathcal{S}$, π) . . . 298
12.2. Typed deterministic genWTS($\mathcal{S}$, π, b) . . . 300

13.1. Typed caching routine for genWTS($\mathcal{S}$, π, b) . . . 337

14.1. Exemplary quiescence error message . . . 368

A.1. PROMELA description for process1 and process2 in Fig. 6.6 . . . 381
A.2. Exemplary autostart function for the initialization without any nondeterminism . . . 382
A.3. Exemplary autostart function for the probabilistic concast with measurements and p being nondeterministic . . . 383

1. Introduction

This thesis improves the quality assurance of (safety-critical) software by increasing the feasibility of model checking and model-based testing. Hereby, larger systems can be analyzed, with lower effort, expertise and risk of missing a bug. This raises the return on investment of these formal methods, which is imperative to get adopted broadly in industry [Hunt, 2011; Weißleder et al., 2011; Faragó et al., 2013].

Roadmap. The next section motivates the importance of software quality, formal methods to assure it, and the need for more feasible methods. The following section described this thesis' goal of increasing the feasibility of FMs, how it is achieved, and the contributions in detail. The last section lists conventions of this thesis.

1.1. Motivation

1.1.1. Motivation for Software Quality

Software today is ubiquitous: embedded and distributed systems have taken over control in many domains, and many systems are interconnected. Consequently, the complexity of software has increased, as well as the necessity of its correctness and reliability. This necessity is strongest for the many applications that human health and lives now depend on, so called **safety-critical systems**. Many examples can be found in medicine, transportation and structural health monitoring. Correctness and reliability is also important for economical reasons: The total annual cost that software bugs cause due to incidents is in the tens of billions (estimated at $15 billion [Jones and Bonsignour, 2012] or at $30 billion [Tassey, 2002]). The total annual cost of fixing software bugs is at least as high (estimated at $30 billion [Tassey, 2002] or $312 billion [Britton et al., 2013]). This cost increases exponentially with the number of development phases containing the bug [Beck and Andres, 2004; McConnell, 2004; Weißleder et al., 2011; Shull et al., 2002]. Complexity has increased because software has to offer more features and interact more with its environment, such as other threads, components and systems. So we now have a **dilemma between software complexity and quality**: software should be released with fewer bugs, yet more and more intricate bugs occur during development.

Industry tries to cope with this dilemma and increased maintainability issues with a strong shift in software development towards addressing quality concerns with high priority, but often with limited success [Zhivich and Cunningham, 2009; Tassey, 2002]. Therefore, this thesis helps to improve the quality of software, especially for verifying and testing safety-critical software. Some techniques can easily be applied to hardware as well.

1.1.2. Motivation for Formal Methods

Industry often tries to resolve this dilemma between software complexity and quality by increasing the number of conventional test cases, which are implemented manually – or even by doing purely manual, exploratory testing. But this no longer scales in the situation described above: The number of required test cases explodes, forcing testing to consume over 50% of the development costs for embedded software [Brook, 2004], and 60% to 70% for safety-critical software [Baker and Habli, 2013]. Still the bug detection rate is too low to prevent frequent bugs and incidents, especially for safety-critical software [Zhivich and Cunningham, 2009; Wong et al., 2010; Faragó et al., 2014; URL:REDHATCVE].

Thus the dilemma needs to be resolved differently. Using formal methods is an approach that has gained strong interest in industry [Woodcock et al., 2009; DO-333 Plenary, 2011; ED-218 Plenary, 2012]. **Formal methods** (**FMs**) process formal descriptions, i.e., semantically unambiguous specifications in a formal language to describe the requirements of the software system, i.e., its intended behavior. Often, functional requirements are considered, which specify what the system should do, i.e., its functionality. But sometimes, non-functional requirements are considered, which state how to perform the intended functionality. These formal specifications can be used for unambiguous human communication and can be processed with mathematical rigor to develop or analyze the system. Thus FMs are the mathematics of software, and transform software development into true engineering [Holloway, 1997; Groote and Mousavi, 2014]. Analyses that do not execute the system are called **static**, those that do execute it are called **dynamic**.

Formal specifications are usually more concise than non-trivial test suites, leading to easier and more flexible maintenance for these formal specifications than for test cases, which is important since requirements can often change [Weißleder, 2009; Weißleder et al., 2011; Hunt, 2011; Mlynarski et al., 2012]. One major application of FMs is checking the system's **correctness**, i.e., that it conforms to its specification. Therefore, formal specifications help in early bug detection, i.e., in avoiding high costs due to bugs being removed only in later development phases. Semantic unambiguousness allows automation, which reduces human labor, is less prone to human errors, and can usually investigate more relevant cases (such as paths or test cases) in less time, leading to higher coverage and ultimately higher correctness and quality compared to classical (i.e., conventional) testing [Holloway, 1997; Kneuper, 1997; Holzmann, 2001; Woodcock et al., 2009; Naik and Tripathy, 2011; Dross et al., 2011; Jeffery et al., 2015]. Hence this thesis focuses on fully automatic FMs: model checking and model-based testing. They (and their feasibility) are described in the next subsection.

1.1.3. Motivation for More Feasible Formal Methods

Depending on the specification, the mathematical method, and to what extent they are applied, various formal methods exist, listed here from heavyweight to lightweight:

- the most **heavyweight** FMs use **deductive verification** to provide a mathematical proof that the system is correct. Their specification languages are usually very expressive, e.g., they can express systems with infinite state space. For instance, the tool KeY [URL:KeY; Beckert et al., 2007] statically analyzes Java source code

and design-by-contract specifications in the Java Modeling Language (JML) [Leavens et al., 2008] to show in a rigorous proof the absence of bugs in relation to the specification (similar to Hoare [1969]). The cost of heavyweight FMs, however, is high: high expertise, time and computer resources are required. To reduce the costly requirement of human expertise and time for writing heavyweight specifications and for interactive theorem proving, research tries to increase heavyweight FMs' automation [Hutter, 2003]. But a sound and complete push button solution cannot exist in general since the problem whether an infinite system is correct is undecidable [Church, 1936; Turing, 1936; Kleene, 1967]. Furthermore, increasing the degree of automation quickly causes search-space explosion [Schumann, 2001]. Unfortunately, a low degree of automation restricts the practicality and impact of deductive verification [DO178C Plenary, 2011] – still successful case studies exit [Schmitt and Tonin, 2007]. The restricted practicality is stated hyperbolically by Benjamin Pierce: "Formal methods will never have a significant impact until they can be used by people that don't understand them" [Pierce, 2002];

- **model checking** (**MC**), e.g., with SPIN or LTSmin (cf. Chapter 5), diminishes these problems by reducing expressiveness: MC usually operates only on finite systems, i.e., on finite state spaces, and verifies temporal properties related to protocols and parallel systems. MC's expressiveness is sufficient for many industrial applications, e.g., for chip and protocol design and embedded software [Gnesi and Margaria, 2012; Margaria and Steffen, 2012]. For these applications, MC is often a more feasible approach than deductive verification [Schumann, 2001]. Sound and complete verification is performed fully automatically, but expertise is still required to come up with good abstractions that avoid the severe state space explosion;
- **software bounded model checking** (**SBMC**) (e.g., by CBMC or LLBMC, cf. Chapter 7) and more generally bounded model checking (BMC) are special cases of model checking that explore the state space only up to a given depth. They are sound and still complete if the applied bound is not smaller than a completeness threshold, but the focus is more on bug finding than on proving that the system is correct. Though less severe, state space explosion still often causes problems;
- **model-based testing** (e.g., by JTorX, cf. Chapter 10) automatically generates test cases from a specification (by using it for test drivers to create the test cases' inputs, for test oracles to determine the outcome of test cases, and for guidance to select test cases). Depending on the approach, state space explosion rarely or never hinders practical application of model-based testing with relevant results. The test cases inspect the actual system (not only a model) since they are executed on that system. But due to state space explosion, complete verification is usually practically impossible. Model-based testing is a suitable advent of FMs into certified, safety-critical software development in industry [Peleska, 2013]. Other terms exist for this kind of formal method, e.g., property-based testing [Fink and Bishop, 1997], which focuses on general properties not specific to one program, e.g., array bounds, race conditions, idempotence, and reflexivity. It often adds static source code analysis techniques like slicing to improve the process, but not always (e.g., not in the tool QuickCheck [Claessen and Hughes, 2000]);
- the following approaches can be considered the most **lightweight** formal methods [Woodcock et al., 2009; Weiß, 2010], or already excluded from the field of formal

methods [Kreiker et al., 2011]: simple static analysis by tools like Findbugs [Hovemeyer and Pugh, 2004], pluggable type checking (e.g., Checker Framework, cf. Subsec. 13.2.4, [Papi et al., 2008; Ernst et al., 2011]) or even static types of statically typed languages, and finally runtime assertion checking, e.g., via lightweight variants of design-by-contract [Meyer, 1997], i.e., via preconditions, postconditions, and invariants.

In spite of their advantages, formal methods have been adopted meagerly in industry due to issues in feasibility [Bienmüller et al., 2000; Kreiker et al., 2011; Hunt, 2011; Gnesi and Margaria, 2012]. Low practicality leads to statements like "testing will always be part of the certification process and formal methods do not get adopted broadly in industry" [Hunt, 2011]. But formal methods might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification [DO178C Plenary, 2011; Dross et al., 2011]. This is also reflected in the **certification** of safety-critical software and their tool qualification: Many safety standards recommend formal methods for verification of systems with the highest Safety Integrity Level, e.g., for avionics [DO178C Plenary, 2011; DO-333 Plenary, 2011; ED-218 Plenary, 2012; DO-278 Plenary, 2002; DO-330 Plenary, 2011] and standards based on the IEC 61508 framework for functional safety [IEC 61508 Plenary, 2010], like ISO 26262 for automotive [URL:ISO26262; Hillenbrand, 2011] and CENELEC EN 50128 for rail transport [Plenary, 2011].

Due to these issues in feasibility, industry is currently adopting mainly lightweight FMs in narrow domains such as safety-critical and embedded software. If FMs will become more feasible, they will be more usable (e.g., with faster and more informative feedback) and able to verify less abstract (e.g., on the code level) and larger problems (e.g., realistic data or number of instances). Then industry will likely adopt FMs that are heavier, and on a broader scale. This is the goal of this thesis, as described next.

1.2. Overview

1.2.1. Goal of This Thesis

The goal of this thesis is to increase the **feasibility** of FMs: mainly by reducing their runtime and memory requirements, so that more problems (larger, less abstract, or new ones) can be handled, but also by increasing their expressiveness and usability. For practicality, this thesis focuses on relatively lightweight, automated FMs: model checking (MC) and model-based testing (MBT). (Fig. 15.1 on page 377 gives a more detailed positioning of this thesis.)

On the meta level, the thesis investigates what methods achieve these improvements.

1.2.2. Achieving The Goal of Increased Feasibility

Since automation increases the applicability of formal methods in industry, we do not investigate interactive methods. For liveness checks, this thesis considers MC; for safety checks, testing is also applicable, so MC and MBT are considered.

Since the state space explosion problem is severe for MC, we developed the MC algorithms DFS_{FIFO}, which increases feasibility by specializing on the important liveness

property of livelock freedom and by using lazy techniques. DFS_{FIFO}'s feasibility is further improved by optimizations: stronger partial order reduction, parallelization with almost linear speedup, and modeling progress via transitions instead of states for better usability and higher performance. Our overall result is four to five orders of magnitude better than all currently established livelock detection techniques. But due to the state space explosion, yet larger problem instances remain infeasible. Therefore, we investigate software bounded MC to check safety properties of a large C program of practical size and complexity (implementing a wireless sensor network). This case study shows that state space explosion still renders large practical applications infeasible. Feasibility is increased by abstraction and by adopting a more lightweight method: integrating the testing approach, which also motivates model-based testing.

Model-based testing (MBT) combines MC algorithms with the testing approach to check safety properties. Unfortunately, exhaustiveness, full coverage and specific objectives are often still infeasible for large applications. Our extensions to the ioco theory slightly improve the situation: our deterministic, bounded test generation algorithm and exhaustiveness thresholds (similar to completeness thresholds in bounded MC) reduce the redundancy and size of the generated test suites. The exhaustiveness threshold often needs to be infeasibly large, but our extensions help for inexhaustive MBT and for implementing a new MBT approach, called LazyOTF: It integrates offline MBT and on-the-fly MBT using lazy techniques. Unlike offline MBT, state space explosion does not cause infeasible time and space requirements for LazyOTF, i.e., does not hinder test execution. Compared to on-the-fly MBT, LazyOTF has better guidance, so achieving high coverage and specific objectives becomes feasible. To further improve feasibility, we derive new heuristics that make use of both backtracking and dynamic information. Usability is provided by designing test objectives, which allow composition and easy configuration of the heuristics. To further optimize LazyOTF's performance, parallelization with almost linear speedup and several technical optimizations are introduced. The result achieves coverage and specific objectives several orders of magnitude better than classical MBT, making MBT also feasible for large, nondeterministic systems.

In summary, this thesis focuses on the lightweight formal method of MBT for checking safety properties, and derives a new and more feasible approach. For liveness properties, dynamic testing is impossible, so feasibility is increased by specializing on an important class of properties, livelock freedom, and deriving a more feasible model checking algorithm for it. All mentioned improvements are substantiated by experiments (cf. Subsec. 15.1.2).

1.2.3. Contributions in Detail

Fig. 1.1 depicts the most relevant contributions of this thesis, which are described in this subsection. More details and further, minor contributions can be found in the subsections entitled "Contribution" at the end of most chapters.

The main contribution in MC is the design, implementation and analysis of the algorithm $\mathbf{DFS_{FIFO}}$ to increase the feasibility of livelock detection, via on-the-fly model checking: by traversing progress lazily, DFS_{FIFO} simultaneously performs state space exploration and non-progress cycle checks in one pass, increasing feasibility directly as well as by being more amendable to optimizations (see further contributions). The result

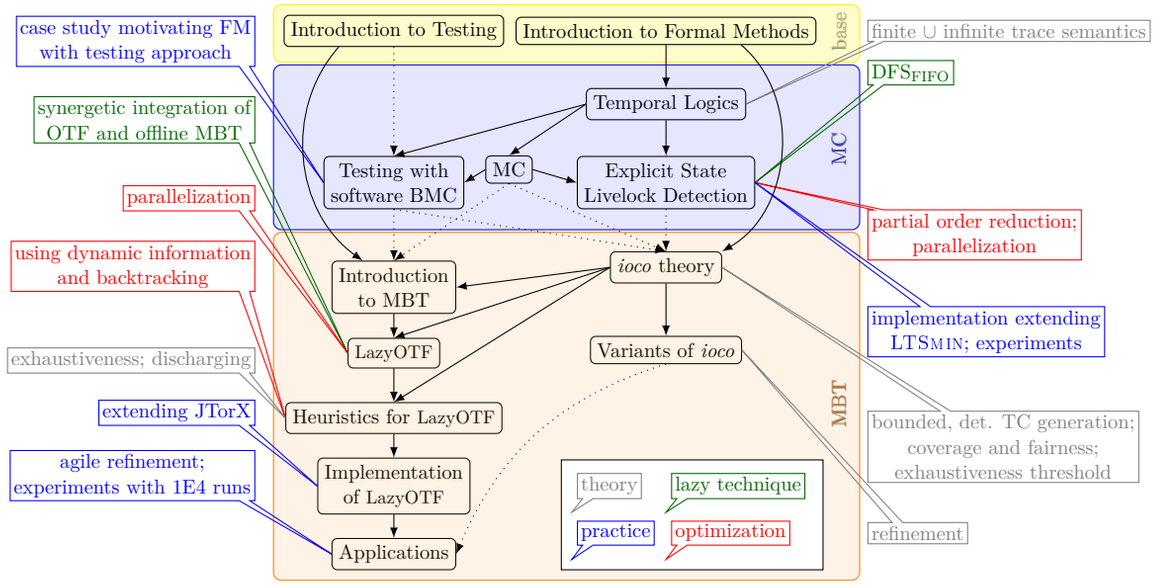


Figure 1.1.: Most relevant contributions of this thesis

is the state of the art in livelock detection.

The main contribution in MBT is the design, implementation and analysis of the approach and corresponding algorithm **lazy on-the-fly MBT** (**LazyOTF**) to improve the guidance of on-the-fly model-based testing: LazyOTF synergetically integrates both offline and on-the-fly MBT, yielding good guidance to efficiently select and execute meaningful test cases while still being able to process large and nondeterministic systems, rendering MBT feasible for those systems.

On the meta level, the main contributions are **lazy techniques** and the insight where and how they can improve formal methods and their optimizations and application.

Further contributions of this thesis are:

- **generalized theoretical foundations** in automata theoretic formalisms that both MC and MBT are based upon, in temporal logics, and in the ioco theory. Additions to the ioco theory include an extensive testing hypothesis, a refinement relation, several fairness constraints and coverage criteria, an abstract deterministic test case generation algorithm, exhaustiveness thresholds, and their relation to fairness and coverage;
- optimizations and adaptions in **partial order reduction** (POR), which can be strengthened for DFS_{FIFO} compared to general LTL MC (by omitting cycle provisos and enabling progress transitions);
- optimizations via **parallelization**, facilitated by our lazy techniques: a **multi-threaded DFS_{FIFO}** ($\text{PDFS}_{\text{FIFO}}$, in collaboration with Alfons Laarman) with almost linear speedup, and a **distributed LazyOTF** with (super-)linear speedup of meaningful test execution and almost linear speedup overall;
- new **heuristics**, enabled by LazyOTF and making use of dynamic information and backtracking: **phase** (and **bound**) **heuristics** for efficiency and dynamic in-

formation, and **guidance heuristics** with a provisos framework for flexible test selection, to guarantee exhaustiveness, coverage or discharging. Weights and composition are introduced, to implement the guidance heuristics;

- **implementations** and integration into established tools: of LazyOTF (implementation in Java, with help from Felix Kutzner, and an integration into JTorX), available at [URL:JTorXwiki]; and of DFS_{FIFO} (implemented mainly by Alfons Laarman, in C, with an integration into LTSMIN), available at [URL:LTSmin];
- considerations for **practical applications**: For DFS_{FIFO}, usability was improved by modeling **progress via transitions** instead of states, and **experiments** (conducted together with Alfons Laarman) practically substantiate all theoretical benefits in feasibility. For LazyOTF, usability was improved by introducing **test objectives** and heuristics; again **experiments** (conducted together with Felix Kutzner) practically substantiate the theoretical benefits in feasibility. Furthermore, an outlook is given on how LazyOTF and refinement can be applied in agile software development;
- two further insights on the meta level: Firstly, a case study in **testing with software bounded model checking** shows how the testing approach can make FMs more lightweight. Secondly, multiple chapters show how choosing the **right level of abstraction** can lead to more efficient algorithms and optimizations, e.g., finite $\cup$ infinite trace semantics, specialization on livelocks for DFS_{FIFO}, and generalized interplay between test generation and test execution for LazyOTF.

1.2.4. Publications

The following publications from the author of this thesis have directly contributed to this thesis, and are subsumed by it:

- [Laarman and Faragó, 2013]: Alfons Laarman and David Faragó. Improved on-the-fly livelock detection: Combining partial order reduction and parallelism for DFS-FIFO. **Proceedings of the 5th NASA Formal Methods Symposium (NFM 2013), NASA Ames Research Center, CA, USA, May 14-16, 2013**. LNCS, Springer, 2013.
- [Faragó, 2011]: David Faragó. Nondeterministic coverage metrics as key performance indicator for model- and value-based testing. **In 31. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV)**, Softwaretechnik-Trends, 2011.
- [Faragó, 2010]: David Faragó. Improved underspecification for model-based testing in agile development. In Stefan Gruner and Bernhard Rumpe, editors,**FM+AM 2010 - Second International Workshop on Formal Methods and Agile Methods, 17 September 2010, Pisa (Italy)**, volume 179 of LNI, pages 63–78. GI, 2010. ISBN 978-3-88579-273-4.
- [Faragó, 2010b]:David Faragó. Coverage criteria for nondeterministic systems. **testing experience, The Magazine for Professional Testers**, pages 104–106, September 2010b. ISSN 1866-5705.
- [Werner and Faragó, 2010]: Frank Werner and David Faragó. Correctness of sensor network applications by software bounded model checking. In **Formal Methods for Industrial Critical Systems - 15th International Workshop, FMICS 2010, Antwerp, Belgium, September 20-21, 2010**. Proceedings, LNCS, pages 115–131. Springer, 2010.
- [Faragó, 2010a]: David Faragó. Model-based testing in agile software development. In **30. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV)**, Softwaretechnik-Trends, 2010a.
- [Faragó and Schmitt, 2009]: David Faragó and Peter H. Schmitt. Improving non-progress cycle checks. In Corina S. Pasareanu, editor, **Model Checking Software, 16th In-**

ternational SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings, volume 5578 of LNCS, pages 50–67. Springer, June 2009.

The following publications from the author of this thesis have only indirectly contributed to this thesis, and are not contained in it:

- [Brandes et al., 2015]: Christian Brandes, Benedikt Eberhardinger, David Faragó, Mario Friske, Baris Güldali, Andrej Pietschker. Drei Methoden, ein Ziel: Testautomatisierung mit BDD, MBT und KDT im Vergleich. **38. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV)**, Softwaretechnik-Trends, 2015.
- [Faragó et al., 2014]: David Faragó, Florian Merz, and Carsten Sinz. Automatic heavyweight static analysis tools for finding bugs in safety-critical embedded C/C++ code. **36. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV)**, Softwaretechnik-Trends, 2014.
- [Faragó et al., 2013]: David Faragó, Stephan Weißleder, Baris Güldali, Michael Mlynarski, Arne-Michael Törsel, and Christian Brandes. Wirtschaftlichkeitsberechnung für MBT: Wann sich modellbasiertes Testen lohnt. **OBJEKTspektrum, 4:32–39**, 2013.
- [Weißleder et al., 2011]: Stephan Weißleder, Baris Güldali, Michael Mlynarski, Arne-Michael Törsel, David Faragó, Florian Prester, and Mario Winter. Modellbasiertes Testen: Hype oder Realität? **OBJEKTspektrum, 6:59–65, Oktober**, 2011.

The following theses were supervised by the author of this thesis, where [Larysch, 2012; Kutzner, 2014] directly contributed to this thesis (as described in Sec. 13.4 and Sec. 14.3):

- [Liu, 2015]: Man Liu. *SBMC-Based Testcase Generation.* Master's thesis, Karlsruhe Institute of Technology, 2015.
- [Weber, 2014]: Andreas Weber. *Modularization and Optimization for the SBMC-Algorithm of LLBMC.* Bachelor's thesis, Karlsruhe Institute of Technology, 2014.
- [Kutzner, 2014]: Felix Kutzner. *A case study for lazy on-the-fly model-based testing.* Bachelor's thesis, Karlsruhe Institute of Technology, 2014.
- [Larysch, 2012]: Florian Larysch. *Improved constraint specification and solving for lazy on-the-fly.* Bachelor's thesis, Karlsruhe Institute of Technology, 2012.
- [Pascanu, 2010]: Alexander Pascanu. *Eine Fallstudie zu modellbasiertem Testen von Webservices.* Master's thesis, Universität Karlsruhe, 2010.

1.2.5. Roadmap of This Thesis

The figure on page ix gives a short overview of the chapters of this thesis, and depicts their dependencies.

In more detail, Part I defines the common theoretical foundation in a generalized way, so that it can be used as basis for both model checking and model-based testing: Chapter 2 describes testing and its classification (cf. Sec. 2.2), artifacts (cf. Sec. 2.3), processes (cf. Sec. 2.4), and techniques (cf. Sec. 2.5). Chapter 3 introduces formal methods: Propositional, resp. first order logic, (cf. Sec. 3.2, resp. Sec. 3.3), automata theory (cf. Sec. 3.4), and lazy techniques (cf. 3.6) are the basis used throughout this thesis. Furthermore, improving FMs via parallelization (cf. Sec. 3.5) and abstraction (cf. Sec. 3.7) is explained.

Part II about MC and starts with Chapter 4 about temporal logics. It covers generalized behavioral properties (cf. Sec. 4.2), temporal logics (cf. Sec. 4.3) and their relationships (cf. Sec. 4.4). Chapter 5 about MC introduces various classifications (cf. Sec. 5.2), algorithms (cf. Sec. 5.3), reductions (cf. Sec. 5.4), and tools (cf. Sec. 5.5). Livelock detection is considered thoroughly in Chapter 6, where the state of the art is

introduced (cf. Sec. 6.2, Sec. 6.3) and the more efficient algorithm DFS_{FIFO} is devised (cf. Sec. 6.4). Its application with progress transitions (cf. Sec. 6.5), improvements to partial order reduction (cf. Sec. 6.6), and parallelization of the algorithm (cf. Sec. 6.7) are investigated. Finally, experiments are presented (cf. Sec. 6.8). Chapter 7 investigates software bounded model checking (cf. Sec. 7.1) and its limitations with the help of a practical case study (cf. Sec. 7.2), which shows how software bounded model checking can be made more feasible using a testing approach.

Part III about MBT starts with Chapter 8, which lays the theoretical foundations for MBT by introducing the ioco theory: Firstly, an extensive interface abstraction and testing hypothesis is given (cf. Sec. 8.1). Then various structures are defined (cf. Sec. 8.2, Sec. 8.3, Sec. 8.4), which are used by the *ioco* relation (cf. Sec. 8.5). Thereafter, test cases (cf. Sec. 8.6) and their execution (cf. Sec. 8.7) are described. Finally, the classical test case generation algorithm, as well as a new, deterministic variant are devised, for which we introduce various fairness constraints, coverage criteria and exhaustiveness thresholds for bounded yet exhaustive test suites (cf. Sec. 8.8). Chapter 9 extends the ioco theory by generalizing it (cf. Sec. 9.1), introducing variants for underspecification (cf. Sec. 9.2), for refinement (cf. Sec. 9.3), and for symbolic transition systems (cf. 9.4). Chapter 10 introduces model-based testing (cf. Sec. 10.1), an extended taxonomy (cf. Sec. 10.2), and tools (cf. Sec. 10.3). Chapter 11 introduces a new algorithm, LazyOTF, and the structures it uses (cf. Sec. 11.1), classifies it according to our MBT taxonomy (cf. Sec. 11.2), and then formalizes (cf. Sec. 11.3) and parallelizes (cf. Sec. 11.5) it. Chapter 12 introduces heuristics for LazyOTF: phase heuristics (cf. Sec. 12.2) via inducing states and bound heuristics, as well as guidance heuristics (cf. Sec. 12.3), which use a provisos framework to guarantee exhaustiveness, coverage or discharging specific objectives, and are implemented via weights and composition. Finally, optimizations for these heuristics are given. Chapter 13 roughly describes the implementation of LazyOTF, covering the core (cf. Sec. 13.2), the JTorX integration (cf. 13.3), a proof of concept implementation with symbolic execution (cf. Sec. 13.4), and finally optimizations (cf. Sec. 13.5). Chapter 14 gives an outlook on how LazyOTF and *refines* can be applied in agile software development (cf. Sec. 14.2), and covers experiments of LazyOTF (cf. Sec. 14.3) to compare heuristics settings as well as LazyOTF to OTF, and to measure parallel speedup and the application on industrial web services. Chapter 15 concludes this thesis.

1.3. Conventions

$\mathbb{B}$ is the set of the **Boolean** values **true** and **false**, i.e., the **truth values**. Sometimes **true** is identified with **1** and **false** with **0**. Therefore, a function $f : M \to \mathbb{B}$ with arbitrary M has the **support** $\boldsymbol{supp(f(\cdot))} := \{m \in M | f(m) = \textbf{true}\}$. As it will not cause ambiguity in this thesis, we use **true**, **false** as both syntactical and semantical elements. Syntactical **true** can be defined as $p \vee \neg p$ for any propositional variable p, in all our logics, **false** as $\neg$**true**.

Since the definition whether $0 \in \mathbb{N}$ varies in literature, we write $\mathbb{N}_{\geq \mathbf{0}}$ for $\mathbb{N} \cup \{0\}$ and $\mathbb{N}_{>\mathbf{0}}$ for $\mathbb{N} \setminus \{0\}$ whenever it is relevant.

Since we also deal with infinite cases, we define a **countable set** to be a finite or countably infinite set and additionally use $\boldsymbol{\omega} := \mathbb{N}_{\geq 0}$, the smallest infinite ordinal, and the

set $\boldsymbol{\omega + 1} := \mathbb{N}_{\geq 0} \cup \{\omega\}$ [Cantor, 1897, §15]. Note that for all $n \in \mathbb{N}_{>0} : \omega = n + \omega \lneq \omega + n$.

For $i, j \in \omega + 1$, we denote closed and open intervals over $\omega + 1$ by closed and open brackets, respectively. Thus $\boldsymbol{[i, \ldots, j]} = \{i, \ldots, j\} (= \varnothing \text{ iff } j < i)$, $\boldsymbol{(i, \ldots, j)} = \{n \in \mathbb{N}_{\geq 0} | i < n < j\}$ $(= \varnothing \text{ iff } -1 + j < i + 1)$ and $[0, \ldots, 1 + \omega) = \mathbb{N}_{\geq 0}$. With this, we can write $[0, \ldots, 1 + i)$ for both finite intervals $(i \in \omega)$ and for $\mathbb{N}_{\geq 0}$ $(i = \omega)$.

This thesis uses the **Kleene closure operators** [Hopcroft and J.D. Ullman, 1979]: Let S be a set, $\boldsymbol{\epsilon} \notin S$ the **empty string**, and concatenation noted as product, i.e., $S^0 = \{\epsilon\}$ and $\forall i \in \mathbb{N}_{>0} : S^i = \{s \text{ concatenated } s' | s \in S^{i-1}, s' \in S\}$, then

$$S^+ := \bigcup_{i \in \mathbb{N}_{>0}} S^i; \qquad S^* := \bigcup_{i \in \mathbb{N}_{\geq 0}} S^i,$$

whereas S^ω only contains the infinite sequences (i.e., $S^\omega \neq \bigcup_{i \in \omega + 1} S^i$).

Since certain meanings can be grasped better if they are expressed concisely, this thesis uses:

- male pronouns for indeterminate gender;
- the abbreviation **"iff"** for "if and only if" in continuous text and **"⇔"** otherwise. Concise phrases are also chosen for other logical connectives [Kleene, 1967];
- the abbreviations **"f.a."** (resp. **"ex."**) for "for all" (resp. "exists") in fluent text and "∀" (resp. "∃") otherwise, to set it apart from the quantifiers "∀" (resp. "∃") on the logical level;
- the notation **"$(x_i)_{i \in I}$"** for the sequence of elements x_i with i ranging over I. If I is clear from the context, **"$(x_i)_i$"** is used for short. If I is not given explicitly, then without loss of generality (**wlog**) $I = [0, \ldots, 1 + j)$ with $j \in \omega + 1$;
- the notation $f(\cdot, c)$ for the function that takes x and returns $f(x, c)$ (i.e., $x \mapsto f(x, c)$ or $\lambda x.f(x, c)$), with c being constant.

Since consistency also increases understandability, this thesis follows some (naming) conventions:

- We will often generalize from one instance to a set of instances. To indicate this generalization and close relation, we name a variable for a set similar to the variable for an instance by using the trema symbol "¨" (e.g., from a state s to a superstate $\ddot{s}$, from a test case $\mathbb{T}$ to a test suite $\ddot{\mathbb{T}}$);
- the listings for code and algorithms contain type information where this adds understandability – even in pseudocode. Pseudocode examples where all values are typed are named "typed" in their caption. Types are always printed in blue;
- all definitions (and their page number in the index) are printed **bold**.

Furthermore, this thesis follows the Chicago Manual of Style [of Chicago Press, 2010] as much as possible.

Finally, this thesis strongly structures the text: All chapters except this and the conclusion contain a roadmap to describe the structure that follows, and a summary at the end. Most chapters also contain their contributions and future work at the end. Within sections, we often add markings for paragraphs, e.g., for examples and notes, and we set off definitions and statements against normal text flow iff they are important at multiple locations of this thesis. Forward references in parentheses help the reader connect all the dots, but most forward references are not required to comprehend the subject at hand.

Part I.

Introduction to Formal Verification and Testing

2. Introduction to Testing

2.1. Introduction

Many definitions of **software testing** (**testing** for short) exist: We define it in a wide sense (similarly to [Kaner et al., 1993]) as the process of assessing through exemplary execution (called dynamic testing) or exemplary analysis (called static testing) whether the system is correct. Besides failure detection, testing can aid risk management and increases the confidence in the correctness of the system under test.

This general definition, as well as many concepts from this thesis, are also applicable to other systems besides software, e.g., hardware. A relevant domain for testing and formal methods are systems where both software and hardware need to be considered, e.g., **embedded systems**. In an embedded system, a computer is embedded in a larger mechanical or electrical system, performing particular tasks to control the system. Embedded systems that process physical input and output have a strong link between computational and physical elements, and are thus often called **cyber-physical system**. Checking embedded software becomes more and more important since there is a shift in industry from electrical and mechanical to embedded systems.

Since executions are always finite, dynamic testing can experimentally check safety properties, but no liveness properties (cf. Subsec. 4.2.2 and Subsec. 11.2.4). But even for safety properties, the number of experiments needed for an exhaustive check is often infeasibly large (cf. Sec. 8.8).

Test engineers are the engineers responsible for ensuring high quality of the system under test, especially its functional correctness and reliability [Naik and Tripathy, 2011]. They still often prefer dynamic testing over static testing via formal verification in practice [Hunt, 2011], even for safety-critical software: even though dynamic testing is rarely exhaustive, it is usually more suitable for current accreditation [Peleska, 2013], investigates the real implementation [Fraser et al., 2009], and is more lightweight and simpler to apply (cf. Chapter 1). For these reasons, testing is still the primary instrument in industry even though exhaustive exploration is infeasible (i.e., "program testing can be used to show the presence of bugs, but never to show their absence!" Dijkstra [1970]).

Since testing is a wide field spanning multiple domains, different testing terminologies evolved. To consolidate them, standards and glossaries were created [Group, 1987; URL:ISO29119; 829WG, 2008; ISO Information Technology, 1992; URL:ISO26262; Board), 2012; URL:ETSIglossaryHP; URL:BS79251]. Unfortunately, they are not fully consistent; but with their introduction, terms differ much less. Thus the definitions in this chapter are based on these standards.

Roadmap. Sec. 2.2 shows several dimensions of the wide field of testing, and positions the testing methods covered in this thesis. Sec. 2.3 introduced the testing artifacts required in this thesis. Sec. 2.4 briefly embeds testing in software engineering, depicting the

process, test case generation and test case execution. Sec. 2.5 describes meaningfulness of tests and heuristics to measure and increase meaningfulness. Finally, Sec. 2.6 gives a summary.

2.2. Classification of Testing

Since testing is such a wide field, we introduce a **taxonomy of testing** to span the field and to enable positioning of this thesis's testing methods in the field.

These aspects are taken from various sources, most of them from the standards mentioned above and from [Weißleder, 2009]:

- static versus dynamic testing: **dynamic testing** assesses the system through exemplary execution, whereas **static testing** assesses the system through exemplary analysis [Myers, 2004; Kaner et al., 1993]. In this thesis, the term "testing" always refers to dynamic testing, unless stated otherwise, e.g., by calling it testing with software bounded model checking (cf. Chapter 7);
- the disclosure of the system: In **black-box testing**, the system under test is a black-box, i.e., its internals are not disclosed to the test engineer. So his only information is the inputs he gave to the black-box and the outputs he observed from the black-box (or absence thereof, cf. Sec. 8.1). This approach is the most general and investigated in this thesis for model-based testing (cf. Part III). In **white-box testing** (aka **glass-box testing**), the test engineer has full insight into the system and its source code, which he can use for testing, e.g., for test case generation and code coverage in dynamic testing, but also for static testing, e.g., for **static code analysis** (**static analysis** for short), which we define as automated static testing, excluding manual variants like reviews (cf. Chapter 7);
- the kind of testing: whether functional or non-functional requirements are tested, resulting in **functional testing** respectively **non-functional testing** (see Sec. 2.4 for details). Similar to most other work [Ghidella and Mosterman, 2005; Whalen et al., 2006; Fraser and Wotawa, 2006; Utting and Legeard, 2007; Langer and Tautschnig, 2008; Lackner and Schlingloff, 2012], this thesis mainly deals with functional testing (though non-functional correctness tests are also considered, e.g., in Chapter 7 and Chapter 14, but also for static testing in Chapter 6). Furthermore, there are various properties that can be tested, e.g., specific safety properties (cf. Sec. 4.2) or overall conformance (cf. Chapter 8) for functional testing, robustness or performance for non-functional testing;
- who is executing the system: in **active testing**, a test driver (cf. Sec. 2.3) that executes the test is included. Conversely, **passive testing** does not control test execution, but only deals with test oracles (cf. Sec. 2.3), so the system is executed in the productive environment or in some experiment defined elsewhere. A main approach to passive testing is the instrumentation of the system to enable monitoring the behavior of the program while it is executed. Therefore, this approach is called **program monitoring** (**monitoring** for short). If monitoring uses formal methods, the term "**runtime verification**" is often used instead [Bauer et al., 2010; Giannakopoulou and Havelund, 2001; Arcaini et al., 2013]. The oracles are often expressed with **assertions** Floyd [1967], which are Boolean expressions at

locations in the source code to check for conditions, i.e., an assertion must hold at the location it is given. Assertions are frequently formulated in the same programming language and are now used widely within most programming languages (the code of Microsoft Office contained over one quarter of a million assertions in 2003 Hoare [2003]). Since this thesis also covers test driver, it deals with active testing;

- the level at which the test is performed: testing can be performed at all levels of software development, resulting in acceptance tests, system tests, integration tests or unit tests (cf. Sec. 2.4). Some, but not all, aspects vary depending on the level, e.g., unit tests are mostly functional tests, whereas acceptance tests usually also cover many non-functional requirements. Testing with software bounded model checking (cf. Chapter 7) deals mainly with system testing, whereas model-based testing (cf. Part III) covers all levels;
- automation for test execution: **manual testing** is a labor-intensive, error-prone and hardly manageable activity (for instance, a case study on a commercially available test suite found errors in 15% of its test cases [Jard et al., 2000]). Hence **automated testing** should be preferred wherever possible. Therefore, this thesis focuses on automated testing;
- automation for test case generation: Automated testing in industry mainly uses **manual test case generation**, especially for unit tests, e.g., via test-driven development (cf. Subsec. 14.2.2). But lacking automated generation, too few exemplars are tested, especially for higher levels than unit testing. Furthermore, test suite maintenance must be performed manually, which is time-consuming and error-prone. Using **automated test case generation**, it is possible to cover the system under test more thoroughly, and only the concise specifications need to be maintained. For (semi-)automated test case generation, many more aspects can be differentiated, as investigated in Sec. 10.2. To achieve higher coverage, Part III deals with automated test case generation. Chapter 7 shows an alternative, where abstract test cases are created manually, but higher coverage is also achieved since a test case fully analyzes general scenarios with the help of nondeterminism .

Often, there are shades of gray between two extremes. For instance between black-box and white-box, there is **gray-box testing** [Zander et al., 2011; Weißleder, 2009], which has some knowledge of the internals and thus combines black-box testing with some techniques from white-box testing, mainly for test case generation. A typical example is a web application, where the interfaces are specified (e.g., as WSDLs, cf. Subsec. 14.3.8), but the source code is not available. Another example is measuring coverage for the source code, not only for the specification.

Notes. Static testing is white-box testing because insights into the system under test must be available to analyze the system without execution.

Passive and active testing are both part of dynamic testing.

2.3. Artifacts

This section defines all major artifacts of black-box testing, depicted in Fig. 2.1.

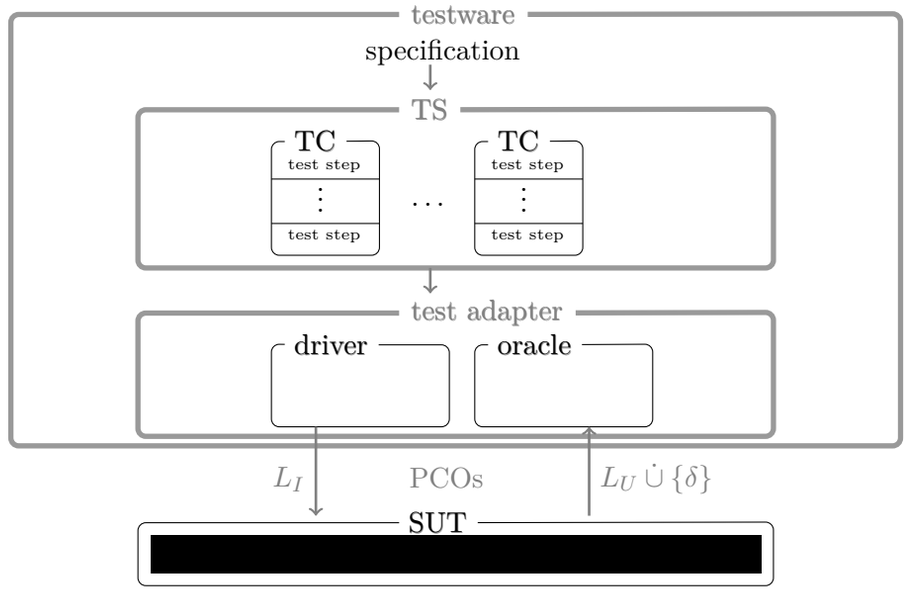


Figure 2.1.: Artifacts of black-box testing

The system that is tested is called **system under test** (**SUT**); we define ***SUT*** as the set of all SUTs.

A system is **closed** [Kupferman et al., 2001; Kupferman and Vardi, 2006] iff its behavior is completely determined by the system's own state. Contrarily, the behavior of an **open system** depends on interaction with an environment (e.g., a human) only partially known in advance. Therefore, there is a trade-off between open and closed systems: An open system gives some control and decisions to the environment, making the system itself less complex and more flexible by allowing the environment to change within the decisions and behaviors left open. But it gives up some control and becomes dependent on the environment for the parts left open.

If the SUT is a closed system, the environment is preferably the same as for the live deployment.

The **test adapter** is the environment around the SUT that performs two tasks:

- it executes the SUT via inputs (from the set L_I of possible inputs): this part of the test adapter is called **test driver (abstr. def.)**;
- it derives verdicts from the execution by observing outputs (from the set L_U of possible outputs), or quiescence δ (i.e., absence of output): this part of the test adapter is called **oracle**.

The interaction points of the interface between the test adapter and the SUT are called **points of control and observation** (**PCOs**).

To perform its tasks, a test adapter contains a **test suite** (**TS**), which is a set of **test cases** (**TCs**): finite sequences (or other structures, cf. Subsec. 10.2.4) of **test steps** that either aid the test driver or the oracle (or both for quiescence, cf. Subsec. 8.2.3).

The **testware** contains all testing artifacts that are created during the testing process [Graham and Fewster, 2012; Board), 2012].

Specifications specify some desired results, such as the final product or TS; so they

define certain functional or non-functional aspects on some level of abstraction. Thus multiple specifications are often created over multiple layers of abstraction (see the following section). Besides determining the desired aspects, they help in communicating. Even though writing specifications is costly, a dime spent on specification is a dollar saved on verification [Avizienis, 1995]. Formal specifications, i.e., specifications with formal descriptions avoid ambiguity and thus potential misunderstanding, and can be processed automatically for static or dynamic testing. Specifications not created by test engineers are usually not considered as part of the testware. The more a specification is simultaneously used in the software development process to specify the final product and the TS, the less the TS can verify the final product: for lack of redundancy, the system and TS are both based on the specification, so specification errors are not detected by the TS [Pretschner and Philipps, 2005; Faragó et al., 2013].

Traceability manages the relationships between different specifications, mainly requirements specifications, e.g., over different versions and over different abstraction layers, or between specifications and other artifacts from design, implementation, and testing. The relationships are usually described by adding **trace links**, i.e., bi-directional references between the artifacts (backward and forward, describing, for instance, how a requirement evolves, is refined by other requirements, satisfied by the system design, implemented by source code, or verified by test artifacts (see the following section or [Eide, 2005; Utting and Legeard, 2007; Mlynarski, 2011; Gotel et al., 2012]). Therefore, the changes of requirements and traces left by requirements on other artifacts, and vice versa, can be captured and followed [Pinheiro and Goguen, 1996]. If this capturing and the addition of trace links is performed automatically, we have **automatic traceability** [Banka and Kolla, 2015]. Traceability also helps to satisfy certification guidelines [Rajan, 2009; DO178C Plenary, 2011]. So the word "traceability" originates from tracing a requirement throughout the software development process, as described in the next section.

2.4. Software Engineering

A test engineer creates and executes test suites within a software development process, which describes the phases of software development and the order in which those phases are executed. The process is also called software engineering process [Boehm, 1984], and is part of the software life cycle process [JTC 1/SC 7, 2008]. Test creation (especially automated generation of test suites, cf. Sec. 2.2) is covered in Part III. Test execution, processes and other software engineering aspects are introduced in this section.

The **V-model** [Spillner and Linz, 2005; Langer and Tautschnig, 2008; JTC 1/SC 7, 2008] is the most prominent model to embed testing over several layers and **phases of the software development process** [Boehm, 1984; Royce, 1970]. The V-model is depicted in Fig. 2.2, taken from [Weißleder, 2009]. The axis downward moves to more and more detailed layers; the axis in the right direction resembles the processing sequence. The left branch describes how the product is constructed:

- **requirements**: Firstly, the needs of the customers and users are gathered, specified and approved; thus these requirements are often called **user requirements** or **business requirements** (or business requirements on top of user require-

ments [Westfall, 2005; Banka and Kolla, 2015]), and focus on conditions and capabilities the stakeholder wants the system to achieve. Each requirement (**REQ** for short) is a need that the final product must meet: either **functional**, i.e., about the system's behavior, specified in a **functional requirement specification** or **non-functional**, i.e., how well the system carries out its functions (e.g., its performance or energy consumption or robustness), specified in a **non-functional requirement specification** [Utting and Legeard, 2007; Langer and Tautschnig, 2008; ISO/IEC/IEEE, 2011; Turban, 2011];

- **system requirements**: The user requirements are translated to a technical level. They are often called **system requirements**, for software also **software requirements**. The **system requirements specification** (**system specification** for short) specifies a finite list of behaviors and features, is written to be verifiable and focuses on how the system will achieve the user requirements [Hayhurst et al., 2001];
- **system design**: A **system architecture** is established: it defines the system's structure by decomposing the system into subsystems, called **units**, and the system's behavior by specifying interfaces between units, as well as towards the environment, i.e., between the system and other systems. The system requirements are mapped to units;
- **unit design**: The system design is refined for each unit, describing in detail the unit's structure and behavior;
- **implementation**: builds all units.

The right branch describes how the product is tested, i.e., how to check whether the implementation conforms (cf. Chapter 8) to the specification:

- **unit testing**: tests for each unit whether it has been implemented according to the unit design; if not, either the unit design specification must be adapted, or the implementation. Often each test is performed in isolation [Fowler, 2007];
- **integration testing**: incrementally the subsystems are integrated, i.e., linked together, and tested whether they work with each other as described in the system design (and according to the interface to the external environment in **system integration testing**);
- **system testing**: once the entire system has been built and integrated, it is tested against the system specification;
- **acceptance testing**: finally, the system is tested against the user requirements, changing the focus from a technical to a user perspective and from verification to validation, i.e., from checking that the system is built in the right way to checking that the right system is built (see Subsec. 14.2.2 or [Boehm, 1984; Balzert, 1997; Pezzé and Young, 2007]).

The cost of fixing software bugs increases exponentially with the number of development phases containing the bug (cf. Subsec. 1.1.1); in case a bug is introduced in the requirements phase and fixed only after acceptance testing, i.e., after delivery of the system, the cost is often 100 times higher than fixing the bug in the design phase already [Shull et al., 2002].

Traceability can be implemented in the V-model for instance by using trace links

- between each artifact and the corresponding requirements, or
- between corresponding artifacts on one layer of the V-model (i.e., horizontal trac-

ing) and additionally between corresponding artifacts on two consecutive levels on the left branch of the V-model (i.e., vertical tracing) [Fisher, 2007; Utting and Legeard, 2007; Mlynarski, 2011; Turban, 2011; Gotel et al., 2012].

Many refinements and extensions to the V-model exist [Hillenbrand, 2011], for instance the current **V-Modell XT** [mbH, 2006]. By not interpreting the processing sequence as one single pass, and allowing gaps, the V-model becomes more flexible than waterfall [Royce, 1970; JTC 1/SC 7, 2008]. For instance, it allows checking the system specification's consistency by enhancing system testing with model checking (cf. Chapter 5, Chapter 7, and [Sijtema et al., 2014]) and iterative and incremental processes like Scrum (cf. Sec. 14.2 and [Turner, 2007]). An iteration can break a feature that worked before, which is called **regression bug** [Nir et al., 2007]. They can often be caught by old tests that already passed, so old tests should be executed in later iterations, too, which is called **regression testing** [Juzgado et al., 2004].

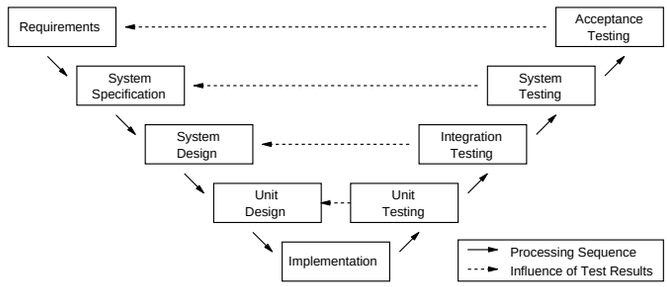


Figure 2.2.: The V-model

The execution of a test suite (resp. test case, resp. test step) on the SUT is called **test run** (resp. **test case execution**, resp. **test execution step** or just **test step**).

A test run can follow along **happy paths**, i.e., paths that exhibit the intended behavior of the SUT. Tests that follow other paths, e.g., exceptional behavior, are sometimes called **negative tests**, the corresponding behavior **bad weather behavior**. A robust SUT should be able to handle both cases, a thorough TS test for both.

The purpose of executing tests is to detect incorrectness, which can be differentiated into different stages according to **fault/failure models** [Ammann and Offutt, 2008; Beizer, 1990]: A **root cause** [Abran et al., 2004; Sommerville, 2010] (sometimes also called human error or just error [Board), 2012; IEEE, 1999]) is a human mistake, an action that produces a static defect in the SUT, called **fault** (or **defect** or **bug**). Exemplary faults are an incorrect data definition or an incorrect statement in the system's source code. If a fault is **activated**, i.e., encountered during execution, it may cause a **failure** of the SUT: deviation of the SUT's observed behavior from its specification, or more generally its expected delivery, service or result [Fenton, 1991; Board), 2012]. In summary, the main goal of testing is activating yet undetected faults (so called dormant faults).

Note. Some literature [Dubrova, 2013] swap the meaning of error and fault. Yet other literature [Weißleder, 2009; Ammann and Offutt, 2008; Offutt and Untch, 2001] define an error as an unintended internal state that is the manifestation of a fault in a running system. If such an error is propagated, i.e., influences the observed behavior, it leads to a failure – unless the resulting behavior happens to conform to the specification, too.

If executing a TC leads to the **verdict** fail , it indicates that a failure during the execution of the SUT has occurred. If the TC has the verdict **pass**, it indicates that no failure has been found. In some approaches, the execution of a TC might be terminated before fail or pass occurs (e.g., when the TC was not designed to consider all uncontrollable nondeterministic choices). This case leads to the verdict **inconclusive**, telling the test engineer that executing the TC neither yielded a bug nor raised the confidence in the SUT's correctness. Termination with inconclusive is often used when a particular functionality or aspect (e.g., a test purpose or test objective, cf. Chapter 12) should be investigated, but testing departs from this aspect due to nondeterminism or technical reasons. In these cases, inconclusive can also be called **miss**, and the more detailed verdicts **(miss,** fail **)** respectively **(miss,pass)** can be used to indicate that the departure from the aspect occurred at a fail respectively not at a fail . The set of allowed verdicts is defined as $\mathbb{V}$ and is { fail , pass} or { fail , pass, inconclusive} or { fail , pass, (miss, fail), (miss, pass)} or, indicating a verdict within the given aspect with **hit**, {(hit, fail),(hit,pass), (miss, fail), (miss,pass)}.

Note. If a test fails, the cause can either be a fault in the software, or a fault in the test itself. A test failure is often called an **anomaly** to avoid falsely concluding a fault in the software must be the reason [Black et al., 2012].

A test that fails should do so reproducibly, to enable the investigation of the corresponding fault so that the tests can also be used for regression testing. Therefore, repeating passing tests is a repeatable evidence of correctness. For SUTs that behave nondeterministically (cf. Subsec. 8.2.5), reproducibility is a challenge that is covered in Subsec. 12.4.3.

2.5. Meaningfulness and Heuristics

To efficiently perform software testing, the chosen test steps, test cases, and test suites should be **meaningful**, i.e., they should have a high potential of revealing many and relevant faults (thus the terms "**fault finding effectiveness**" or "**revealing**" are sometimes used equivalently) [Hamon et al., 2005; Weißleder, 2009; Gay et al., 2015]. The meaningfulness of a test step or TC depends on the previously executed TCs. In summary, the meaningfulness of a TS is a quality measure describing its failure detection capability. Therefore, the meaningfulness of the executed TS influences the confidence in the system, i.e., the probability of faults remaining in the system undetected.

For automation, meaningfulness is often approximated by standard heuristics, most prominently coverage criteria, as defined in Def. 2.1.

Definition 2.1. Let $\ddot{\mathbb{T}}$ be a test suite. Then

- a **coverage task** comprises one or more artifacts that $\ddot{\mathbb{T}}$ should **cover**, i.e., visit;
- a **coverage criterion** is a set C of coverage tasks;
- the **coverage level** of $\ddot{\mathbb{T}}$ for C is the percentage of coverage tasks that have been covered, i.e., the ratio (achieved coverage tasks)/$|C|$.

On the lowest level, there are coverage criteria with artifacts from the source code, called **code coverage criteria**. Thus they can only be used for white-box (and gray-box) testing. Examples are C =all statements, C =all branches in the control flow graph

or similarly C = all decisions (i.e., top-level Boolean expressions, aka predicates) with both outcomes, or C = all conditions (i.e., atomic Boolean expressions like a Boolean variable or relational expression) with both outcomes, or C = all paths through the control flow graph. Sometimes, the constraints are adapted [Weißleder, 2009], for instance additionally demanding that every point of entry and exist is covered. If some constraint is not feasible for each coverage task, a coverage level of 100% is not possible. Alternatively, a coverage criterion can be weakened to only consider feasible coverage tasks. Often, one coverage criterion subsumes another (cf. Subsec. 12.3.5 and [Chilenski and Miller, 1994; Ammann and Offutt, 2008; Weißleder, 2009]). Depending on the use of coverage criteria, the subsuming coverage criterion can lead to more meaningful TSs [Zhu, 1996; Kapoor and Bowen, 2003; Yu and Lau, 2006]. Many complex code coverage criteria exist, but no single one is best for all scenarios [Yu and Lau, 2006; Kapoor and Bowen, 2005; Utting and Legeard, 2007; Jorgensen, 2013].

Example. Modified Condition/Decision Coverage (MC/DC) [Chilenski and Miller, 1994; Hayhurst et al., 2001] strengthens condition coverage by demanding that the TS shows for each condition c within a decision d that c independently impacts the outcome of d (so c is not masked in d). The precise definition of MC/DC depends on what independent impact precisely means [Chilenski and Miller, 1994; Yu and Lau, 2006; Gay et al., 2015]). MC/DC is an efficient to compute while often more reliable and stable coverage criterion than many others [Kapoor and Bowen, 2003; Kandl and Kirner, 2010] and thus highly recommended for safety-critical software [DO178C Plenary, 2011; URL:ISO26262].

MC/DC is more efficient than many other coverage criteria, but still often does not yield sufficiently meaningful TSs [Whalen et al., 2013; Gay et al., 2015]: an exemplary case study on automotive software [Kandl and Kirner, 2010] using MC/DC led to a TS that revealed all erroneous values in the source code, but missed 8% of erroneous operators and 22% of erroneous variable names. Furthermore, MC/DC is very sensitive to the structure of the source code [Rajan et al., 2008a], where inlining a Boolean expression already has an effect on the coverage level. To avoid these deficits, Observable MC/DC (OMC/DC) [Gay et al., 2015] improves MC/DC by additionally demanding that the TS also reveals the outcome of the decision d: d must not be masked but affects a PCO, i.e., must be observable. Using OMC/DC reveals up to 88% more faults than using MC/DC, is less sensitive to program structure [Whalen et al., 2013], but more complex.

In short, coverage criteria are still evolving and should be chosen according to the situation, so formal methods should ideally be able to apply arbitrary criteria.

The described code coverage criteria all focused on control flow, i.e., on statements, decisions and control constructs. Hence they are called **control flow coverage criteria**. There are also **data flow coverage criteria** that focus on the status of variables (or data objects): on a path, a variable v can be defined (**d**), used in a computation (**c**) or used in a predicate (**p**). Certain sequences of such statuses, e.g., multiple definitions of v without a use in between (called dd anomaly), have a high probability of containing a fault and should thus be covered by tests. Typical coverage tasks for v are **def-use pairs**: a path in the control flow from a definition of v to a use of v without further definitions of v in between (a so called def-clear path). Typical data flow coverage criteria are: **all-defs** = f.a. variables v f.a. definitions of v: at least one def-use pair, i.e., all definitions get used; **all-uses** = f.a. variables v: all def-use pairs, i.e., all uses affected by

a definition are exercised; **all-def-use-paths** =f.a. variables v f.a. def-use pairs: all def-clear paths (modulo loops) from the def to the use, i.e., all uses affected by a definition are exercised via all possible paths. Many more criteria exist, especially differentiating c and p [Frankl et al., 1997; Juzgado et al., 2004; Utting and Legeard, 2007].

Note. Most specifications used in this thesis (e.g., LTSs, see Subsec. 3.4.1) contain no variables, so data flow coverage criteria are of no use. For more complex specifications with variables, strong data flow coverage criteria often become difficult due to aliasing, the high amount of coverage tasks, bad scalability, path explosion, and infeasible paths [Frankl et al., 1997; Pezzé and Young, 2007; Naik and Tripathy, 2011; Su et al., 2015b,a]. Consequently, "practical data flow testing remains a significant challenge" [Su et al., 2015a], so industry usually focuses on control flow coverage criteria instead of data flow coverage criteria [Hayhurst et al., 2001; DO178C Plenary, 2011], which we will also do in this thesis.

Typical coverage criteria on the specification level, called **specifications coverage**, are $C =$ all states of the model specification, $C =$ all transitions of the specification, or $C =$ all paths of the specification (cf. Def. 8.56 and [Weißleder, 2009; Peleska, 2013]). Many complex coverage criteria and combinations exist [Abdurazik et al., 2000; Friske et al., 2008; Ammann and Offutt, 2008]. Some are the same as on the lowest level, for instance decision-based coverage criteria, but their meaningfulness can vary depending on the level [Yu and Lau, 2006; Krishnan et al., 2012].

On the highest level, there is **requirements coverage**, for instance $C =$ all requirements [Ghidella and Mosterman, 2005; Whalen et al., 2006; Fraser and Wotawa, 2006; Rajan et al., 2008b; Rajan, 2009; Lackner and Schlingloff, 2012]. Generating tests for high requirements coverage is often called **requirements-based testing**. If these tests directly check the conformance of the implementation to the requirements, or use traceability to indicate which requirements might be violated if a test fails, the term **requirements conformance testing** can also be used.

Notes. Often a mix of coverage criteria on multiple levels are used, e.g., tests are generated to maximize specifications coverage and requirements coverage.

The process of applying coverage criteria on multiple levels is according to the V-model, i.e., reverse to the didactical order listed here.

The coverage criteria on different levels complement one another; for instance using requirements coverage when producing tests reveals if there are requirements without corresponding implemented functions, while code coverage reveals if there are implemented functions without corresponding requirements.

Most coverage criteria on the source code and specification level focus on the structure of the artifacts and are hence called **structural coverage criteria**. Non-structural coverage criteria on those levels are input coverage criteria; requirements coverage criteria are non-structural coverage criteria on a higher level [Hayhurst et al., 2001].

For black-box testing, the used coverage criteria should be implementation independent and not include code coverage. Therefore, the generated TS can be applied to multiple SUTs, e.g., when the source code is modified.

A coverage criterion can be used as

- **exit criterion**, i.e., to decide when to stop testing;
- similarly as **adequacy metric**, i.e., feedback of the coverage level to the test engineer to show the quality and progress of testing, especially whether some functionality has not yet been tested;
- guidance of which choices to take during testing or test generation, to find more meaningful test cases (cf. Chapter 12);
- **test suite reduction**, which is usually inefficient and ineffective post-processing as firstly generating a huge test suite and only thereafter applying heuristics is often too costly, and often leads to weak meaningfulness [Fraser et al., 2009; Heimdahl and Devaraj, 2004].

Meaningfulness of a TS, TC or test step is highly dependent on the domain and the way the SUT was developed. Therefore, it also varies how well a coverage criterion approximates meaningfulness [Weyuker and Jeng, 1991; Horgan et al., 1994; Frankl et al., 1997; Gutjahr, 1999; Juzgado et al., 2004; Heimdahl and Devaraj, 2004; Heimdahl et al., 2004; Fraser and Wotawa, 2006; Weißleder, 2009; Mockus et al., 2009; Derderian et al., 2006; Ali et al., 2010; Utting and Legeard, 2007; Cadar et al., 2008b; Staats et al., 2012; Pretschner et al., 2013; Godefroid et al., 2005; Gay et al., 2015] (see also Subsec. 12.3.1). Consequently, it is concerning that several test engineers use some coverage criterion as though it universally guarantees effective testing [Rajan et al., 2008a; Kandl and Kirner, 2010; Gay et al., 2015]. Therefore, testing methods and tools should be sufficiently generic to apply different coverage criteria as well as other heuristics.

Notes. In the extreme case of exhaustiveness (cf. Sec. 8.8), meaningfulness can, however, be strictly related to a suitable coverage criterion: By considering exhaustiveness as maximal meaningfulness of a TS, meaningfulness relates to fully achieving a specific coverage criterion, as shown in Subsec. 8.8.4.

The term **directed test generation** sometimes denotes that tests are generated specifically to satisfy a coverage criterion, sometimes that no (purely) random guidance is used for test generation.

Mutation testing (aka **fault-based testing**) [Offutt and Untch, 2001] is an alternative to coverage criteria that is often more meaningful [Jia and Harman, 2011; Baker and Habli, 2013]. The main reason is that mutation testing emulates earlier common mistakes by **fault injections**: the source code is changed slightly, according to some syntactic rule for a mistake, described by a **mutation operator**; the modified source code is called a **mutant**. Typical mutation operators add, delete or replace programming language operators, object oriented language constructs or statements, or replace variables or constants. Therefore, mutation operators are language dependent. Since the C language already has over 70 mutation operators [Richard et al., 1989; Jia and Harman, 2008], and each can be applied at multiple locations, the overall set of mutants, M, can become huge. A TC or TS **kills** a mutant $m \in M$ if the TC or a TC from the TS fails on m [Weißleder, 2009; Grün et al., 2009; Jia and Harman, 2011]. Since a TC can check for the internal state of a program, this is **weak mutation testing**, in contrast to **strong mutation testing** where the outputs of the original program and m must differ, not only the internal state [Offutt and Untch, 2001; Jia and Harman, 2011; Krishnan et al., 2012] (there are, however, other definitions of weak and strong mutation

testing [Juzgado et al., 2004]). A mutant that is functionally equivalent to the original is called **equivalent mutant**. The **mutation kill ratio** (aka **mutation score**) k of a TS is the number of killed mutants divided by $|M|$, which is a metric for the meaningfulness of the TS. Occasionally, the mutation score is considered as coverage level, and mutation testing as one form of coverage testing, which we will not adopt. If the mutation operators reflect typical mistakes made by the software developers, relative values of k for different TSs accurately reflect their meaningfulness. The absolute value k for a TS usually has little informative value since an unknown but usually high amount of mutants are equivalent (about 10% to 50% [Grün et al., 2009; Aichernig et al., 2011; Krishnan et al., 2012; Baker and Habli, 2013; Madeyski et al., 2014]). Just as for coverage metrics, mutation testing must be restricted to feasible situation to get a more informative value; but this requires detecting and excluding equivalent mutants, which is hard in practice and in general undecidable [Frankl et al., 1997; Grün et al., 2009; Madeyski et al., 2014; Aichernig et al., 2014] (i.e., the check for equivalence is non-computable). Like coverage metrics, mutation testing can be used as adequacy metric and test suite reduction. If the number of equivalent mutants is known, mutation testing can also be used as exit criterion. Since M can become huge and the whole TS has to be executed for each mutant $m \in M$, mutation testing unfortunately has a high time complexity [Jia and Harman, 2011; Baker and Habli, 2013], especially for guidance of test case generation [Aichernig et al., 2011; Jia and Harman, 2011; Aichernig et al., 2014]. The execution time can be reduced to some extent [Jia and Harman, 2011], for instance by using structural coverage criteria to individually select the mutants each TC should execute [Grün et al., 2009]. But test execution, especially due to the high amount of equivalent mutants, still often hinders the application of mutation testing in practice [Frankl et al., 1997; Jia and Harman, 2011; Baker and Habli, 2013]. Therefore, mutation testing is only considered in a few notes in this thesis.

In summary, meaningfulness cannot be deduced at large by the specification in isolation, or by a single metric in practice; thus heuristics need to be chosen carefully for each domain and hence are best supplied by the user (cf. [Feijs et al., 2002] and Subsec. 11.2.4).

For SUTs that behave nondeterministically (cf. Subsec. 8.2.5), new aspects arise for coverage criteria: The **static coverage level** of $\ddot{\mathbb{T}}$ (**static coverage** for short), which is measured without executing $\ddot{\mathbb{T}}$ on the SUT, does not yield reliable results. Instead, the **dynamic coverage level** of $\ddot{\mathbb{T}}$ (**dynamic coverage** for short), measured while executing $\ddot{\mathbb{T}}$ on the SUT, gives the accurate results that reflect the dynamic information about the SUTs nondeterministic behavior (cf. Subsec. 11.1.2). Furthermore, new coverage criteria that measure the SUT's nondeterminism are possible (cf. Subsec. 12.3.1, [Faragó, 2011]).

2.6. Summary

This chapter gave an overview of testing. To briefly span the wide field and position the testing methods used in this thesis, a taxonomy was introduced. After defining the artifacts used for testing, the relevant software engineering aspects for testing were covered, including the V-model, fault/failure models, meaningfulness, and coverage criteria.

3. Introduction to Formal Methods

3.1. Introduction

Formal methods are mathematically rigorous techniques using formal languages, logics or automata theory, for developing, specifying and verifying (cf. Def. 3.1) software and hardware systems.

Definition 3.1. Verification is the process of mathematically proving or disproving the correctness of a system with respect to the formally given properties.

Note. In the domain of (model-based) testing, verification is often defined wider by including testing [Ammann and Offutt, 2008], in which case it is often called **dynamic verification**. This thesis does not include testing in the term "verification", but uses the term "**formal verification**" where misunderstanding could occur.

Many terms introduced in the last chapter for testing can be transferred to verification, e.g., **(non-) functional verification**, **software verification**, **manual verification**, which is usually called **interactive verification**, and **system under verification** (**SUV**) for the system being checked by verification.

Def. 3.2 about soundness and completeness [Kleene, 1967] is sufficiently general to cover all formal methods within this thesis. We will give more precise instances of this definition for all formal methods in their respective chapters.

Definition 3.2. Let $\mathcal{M}$ be a formal method. Then:

$$
\begin{aligned}
\mathcal{M} \text{ is } \textbf{sound} \quad :\Leftrightarrow\ & \forall \text{ systems under consideration } \mathcal{S} \\
& \forall \text{ relevant property statements } \mathcal{P} \text{ about } \mathcal{S} : \\
& (\mathcal{M} \text{ deduces } \mathcal{P} \text{ about } \mathcal{S} \Rightarrow \mathcal{P} \text{ holds for } \mathcal{S}) \\
\mathcal{M} \text{ is } \textbf{complete} :\Leftrightarrow\ & \forall \text{ systems under consideration } \mathcal{S} \\
& \forall \text{ relevant property statements } \mathcal{P} \text{ about } \mathcal{S} : \\
& (\mathcal{M} \text{ deduces } \mathcal{P} \text{ about } \mathcal{S} \Leftarrow \mathcal{P} \text{ holds for } \mathcal{S})
\end{aligned}
$$

Notes. So if a sound method $\mathcal{M}$ deduces a property $\mathcal{P}$ about $\mathcal{S}$, then $\mathcal{P}$ really holds for $\mathcal{S}$, i.e., $\mathcal{M}$ never gives **false positives** (called **false alarms** for $\mathcal{P}$ describing bad behavior). Conversely, a complete method $\mathcal{M}$ eventually does deduce property $\mathcal{P}$ if $\mathcal{P}$ really holds for $\mathcal{S}$. So $\mathcal{M}$ never gives **false negatives**, i.e., **misses** a property (cf. [Neyman and Pearson, 1933]).

The more heavyweight sound and complete methods are in general not computable (cf. Chapter 1), i.e., represent a function or set that is not computable by a Turing machine [Turing, 1936].

3.2. Propositional Logic

Propositional logic is a basic core used throughout this thesis. This section firstly describes its syntax, then its semantics.

3.2.1. Syntax

Def. 3.3 uses the operator basis $\{\neg, \vee\}$ to define propositional formulas. Thereafter, formulas with other operators are covered by considering those operators as abbreviations.

Definition 3.3. Let $\boldsymbol{\Sigma}$ be a countable set, called **signature**, whose elements are called **propositional variables**.

Then $\mathbf{PROP}_\Sigma$ (shortly **PROP** if Σ is clear) is the set of all **propositional formulas**, defined by

$$\langle \text{prop} \rangle ::= p \mid (\langle \text{prop} \rangle \vee \langle \text{prop} \rangle) \mid \neg \langle \text{prop} \rangle; \qquad \text{with } p \in \Sigma$$

Secondary Operators. Besides the operator basis $\{\vee, \neg\}$, the following operators can be expressed within our operator basis, with $f_i \in \text{PROP}$:

- $(f_1 \wedge f_2)$ as abbreviation for $\neg(\neg f_1 \vee \neg f_2)$;
- $(f_1 \to f_2)$ as abbreviation for $(\neg f_1 \vee f_2)$;
- $(f_1 \leftrightarrow f_2)$ as abbreviation for $\big((f_1 \to f_2) \wedge (f_2 \to f_1)\big)$.

Note. Round brackets are used to determine the **priority of the operators** (i.e., how the formula's abstract syntax tree looks like). Often they are omitted and an implicit precedence is fixed: $\neg$ before $\wedge$ before $\vee$ before $\to$ before $\leftrightarrow$. Amongst the same binary operator, precedence is irrelevant for $\wedge$, $\vee$ and $\leftrightarrow$ (i.e., they are associative); for $\to$, we fix the precedence right to left (i.e., right-associativity).

3.2.2. Semantics

Def. 3.4 defines terminology, so that Def. 3.5 can inductively define the **semantics of $\mathbf{PROP}_\Sigma$** .

Definition 3.4. Let Σ be given and $F \in \text{PROP}_\Sigma$. Then:

- $I : \Sigma \to \mathbb{B}$ is called an **interpretation** over Σ;
- $\boldsymbol{val_I} : \text{PROP}_\Sigma \to \mathbb{B}$ is the **evaluation function for PROP**, which extends I to PROP_Σ, describing PROP_Σ's semantics;
- I **satisfies** $F \;:\Leftrightarrow val_I(F) = \textbf{true}$ (written $I \models F$)
  $\Leftrightarrow$: I is a **model** of F;
- F is **valid** $:\Leftrightarrow$ f.a. interpretations I over $\Sigma : I \models F$.

Definition 3.5. Let Σ be given and $f_i \in \text{PROP}_\Sigma$. Then:

$$\begin{aligned} &\bullet I \models \neg f_1 && :\Leftrightarrow I \not\models f_1; \\ &\bullet I \models (f_1 \vee f_2) && :\Leftrightarrow I \models f_1 \text{ or } I \models f_2. \end{aligned}$$

3.2.3. SAT Solvers

Before defining SAT and SAT solvers in Def. 3.6, we introduce the general case, which often occurs in practice and is also applicable to SMT and SMT solvers (cf. Subsec. 3.3.3): a **constraint satisfaction problem** (**CSP**) [Kroening and Strichman, 2008; Petke, 2015] is a finite set of constraints over a finite set of variables. A program or algorithm that solves CSPs is called a **constraint solver**. Restricting the constraints and variable domains to Boolean values, we get propositional formulas as constraints and SAT solvers to solve them, as defined in Def. 3.6.

Definition 3.6. Let Σ be given and $F \in \mathrm{PROP}_\Sigma$. Then:

- the **Boolean satisfiability problem** (**SAT**) for F determines if there exists an interpretation I with $I \models F$;
- a **SAT solver** determines whether F is satisfiable. If so, the SAT solver returns the string "SAT" and an interpretation, otherwise the string "$UNSAT$".

SAT is NP-complete [Cook, 1971], so the worst case time complexity is in $O(2^{|\Sigma|})$. Yet modern SAT solvers are efficient for many practical applications [Clarke et al., 2009; Järvisalo et al., 2012; Balint et al., 2013; URL:SATChallenge2012; Belov et al., 2014; URL:SATcompetition2014HP]. A prominent Boolean decision procedure applied by SAT solvers for formulas in conjunctive normal form is DPLL [Davis, 1962], also used for SMT solvers (see Subsec. 3.3.3). An exemplary SAT solver based on DPLL is **Minisat2** [Eén and Sörensson, 2003; Sörensson, 2008], which is popular and often used as basis or reference [URL:SATChallenge2012; Belov et al., 2014; URL:SATcompetition2014HP].

Note 3.7. Usually only sound and complete SAT solvers are considered. As determined by Def. 3.2, a SAT solver $\mathcal{M}$ is:

$$
\begin{aligned}
\textbf{sound} \quad &:\Leftrightarrow \forall \text{ signatures } \Sigma \, \forall \text{ formulas } F \in PROP_\Sigma \, \forall \text{ statements } \mathcal{P} \in \{SAT, UNSAT\} : \\
&\quad (\mathcal{M} \text{ deduces } \mathcal{P} \text{ about } F \Rightarrow \mathcal{P} \text{ holds for } F) \\
\textbf{complete} &:\Leftrightarrow \forall \text{ signatures } \Sigma \, \forall \text{ formulas } F \in PROP_\Sigma \\
&\quad \forall \text{ statements } \mathcal{P} \ \in \{SAT, UNSAT\} : \\
&\qquad\qquad (\mathcal{M} \text{ deduces } \mathcal{P} \text{ about } F \Leftarrow \mathcal{P} \text{ holds for } F)
\end{aligned}
$$

In this definition [Maric, 2009], the property statements are from $\{SAT, UNSAT\}$ and the systems under consideration from $PROP_\Sigma$, which corresponds to the typical use of SAT solving for verification, where the system is encoded as formula (cf. Chapter 7).

If $\mathcal{M}$ terminates, soundness and completeness of SAT and $UNSAT$ are related; soundness of both imply their completeness.

Alternatively (as for model checking, cf. Chapter 5), the unsatisfiability of some formula in $PROP_\Sigma$ can be considered as the property statement, and the interpretations are the systems under consideration.

3.2.4. BDD-based Techniques

Binary decision diagrams are data structures to efficiently represent propositional formulas and perform operations on them. They are often used for model checking (cf. Chapter 5).

Definition 3.8. An **ordered binary decision diagram** (**OBDD**) over the ordered propositional variables $\Sigma = (p_i)_{i \in [0,\ldots,k]}$ is a single-rooted, connected, finite, directed, acyclic graph where:

- an inner node n is labeled with a propositional variable p_j, where $j > i$ if n has a direct predecessor labeled with p_i;
- an inner node has two outgoing edges labeled **true** and **false**;
- a terminal node is labeled with **true** or **false**.

The label of the outgoing edge from an inner node labeled with p_i determines its Boolean assignment, i.e., $I(p_i)$. Thus each interpretation I over Σ leads to a maximal path π_I in the OBDD. The OBDD represents the propositional logic formula f iff $\forall$ interpretations I : $val_I(f) =$ label of π_I's terminal node.

Note. The size $|B|$ of an OBDD B representing a formula f, i.e., its number of nodes, strongly depends on the variable order on Σ, with up to exponential difference. Unfortunately, improving the variable order is NP-complete [Bollig and Wegener, 1996], so heuristics are used. Besides domain-specific solutions, there are general approaches using dynamic reordering, which optimizes the order during runtime by lazily repositioning one variable at a time, e.g., using the sifting algorithm [Rudell, 1993].

Definition 3.9. A **reduced ordered binary decision digram** (**ROBDD**) is a normal form for OBDDs where the following transformations can no longer be applied:

- if two terminal nodes have the same label, then remove one and redirect incoming edges to the other one;
- if two inner nodes n_1, n_2 have the same label, the same successor for their outgoing edge **true** and the same successor for their outgoing edge **false**, then remove n_1 and redirect its incoming edges to n_2;
- if an inner node has the same direct successor twice, then remove the inner node and redirect its incoming edges to the direct successor.

Lemma 3.10 from [Bryant, 1986] shows that ROBDDs are very useful for MC. Two ROBDDs are isomorphic if the graph has the same structure and same labels [Clarke et al., 1999b].

Lemma 3.10. *For all propositional formulas f_1, f_2 : $f_1 \equiv f_2$ iff ROBDD of f_1 is isomorphic to ROBDD of f_2.*

If multiple ROBDDs are used, sub-graphs are also shared amongst them, leading to the multi-rooted **shared reduced ordered binary decision diagrams** (often shortened to **BDD**). All propositional operators can be implemented on BDDs with polynomial worst case time complexity: for BDDs B_1, resp. B_2, representing f_1, resp. f_2, and a binary operator $\circ$ (e.g., $f_1 \circ f_2 = f_1 \wedge f_2$ or $f_1 \circ f_2 = \neg f_2$), a BDD for $f_1 \circ f_2$ can be computed by operating directly on B_1 and B_2 with a worst case time complexity linear in $|B_1| \cdot |B_2|$ [Bryant, 1986].

3.3. First Order Logic

First order logic (**FOL**) extends propositional logic with functions, predicates, and quantified variables; again, the syntax is described before the semantics.

3.3.1. Syntax

Def. 3.11 defines the syntax of FOL using the operator basis $\{\neg, \vee, \exists\}$ to define formulas. Thereafter, Def. 3.12 defines free variables of formulas. Finally, formulas with other operators are introduced by considering those operators as abbreviations.

Definition 3.11. For FOL, a **signature $\boldsymbol{\Sigma_{FOL}} = (F, P, \alpha)$** consists of
- a countable set $\boldsymbol{F}$ of **function symbols**;
- a countable set $\boldsymbol{P}$ of **predicate symbols**, with $F \cap P = \varnothing$;
- a function $\boldsymbol{\alpha} : F \dot\cup P \to \mathbb{N}_{\geq 0}$, mapping each symbol to its **arity**.

$\boldsymbol{Var}$ is the countable set of **variables**.

$\boldsymbol{TERM_{\Sigma_{FOL}}(Var)}$ (shortly $\boldsymbol{TERM_{\Sigma_{FOL}}}$ or $\boldsymbol{TERM}$ if the context is clear) is the set of all **terms** over Var for Σ_{FOL}, defined by

$$\langle \text{term} \rangle ::= v \mid f(\overbrace{\langle \text{term} \rangle, \ldots, \langle \text{term} \rangle}^{\alpha(f) \text{ parameters}}); \qquad \text{with } f \in F,\ v \in Var$$

$\boldsymbol{FORM_{\Sigma_{FOL}}(Var)}$ (shortly $\boldsymbol{FORM_{\Sigma_{FOL}}}$ or $\boldsymbol{FORM}$ if the context is clear) is the set of all **first order formulas** (with equality) over Var for Σ_{FOL}, defined by

$$\begin{aligned} \langle \text{form} \rangle ::= {} & \langle \text{atomic} \rangle \mid \neg \langle \text{form} \rangle \mid (\langle \text{form} \rangle \vee \langle \text{form} \rangle) \mid \\ & \exists x \, \langle \text{form} \rangle; \qquad \text{with } x \in Var \\ \langle \text{atomic} \rangle ::= {} & p(\overbrace{\langle \text{term} \rangle, \ldots, \langle \text{term} \rangle}^{\alpha(p) \text{ parameters}}) \mid (\langle \text{term} \rangle \doteq \langle \text{term} \rangle); \qquad \text{with } p \in P \end{aligned}$$

Definition 3.12. Let Σ_{FOL}, $x \in Var$, $f \in F$, $t_i \in$TERM, $a, p \in P$ with $\alpha(a) = 0 \neq \alpha(p)$, and $F_i \in$ FORM be given.

Then the function $\boldsymbol{free}$: $\text{TERM}_{\Sigma_{FOL}} \dot\cup \text{FORM}_{\Sigma_{FOL}} \to 2^{Var}$ maps each first order term or formula X to the set of **free variables** in X, inductively defined as:

$$\begin{aligned} &\bullet free(x) && := \{x\}; \\ &\bullet free(f(t_1, \ldots, t_{\alpha(f)})) && := free(t_1) \cup \cdots \cup free(t_{\alpha(f)}); \\ &\bullet free(t_1 \doteq t_2) && := free(t_1) \cup free(t_2); \\ &\bullet free(a) && := \varnothing; \\ &\bullet free(p(t_1, \ldots, t_{\alpha(p)})) && := free(t_1) \cup \cdots \cup free(t_{\alpha(p)}); \\ &\bullet free(\neg F_1) && := free(F_1); \\ &\bullet free(F_1 \vee F_2) && := free(F_1) \cup free(F_2); \\ &\bullet free(\exists x F_1) && := free(F_1) \setminus \{x\}. \end{aligned}$$

A variable $x \notin free(F_1)$ that occurs in F_1 is called **bound (by a quantifier)** in F_1.

F_1 is called a **closed formula** iff $free(F_1) = \varnothing$, i.e., it only contains bound variables.

Secondary Operators. Besides the operator basis $\{\neg, \vee, \exists\}$, the following operators can be expressed within our operator basis, with $f_i \in$ FORM:
- $(f_1 \wedge f_2)$ as abbreviation for $\neg(\neg f_1 \vee \neg f_2)$;
- $(f_1 \to f_2)$ as abbreviation for $(\neg f_1 \vee f_2)$;
- $(f_1 \leftrightarrow f_2)$ as abbreviation for $((f_1 \to f_2) \wedge (f_2 \to f_1))$.

- $\forall x f_1$ as abbreviation for $\neg\exists x \neg f_1$.

Note. Round brackets are used to determine the priority of the operators. Often they are omitted and an implicit precedence is fixed: $\{\neg, \exists, \forall\}$ before $\wedge$ before $\vee$ before $\rightarrow$ before $\leftrightarrow$.

3.3.2. Semantics

Def. 3.13 defines terminology, so that Def. 3.14 can inductively define the **semantics of FORM**$_{\Sigma_{FOL}}(Var)$.

Definition 3.13. Let Σ_{FOL}, $t \in \text{TERM}$, and $F \in \text{FORM}$ be given. Then:

- a **first order structure** $\boldsymbol{\mathcal{D}_{\Sigma_{FOL}}} = (D_{\Sigma_{FOL}}, I_{\Sigma_{FOL}})$ over Σ_{FOL} (shorty $\boldsymbol{\mathcal{D}} = (D, I)$ if the context is clear) consists of:
  - a nonempty set $\boldsymbol{D_{\Sigma_{FOL}}}$, called **universe**;
  - a function $\boldsymbol{I_{\Sigma_{FOL}}}$, called **interpretation**, that maps every $f \in F$ to a function $I_{\Sigma_{FOL}}(f) : D_{\Sigma_{FOL}}^{\alpha(f)} \rightarrow D_{\Sigma_{FOL}}$ and every $p \in P$ to a relation $I_{\Sigma_{FOL}}(p) \subseteq D_{\Sigma_{FOL}}^{\alpha(p)}$;
- a **variable assignment (first order logic)** $\boldsymbol{\beta}$ (also called **valuation**) is a function $\beta : Var \rightarrow D_{\Sigma_{FOL}}$. Furthermore, with $x \in Var$ and $d \in D_{\Sigma_{FOL}}$, we define $\beta_x^d : Var \rightarrow D_{\Sigma_{FOL}}, y \mapsto \begin{cases} d & \text{if } y = x \\ \beta(y) & \text{if } y \neq x \end{cases}$
- the **evaluation function for FOL** for the first order structure $\mathcal{D}_{\Sigma_{FOL}}$ and variable assignment β is $\boldsymbol{val_{\mathcal{D},\beta}}$: $\text{TERM}_{\Sigma_{FOL}} \dot\cup \text{FORM}_{\Sigma_{FOL}} \rightarrow D \cup \mathbb{B}$. It extends $I_{\Sigma_{FOL}}, \beta$ to $\text{TERM}_{\Sigma_{FOL}}$ and $\text{FORM}_{\Sigma_{FOL}}$, describing their semantics;
- $\mathcal{D}, \beta$ **satisfies** $F :\Leftrightarrow val_{\mathcal{D},\beta}(F) = \textbf{true}$ (written $\mathcal{D}, \beta \models F$);
- for $\beta' = \beta|_{Var'}$ with $free(F) \subseteq Var'$, we also write $\mathcal{D}, \beta' \models F$;
- $\mathcal{D} \models F :\Leftrightarrow$ f.a. $\beta : Var \rightarrow D : \mathcal{D}, \beta \models F$; we then say $\mathcal{D}$ is a **model** of F;
- F is **valid** $:\Leftrightarrow$ f.a. first order structures $\mathcal{D} : \mathcal{D} \models F$.

Definition 3.14. Let Σ_{FOL}, first order structure $\mathcal{D}$, variable assignment β, $x \in Var$, $f \in F$, $t_i \in \text{TERM}$, $a, p \in P$ with $\alpha(a) = 0 \neq \alpha(p)$, and $F_i \in \text{FORM}$ be given. Then:

$$
\begin{array}{ll}
\bullet val_{\mathcal{D},\beta}(x) & := \beta(x); \\
\bullet val_{\mathcal{D},\beta}(f(t_1, \ldots, t_{\alpha(f)})) & := I(f)(val_{\mathcal{D},\beta}(t_1), \ldots, val_{\mathcal{D},\beta}(t_{\alpha(f)})); \\
\bullet \mathcal{D}, \beta \models (t_1 \doteq t_2) & :\Leftrightarrow val_{\mathcal{D},\beta}(t_1) = val_{\mathcal{D},\beta}(t_2); \\
\bullet \mathcal{D}, \beta \models a & :\Leftrightarrow val_{\mathcal{D},\beta}(a) = \textbf{true}; \\
\bullet \mathcal{D}, \beta \models p(t_1, \ldots, t_{\alpha(p)}) & :\Leftrightarrow (val_{\mathcal{D},\beta}(t_1), \ldots, val_{\mathcal{D},\beta}(t_{\alpha(p)})) \in I(p); \\
\bullet \mathcal{D}, \beta \models \neg F_1 & :\Leftrightarrow \mathcal{D}, \beta \not\models F_1; \\
\bullet \mathcal{D}, \beta \models (F_1 \vee F_2) & :\Leftrightarrow \mathcal{D}, \beta \models F_1 \text{ or } \mathcal{D}, \beta \models F_2; \\
\bullet \mathcal{D}, \beta \models \exists x F_1 & :\Leftrightarrow \exists d \in D : \ \mathcal{D}, \beta_x^d \models F_1.
\end{array}
$$

3.3.3. SMT Solvers

Since FOL is undecidable [Church, 1936; Turing, 1936; Kleene, 1967], many automatic solvers consider some predetermined background **theory**: a set A of axioms, i.e., closed formulas in $\text{FORM}_{\Sigma_{FOL}}$ that should hold. The solvers hence only consider the subset

of all first order structures over Σ_{FOL} that are models of each axiom, and operate on subsets of $\text{FORM}_{\Sigma_{FOL}}$. These constraint solvers are called **SAT modulo theories** (**SMT**) solvers [Biere et al., 2009].

Various subsets of $\text{FORM}_{\Sigma_{FOL}}$, theories and combinations are possible, which are assembled and standardized in the SMT-LIB 2.0 standard [URL:SMT-LIB; Barrett et al., 2010]. Exemplary theories are given in Table 3.1, together with their decidability, for the corresponding subset of closed formulas in $\text{FORM}_{\Sigma_{FOL}}$, and for its subset of quantifier-free formulas, also called unquantified formulas. Fortunately, many practically relevant problems can be solved efficiently (cf. [Barrett et al., 2013] and Subsec. 3.2.3).

Table 3.1.: Exemplary theories and their decidability

theory	quantified subset of closed $\text{FORM}_{\Sigma_{FOL}}$	quantifier-free subset (QF)
core theory	undecidable	decidable
Presburger arithmetic	decidable	decidable
Ints theory	decidable	decidable
ArraysEx theory	undecidable	decidable
FixedSizeBitVectors theory	decidable	decidable

Common subsets of $\text{FORM}_{\Sigma_{FOL}}$ for some theory (or multiple theories) are defined as so-called **SMT logics** (**logics** for short) [URL:SMT-LIB; Barrett et al., 2010; Kroening and Strichman, 2008], e.g.,

- QF_UF, using quantifier-free uninterpreted sort and function symbols over the core theory;
- QF_LIA, using quantifier-free linear integer arithmetic over the Ints theory;
- QF_AX, using quantifier-free formulas with arrays over the ArraysEx theory;
- QF_BV, using quantifier-free formulas with bitvectors over the FixedSizeBitVectors theory;
- QF_AUFBV, using quantifier-free formulas with uninterpreted sort and function symbols, integer arithmetic via bitvectors, and arrays mapping bitvectors to bitvectors, over the theory combination core, FixedSizeBitVectors, and ArraysEx;
- AUFLIA, using closed formulas with uninterpreted sort and function symbols, linear integer arithmetic and arrays with integer indices and values over the theory combination core, Ints and ArraysEx.

The SMT-LIB also offers a uniform interface for SMT solvers. **SMT solvers** lift SAT solving (e.g., via DPLL [Nieuwenhuis et al., 2006]) to the level of SMT (cf. Subsec. 3.6.2 or [Sebastiani, 2007; Biere et al., 2009]:

- either **eagerly** by translating the SMT formula to an equisatisfiable propositional formula F in a first step, and then solve F with a SAT solver;
- or **lazily** by integrating theory solving and SAT solving that treats theory atoms as propositional atoms.

To simultaneously use multiple theories, they have to be combined by the solvers [Kroening and Strichman, 2008] (e.g., using the Nelson and Oppen approach [Nelson and Oppen, 1979], or delayed theory combination [Bozzano et al., 2005], or model-based theory combination [de Moura and Bjorner, 2008a]).

One of the most powerful and popular SMT solvers is **Z3** [de Moura and Bjorner, 2008b]: it offers many theories of the SMT-LIB 2.0 standard [Barrett et al., 2010], extensions such as algebraic data types, the combination of these, and quantified formulas.

Notes. The definition of soundness and completeness is the same for SMT solvers as for SAT solvers (cf. Note 3.7), except that signatures, formulas and structures are no longer from propositional logic, but some SMT logic. The corresponding theory (combination) is decidable iff the SMT logic has a terminating, sound, and complete SMT solver [Kroening and Strichman, 2008].

3.4. Automata Theory

Automata theory is another basic core in this thesis besides propositional logic, since most formal methods in this thesis apply automata theoretic structures and concepts, mainly based on various kinds of transition systems. The terminology and details of these transition systems differ in literature, especially between different fields, such as model checking and model-based testing. Thus, transition systems are described and motivated thoroughly in this section, uniting the common parts – partly through generalizations.

Roadmap. Subsec. 3.4.1 introduces transition systems, Subsec. 3.4.2 relates them to finite state machines (which are often excluded when talking about general transition systems). Finally, Subsec. 3.4.3 introduces specification languages for our transition systems.

3.4.1. Transition Systems

This subsection introduces four kinds of transition systems, as depicted in Fig. 3.1 on page 37, formalisms over them, and how they relate to FSMs.

Transition Systems. A system being inspected by automata theoretic methods describes a real world system that has states as well as behavior, i.e., state changes. So an **atomic behavior** is a transition from one state to another, and often called **action**. The change in state is called **side effect** of the action. Therefore, a real world system can be modeled by a transition system (**TS**), as defined in Def. 3.15.

Definition 3.15. A **transition system** $\mathcal{S} = (S, T)$ has
- a nonempty, countable set $\boldsymbol{S}$ of **states**;
- a **relation** $\boldsymbol{T} \subseteq S^2$ of **transitions**.

$\mathbb{S}_{\boldsymbol{TS}}$ denotes the set of all transition systems, $\mathbb{S}_{\boldsymbol{TS,finite}}$ its subset of transition systems with finite S.

$\mathcal{S}$ is often also called an **(unlabeled) state transition system** or **(unlabeled) Kripke frame**. Usually some starting states or default states are given. Therefore, a transition system can additionally specify a subset $\boldsymbol{S^0} \subseteq S$ of **initial states** – either implicitly or as an additional annotation: (S, T, S^0). Therefore, all following structures that contain (S, T) and implicit initial states can instead contain (S, T, S^0) (cf. (S, T, Σ, I, S^0) in Def. 3.18, $(S, \mathfrak{T}, L, S^0)$ in Def. 3.19, $(S, \mathfrak{T}, L, \Sigma, I, S^0)$ in Def. 3.25). Often we have a

unique initial state (i.e., $|S^0| = 1$), which we call $\boldsymbol{init_{\mathcal{S}}}$ or just $\boldsymbol{init}$ if $\mathcal{S}$ is clear from the context.

Notes. S^0 should not be confused with $\{\epsilon\}$ (cf. Sec. 1.3).

$\mathbb{S}_{TS}$ includes transition systems with countable sets of states S, to be sufficiently general for model-based testing, covered in Part III. For other chapters, e.g., Chapter 5 and Chapter 7, we need the restriction to $\mathbb{S}_{TS,finite}$.

For many continuous systems, e.g. most real-time systems, the set of states can be reduced to a countable set by abstractions, e.g. via time regions [Alur and Dill, 1990; Bouyer and Laroussinie, 2010].

Definition 3.16. Let $\mathcal{S} = (S, T, S^0) \in \mathbb{S}_{TS}$, $l \in \omega + 1$, $\pi := (s_i)_{i \in [0,\ldots,1+l)}$ a sequence of states, and $s \in S$.

If $\forall i \in [1, \ldots, 1 + l) : (s_{i-1}, s_i) \in T$, then π is called a **path** in $\mathcal{S}$ of **length** l starting from s_0. For the length l, we write $\boldsymbol{|\pi|}$**.** Furthermore:

- $\boldsymbol{paths(\mathcal{S}, s)}$:= the set of all paths in $\mathcal{S}$ starting in s;
- $\boldsymbol{paths^{fin}(\mathcal{S}, s)}$:= $\{\pi \in paths(\mathcal{S}, s) \big| |\pi| \in \mathbb{N}_{\geq 0}\}$, the subset of $paths(\mathcal{S}, s)$ with all finite paths;
- $\boldsymbol{paths^{\omega}(\mathcal{S}, s)}$:= $paths(\mathcal{S}, s) \setminus paths^{fin}(\mathcal{S}, s)$;
- $\boldsymbol{paths^{fin}_{max}(\mathcal{S}, s)}$:= $\{\pi \in paths^{fin}(\mathcal{S}, s) \big| \nexists s' \in S : (s_{|\pi|}, s') \in T\}$, the subset of $paths^{fin}(\mathcal{S}, s)$ with all **maximal paths**;
- $\boldsymbol{paths_{max}(\mathcal{S}, s)}$:= $paths^{fin}_{max}(\mathcal{S}, s) \dot\cup paths^{\omega}(\mathcal{S}, s)$;
- $\boldsymbol{paths_{<max}(\mathcal{S}, s)}$:= $paths(\mathcal{S}, s) \setminus paths_{max}(\mathcal{S}, s)$, the set of all (finite) **non-maximal paths**.

Without the state given as parameter, all initial states are used:
$\boldsymbol{paths(\mathcal{S})} := paths(\mathcal{S}, S^0)$, likewise with $\boldsymbol{paths^{fin}(\mathcal{S})}$, $\boldsymbol{paths^{\omega}(\mathcal{S})}$, $\boldsymbol{paths_{max}(\mathcal{S})}$, $\boldsymbol{paths_{<max}(\mathcal{S})}$, $\boldsymbol{paths^{fin}_{max}(\mathcal{S})}$.

The **source** of $\pi \in paths(\mathcal{S}, s)$ is $\boldsymbol{source(\pi)} := s$; the **destination** of $\pi \in paths^{fin}(\mathcal{S}, s)$ is $\boldsymbol{dest(\pi)} := s_{|\pi|}$.

For $k \in \mathbb{N}_{\geq 0}$, $\boldsymbol{\pi_{\leq k}} := (s_i)_{i \in [0,..,1+min(k,|\pi|))}$ is called a **prefix** of π, similarly $\boldsymbol{\pi_{\geq k}} := (s_i)_{i \in [min(k,|\pi|)),..,1+|\pi|)}$ a **suffix** of π.

For $\pi' = (s'_i)_i \in paths^{fin}(\mathcal{S}, s)$ and $\pi'' = (s''_i)_i \in paths(\mathcal{S}, dest(\pi'))$, the **concatenated path** $\pi' \cdot \pi'' \in paths(\mathcal{S}, s)$ is the path π with $\pi_{\leq |\pi'|} = \pi'$ and $\pi_{\geq |\pi'|} = \pi''$.

A path $\pi \in paths^{\omega}(\mathcal{S}, s)$ is called a **cycle** iff $\exists i \in \mathbb{N}_{>0} : \pi = (\pi_{\leq i})^{\omega}$, and more generally a **lasso** iff $\exists j \in \mathbb{N}_{\geq 0} : \pi_{\geq j}$ is a cycle. A **single unwinding** of a lasso π is the path $\pi_{\leq k}$ for the smallest $k \in \mathbb{N}_{>0}$ such that $\exists j \in [0, \ldots, k-1] : \pi = \pi_{\leq j} \cdot ((\pi_{\leq k})_{\geq j})^{\omega}$. If s_k occurs more than twice in $\pi_{\leq k}$, the size of the cycle, i.e., $k - j$, has to be given as additional information to reconstruct π from $\pi_{\leq k}$.

Note. For path π and $j > |\pi| : \pi_{\geq j} = (s_{|\pi|})$. If $|\pi| = 0$, it contains exactly one state.

Definition 3.17. Let $\mathcal{S} = (S, T, S^0) \in \mathbb{S}_{TS}, s, s' \in S$ and $\pi \in paths(\mathcal{S}, s)$. Then we define for simpler notation (similar to Kleene's closure operators):

- $s \rightarrow s'$ $:\Leftrightarrow\ (s, s') \in T$;
- $s \rightarrow s'$ $:= \{(s, s')\} \cap T$, but only when unambiguous to the previous line;
- $s \xrightarrow{\pi}{}^{*} s'$ $:\Leftrightarrow \pi \in paths^{fin}(\mathcal{S}, s)$ and $dest(\pi) = s'$;
- $s \rightarrow^{*} s'$ $:\Leftrightarrow \exists \pi \in paths^{fin}(\mathcal{S}, s) : s \xrightarrow{\pi}{}^{*} s'$(the reflexive, transitive closure for T);
- $s \rightarrow^{*} s'$ $:= \{\pi \in paths^{fin}(\mathcal{S}, s) \mid s \xrightarrow{\pi}{}^{*} s'\}$, but only when unambiguous;
- $s \xrightarrow{\pi}{}^{+} s'$ and (transitive closure) $s \rightarrow^{+} s'$ analogously, but with $|\pi| > 0$;
- $s \rightarrow$ $:\Leftrightarrow\ \exists s' \in S : s \rightarrow s'$;
- $s \xrightarrow{\pi}{}^{+}$ $:\Leftrightarrow\ \exists s' \in S : s \xrightarrow{\pi}{}^{+} s'$;
- $\boldsymbol{dest(s, \rightarrow)} := \{s' \in S \mid s \rightarrow s'\}$, the states **directly reachable** from s;
- $\boldsymbol{dest(s, \rightarrow^{*})}$ and $\boldsymbol{dest(s, \rightarrow^{+})}$ analogously, the states **reachable** from s;
- $\boldsymbol{dest_{\mathcal{S}}(\cdot, \cdot)}$ is used instead if $\mathcal{S}$ is not clear from the context;
- $\boldsymbol{branch_{\mathcal{S}}}$ $:= \underset{s \in S}{supremum}(|dest(s, \rightarrow)|)$;
- $\boldsymbol{depth_{\mathcal{S}}}$ $:= \underset{\pi \in paths_{max}(\mathcal{S})}{supremum}(|\pi|)$;
- $s \rightarrow s' \rightarrow s''$ $:\Leftrightarrow\ s \rightarrow s'$ and $s' \rightarrow s''$;
- $s \rightarrow s' \rightarrow s''$ $:= \{(s, s') \cdot (s', s'') \in paths(\mathcal{S}, s)\}$, but only when unambiguous;
- all combinations with $\rightarrow, \rightarrow^{*}, \rightarrow^{+}, \xrightarrow{\pi}{}^{*}, \xrightarrow{\pi}{}^{+}$ analogously.

For each of these relations, s is called its **source** and s' its **destination**. For $s \rightarrow s'$, the **sibling transitions** are $\{(s, s'') \in T | s'' \in dest(s, \rightarrow)\}$, the **sibling states** of s' are $dest(s, \rightarrow)$.

The **state space** of $\mathcal{S} = (S, T, S^0) \in \mathbb{S}_{TS}$ refers to the states $s \in S$ that are considered; depending on the approach (cf. Chapter 5), this is the set S, $dest(S^0, \rightarrow^{*}) \subseteq S$, also written $\boldsymbol{S_{\rightarrow^{*}}}$, or some subset thereof. Likewise, we restrict T to $\boldsymbol{T_{\rightarrow^{*}}} := T \cap S^2_{\rightarrow^{*}}$. Finally, $\boldsymbol{\mathcal{S}_{\rightarrow^{*}}} := (S_{\rightarrow^{*}}, T_{\rightarrow^{*}}, S^0)$.

$\boldsymbol{|\mathcal{S}|}$ is called **size** or **complexity of** $\mathcal{S}$, and is defined as $|S| + |T|$. Therefore, $|\mathcal{S}_{\rightarrow^{*}}| = |S_{\rightarrow^{*}}| + |T_{\rightarrow^{*}}|$, which is in $\Theta(|T_{\rightarrow^{*}}|)$.

$(S, T) \in \mathbb{S}_{TS}$ is **deterministic** iff T is a partial function (i.e., right-unique or uniquely defined): $\forall s \in S : \big|dest(s, \rightarrow)\big| \leq 1$. Then $\forall s \in S : |paths_{max}(\mathcal{S}, s)| = 1$ and $\boldsymbol{T(s)}$ is the element s' if $s \rightarrow s'$, and undefined if $s \not\rightarrow$.

Note. A transition system is often called a **model** because it models the functional behavior of a system from the real world. This thesis does not use this terminology to avoid ambiguity with the logical meaning of the term "model".

Kripke Structures. States of a system from the real world exhibit various **properties**, which we also want to model. We formalize such a property with a **propositional variable** whose interpretation depends on the state. To reflect this, we extend TSs to (unlabeled) Kripke structures:

Definition 3.18. A **Kripke structure** (S, T, Σ, I) has
- a transition system $(S, T) \in \mathbb{S}_{TS}$;
- a **signature** $\boldsymbol{\Sigma}$, which is a countable set of propositional variables;

- an **interpretation function** $\boldsymbol{I} : \Sigma \times S \to \mathbb{B}$.

$\mathbb{S}_{\boldsymbol{Kripke}}$ denotes the set of all Kripke structures, $\mathbb{S}_{\boldsymbol{Kripke,1}}$ the subset of Kripke structures with $|S^0| = 1$, $\mathbb{S}_{\boldsymbol{Kripke,finite}}$ the subset of Kripke structures with S and Σ finite.

For ease of use, we regard **true** as 1, **false** as 0 and thus the **support** of the function $I(p,\cdot)$ as $\boldsymbol{supp(I(p,\cdot))} := \{s \in S | I(p,s) = \textbf{true}\}$; likewise, $\boldsymbol{supp(I(\cdot,s))} := \{p \in \Sigma | I(p,s) = \textbf{true}\}$.

Labeled Transition Systems. To be able to determine what a transition t describes, e.g., what condition or action of the real world system, t can be labeled with information. If multiple labels apply, t can be replicated. Often actions are defined by (basic) statements in a specification language (cf. Subsec. 3.4.3), which are frequently used as labels. Extending TSs with labels results in LTSs:

Definition 3.19. A **labeled transition system** (**LTS**) $(S, \mathfrak{T}, L)$ has

- a nonempty, countable set S of states;
- a countable set $\boldsymbol{L}$ of **labels**;
- a set $\boldsymbol{\mathfrak{T}} \subseteq S \times L \times S$ of **labeled transitions**.

$\mathbb{S}_{\boldsymbol{LTS}}$ denotes the set of all LTSs, $\mathbb{S}_{\boldsymbol{LTS,finite}}$ its subset of LTSs with S and L finite.

A TS (S,T) without labels is a special case of an LTS with $L = \{\epsilon\}$. Conversely, an LTS is a TS if the labels are ignored. More formally, we can perform a transformation as given in Def. 3.20.

Definition 3.20. $\boldsymbol{\mathcal{F}_L} : \mathbb{S}_{LTS} \to \mathbb{S}_{TS}, (S, \mathfrak{T}, L) \mapsto (S, \bigcup_{l \in L} \{(s,s') | (s,l,s') \in \mathfrak{T}\})$ is called the **forgetful transformation for labels** .

The notations $\to, \to^+, \to^*, dest(s,\to), dest(s,\to^+), dest(s,\to^*), S_{\to^*}, T_{\to^*}, |\mathcal{S}|$ defined in Def. 3.17 can hence also be applied for LTSs, meaning that arbitrary labels are allowed for the transitions.

Definition 3.21. Let $\mathcal{S} = (S, \mathfrak{T}, L) \in \mathbb{S}_{LTS}, l \in L, s, s' \in S$. Then

- $s \xrightarrow{l} s' :\Leftrightarrow (s,l,s') \in \mathfrak{T}$;
- $s \xrightarrow{l} \ :\Leftrightarrow \exists s'' \in S : (s,l,s'') \in \mathfrak{T}$;
- label l is **enabled** in s iff $s \xrightarrow{l}$, otherwise l is **blocked** in s;
- $\boldsymbol{enabled_{\mathcal{S}}(s)} := \{l \in L | s \xrightarrow{l}\}$; If $\mathcal{S}$ is clear from the context, $\boldsymbol{enabled(s)} := enabled_{\mathcal{S}}(s)$;
- $\boldsymbol{\mathfrak{T}_{\to^*}} := \mathfrak{T} \cap (S_{\to^*} \times L \times S_{\to^*})$.

Definition 3.22. Let $(S, \mathfrak{T}, L) \in \mathbb{S}_{LTS}, l \in \omega + 1, \pi := (s_{i-1} \xrightarrow{l_i} s_i)_{i \in [1,\dots,1+l)}$ a sequence of labels, each with its source state and destination state. Then π is called a **path** in $(S, \mathfrak{T}, L)$ of **length** l starting from s_0. For the length l, we write $\boldsymbol{|\pi|}$.

All terminology of Def. 3.16 can be inherited: $\boldsymbol{paths}$, $\boldsymbol{paths^{fin}}$, $\boldsymbol{paths^{\omega}}$, $\boldsymbol{paths_{max}}$, $\boldsymbol{paths_{<max}}$, $\boldsymbol{paths^{fin}_{max}}$, $\boldsymbol{source(\pi)} := s_0$, $\boldsymbol{dest(\pi)} := s_{|\pi|}$, **prefix** $\boldsymbol{\pi_{\leq k}}$, **suffix** $\boldsymbol{\pi_{\geq k}}$, **concatenation** $\pi \cdot \pi'$, **cycle**, and **lasso**.

Definition 3.23. Let $(S, \mathfrak{T}, L) \in \mathbb{S}_{LTS}$ and $\pi = (s_{i-1} \xrightarrow{l_i} s_i)_{i \in [1,\dots,1+|\pi|)}$ a path in $(S, \mathfrak{T}, L)$. The **trace** of π, $\boldsymbol{trace(\pi)}$, is the sequence $(l_i)_{i \in [1,\dots,1+|\pi|)}$. Its **length** is

$|\boldsymbol{trace(\pi)}| := |\pi|$. All terminology in Def. 3.22 except *source*, *dest*, cycle, and lasso can be lifted from paths to traces: $\boldsymbol{traces}$, $\boldsymbol{traces^{fin}}$, $\boldsymbol{traces^{\omega}}$, $\boldsymbol{traces_{max}}$, $\boldsymbol{traces_{<max}}$, $\boldsymbol{traces^{fin}_{max}}$ (also called **complete traces**), **prefix** $\boldsymbol{(l_i)_{i\leq k}}$, **suffix** $\boldsymbol{(l_i)_{i\geq k}}$, and **concatenation** $t \cdot t'$ (which no longer makes restrictions via *source* and *dest*).

Definition 3.24. Let $\mathcal{S} = (S, \mathfrak{T}, L, S^0) \in \mathbb{S}_{LTS}$, $s, s', s'' \in S$, and $\sigma, \sigma' \in L^* \cup L^\omega$. Then we define for simpler notation (similar to Def. 3.17):

- $s \xrightarrow{\sigma}{}^*$ $:\Leftrightarrow \sigma \in traces(\mathcal{S}, s)$;
- $s \xrightarrow{\sigma}{}^+$ $:\Leftrightarrow s \xrightarrow{\sigma}{}^*$ and $\sigma \neq \epsilon$;
- $s \xrightarrow{\sigma}{}^* s'$ $:\Leftrightarrow \exists \pi \in paths^{fin}(\mathcal{S}, s) : trace(\pi) = \sigma$ and $dest(\pi) = s'$; the **reflexive, transitive closure** for $\mathfrak{T}$,
- $s \xrightarrow{\sigma}{}^+ s'$ $:\Leftrightarrow s \xrightarrow{\sigma}{}^* s'$ and $\sigma \neq \epsilon$, the **transitive closure** for $\mathfrak{T}$;
- $\boldsymbol{branch_{\mathcal{S}}}$ $:= \max_{s \in S}(\left|\{(l, s') \in L \times S \mid s \xrightarrow{l} s'\}\right|)$;
- $s \xrightarrow{\sigma}{}^* s' \xrightarrow{\sigma'}{}^* s'' :\Leftrightarrow s \xrightarrow{\sigma}{}^* s'$ and $s' \xrightarrow{\sigma'}{}^* s''$;
- all combinations with $\xrightarrow{\sigma}{}^*$ *and* $\xrightarrow{\sigma}{}^+$ analogously.

Notes. In Def. 3.22, a path of length 0 starting from s is the empty sequence. Alternatively, we can define that path, as for TSs, to be the singleton sequence (s).

For $\mathcal{S} = (S, \mathfrak{T}, L) \in \mathbb{S}_{LTS}$, $s_0 \in S$, path $\pi \in paths^{fin}(\mathcal{S}, s_0)$ can now also be written as $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} \ldots \xrightarrow{l_{|\pi|}} s_{|\pi|}$.

$(S, \mathfrak{T}, L) \in \mathbb{S}_{LTS}$ is **deterministic** iff $\forall l \in L : \xrightarrow{l}$ is a partial function. Then $\boldsymbol{l(s)}$ is the element s' for $s \xrightarrow{l} s'$, and undefined if $s \not\xrightarrow{l}$.

Let $\mathcal{S}$ be a deterministic LTS. Then $\mathcal{F}_L(\mathcal{S})$ need not be a deterministic unlabeled TS. Fig. 4.2 on page 84 is a counterexample.

Labeled Kripke Structures. Finally, transition systems with both interpretations on states and labels on transitions are considered:

Definition 3.25. A **labeled Kripke structure** $(S, \mathfrak{T}, L, \Sigma, I)$ is the integration of an LTS $(S, \mathfrak{T}, L)$ and a Kripke structure (S, T, Σ, I) with $(S, T) = \mathcal{F}_L(S, \mathfrak{T}, L)$.

$\boldsymbol{\mathbb{S}_{Kripke,labeled}}$ denotes the set of all labeled Kripke structures, $\boldsymbol{\mathbb{S}_{Kripke,labeled,finite}}$ its subset of labeled Kripke structures with both their LTS and Kripke structure finite.

Note. All mentioned structures are transition systems: they have a set of states S and a set of transitions T (or labeled transitions, which can be forgetfully transformed into T). Hence this thesis uses the general terminology **transition system** for all TS, Kripke structures, LTS and labeled Kripke structures, the terminologies **Kripke frame** to exclude interpretation functions and **unlabeled** to exclude labels. These relationships are depicted in Fig. 3.1. As has been defined, further predicates like **finite**, 1, and **deterministic** restrict these sets, e.g., $\mathbb{S}_{Kripke,labeled,deterministic} = \{\mathcal{S} \in \mathbb{S}_{Kripke.labeled} \mid \mathcal{S} \text{ is deterministic}\}$. If a predicate is allowed but not required, we write it in round brackets, e.g., $\mathbb{S}_{(L)TS}$.

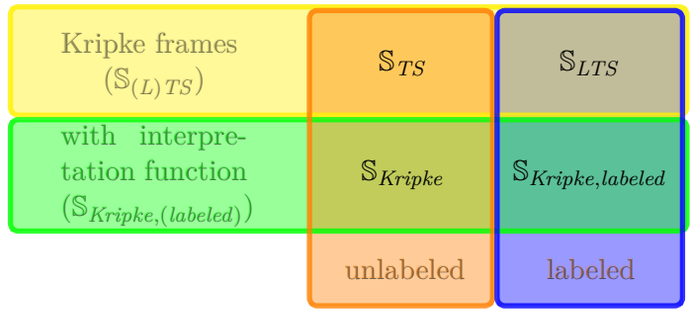


Figure 3.1.: Relationships between various transition systems

Example. Fig. 4.2 on page 84 shows a simple graphical representation of a Kripke structure if transitions t_1 and t_2 are not labeled. If they are labeled (e.g. t_1 with skip and t_2 with toggle), we have a labeled Kripke structure. If the states are not annotated (i.e., $\neg q$ and q are omitted), we have a transition system or labeled transition system, respectively.

3.4.2. Finite State Machines

Def. 3.26 defines **finite automata**, also called finite state machines (**FSMs**). Thereafter, the relation between transitions systems (cf. Subsec. 3.4.1) and FSMs is shown.

Definition 3.26. A **finite state machine** $\mathcal{A} = (Q, \Delta, A, Q^0, F)$ has
- a finite, nonempty set of **states** Q;
- a finite, nonempty **alphabet** set A;
- a **transition relation** $\Delta \subseteq Q \times A \times Q$;
- a nonempty set of **initial states** $Q^0 \subseteq Q$;
- a set of **final states** (or **accepting states**) $F \subseteq Q$.

$\mathbb{S}_{FSM}$ denotes the set of all FSMs.

Example. Fig. 3.2 shows an example for a FSM with $Q = \{q_0, q_1\}, A = \{l_0, l_1, l_2\}, \Delta = \{(q_0, l_0, q_1), (q_1, l_1, q_0), (q_1, l_2, q_0)\}$ and $Q^0 = F = \{q_0\}$.

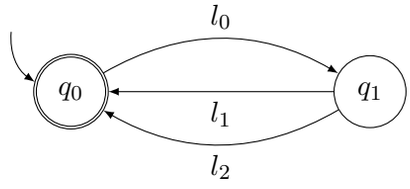


Figure 3.2.: FSM

Using the canonical embedding c from Def. 3.27, we can relate FSMs to finite labeled Kripke structures and thus transfer many definitions for transition systems (cf. Subsec. 3.4.1) to FSMs.

Definition 3.27. The **canonical embedding** $\boldsymbol{c}$: $\mathbb{S}_{FSM} \rightleftharpoons (\mathbb{S}_{Kripke,labeled,finite}$ with $|\Sigma| = 1)$ maps $(Q, \Delta, A, Q^0, F) \mapsto (S, \mathfrak{T}, L, \Sigma, I, S^0)$ with $S = Q, S^0 = Q^0, L = A, \mathfrak{T} = \Delta$ and $supp(I(accept, \cdot)) = F$ (for $\Sigma = \{accept\}$).

Using the canonical embedding yields the following definitions for FSMs:

- **paths** and all the sets defined on them, **source** and **destination** of (finite) paths, **concatenation**, **prefix** and **suffix** of paths, **cycle** and **lasso** (cf. Def. 3.22);
- **traces** and all the sets defined on them, as well as **prefix**, **suffix** and **concatenation** on traces (cf. Def.3.23);
- the relations $\rightarrow$, $\rightarrow^+$, and $\rightarrow^*$ on paths, as well as $\boldsymbol{dest(s, \rightarrow)}$ and analogously for $\rightarrow^+$ and $\rightarrow^*$, the relations $\xrightarrow{\sigma}{}^*$ and $\xrightarrow{\sigma}{}^+$ on traces, and finally $\boldsymbol{branch_S}$ and $\boldsymbol{|S|}$ (cf. Def. 3.24 and Def. 3.20).

Often, embeddings of unlabeled finite Kripke structures with finite Σ into FSMs is required (cf. Subsec. 5.3.2 and [Clarke et al., 1999b, Fig. 9.2]), for which we define $K_{fin}2FSM$ in Def. 3.28.

Definition 3.28. The embedding $K_{fin}2FSM : \mathbb{S}_{Kripke,finite} \hookrightarrow \mathbb{S}_{FSM}$ maps $(S, T, \Sigma, I, S^0) \mapsto (Q, \Delta, A, Q^0, F)$ with $Q = S \dot\cup \{\boldsymbol{init_{\mathcal{A}}}\}, A = 2^\Sigma, Q^0 = \{init_{\mathcal{A}}\}, F = Q$, and $\Delta = \{init_Q \xrightarrow{supp(I(\cdot,s))} s | s \in S^0\} \dot\cup \{s \xrightarrow{supp(I(\cdot,s'))} s' | (s, s') \in T\}$.

Note. $K_{fin}2FSM : \mathbb{S}_{Kripke,finite} \hookrightarrow \mathbb{S}_{FSM}$ is not surjective since the transformation between T and Δ cannot handle states of FSMs having multiple incoming transitions with different labels, e.g., as in Fig. 3.2.

$K_{fin}2FSM$ can be extended to $\mathbb{S}_{Kripke,labeled,finite}$ by adding (further) labels to the Kripke structure and to the FSM.

Example. Fig. 3.3 shows the FSM $K_{fin}2FSM(\mathcal{S})$ for the unlabeled finite Kripke structure $\mathcal{S}$ from our example in Fig. 4.2 on page 84 (i.e., for $t_1 = t_2 = \epsilon$).

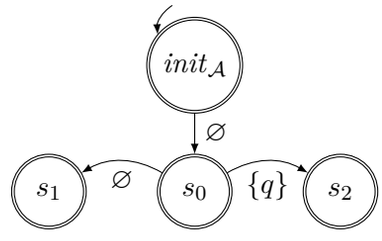


Figure 3.3.: FSM $K_{fin}2FSM(\mathcal{S})$ of the unlabeled finite Kripke structure from Fig. 4.2

3.4.3. System Specification Description Languages for Transition Systems

Subsec. 3.4.2 defined various kinds of transition systems, which are used as system specifications. **System specification description languages** (**specification language** for short) describe such specifications and should efficiently model states, transitions, and nondeterminism [Frappier et al., 2010; ETSI, European Telecommunications Standards Institute, 2011]. From such descriptions, unlabeled transition systems, Kripke

structures, LTSs or labeled Kripke structures can be derived. Depending on the preferred transition system and its utilization, different language paradigms are used. We describe the most popular paradigms, which are direct enumeration, process calculus, and programming languages. Often a mix of paradigms and variables are used for concise modeling, as the examples below in this subsection show.

Further paradigms exists, for instance graphical modeling [ETSI, European Telecommunications Standards Institute, 2011], which must be combined with another paradigm and is only marginally considered in this thesis. Another example is **design-by-contract** (**DbC**), which is mainly applied to specify program functions – in this thesis, too, in a syntax similar to JML [Leavens et al., 2008; Schmitt and Tonin, 2007]. But DbC is also used occasionally to specify transition systems [Große-Rhode, 2001; Nebut et al., 2003; de Oliveira Jr et al., 2007]. Similar to a Hoare triple Hoare [1969], a **code contract** (**contract** for short) [Meyer, 1992] specifies (usually in FOL) how a transition or function t changes the state: an assertion **PRE**, called **precondition**, defines when t is applicable, another assertion **POST**, called **postcondition**, defines t's effect. Furthermore, assertions **INV**, called **invariants**, define constraints that must always hold (at specific, visible states). Contracts thus define a formal interface: t's caller guarantees that it only calls t if PRE holds. In this case, the implementation of t guarantees that POST holds upon t's termination; otherwise, no guarantee is given. Invariants define invariant properties of the system's states and are preserved by the implementation.

Roadmap. Firstly, specification languages based on direct enumeration of all elements of the transition system are introduced. Then we give an overview of symbolic transition systems, which are based on direct enumeration and achieve compact descriptions with the help of variables as well as constraints and updates formulated in first order logic. They are the main specification language used for model-based testing in this thesis (cf. Part II). Thereafter, specification languages based on process algebra are introduced, mainly covering channels, processes, and their composition. Finally, specification languages that are syntactically similar to common programming languages are introduced, covering procedural and guarded command languages. Having covered these language paradigms, we describe PROMELA, which is based on process algebra, the procedural programming language C and the guarded command language. It is the main specification language used for model checking in this thesis (cf. Part III).

Specification Languages Based on Enumeration

Specification languages based on enumeration describe the specification by enumerating the states and transitions, either explicitly or implicitly, e.g., with the help of variables and how they change. If the state space is infinite, its specification must still be finite to be processable (e.g., lazily via OPEN/CAESAR [Garavel, 1998; Garavel et al., 2011]).

Example 3.29. The **Binary Coded Graph** (**BCG**) format from CADP [Bowman and Gómez, 2006; URL:CADP; Garavel et al., 2011] can describe unlabeled as well as labeled (and probabilistic) transition systems. The transition system is enumerated explicitly, e.g., by firstly enumerating all states and then for each state all transitions. Because of this enumeration, only finite state spaces can be described; but with the help of minimization and compression, up to 10^{13} states and transitions can be stored.

An example for an implicit enumeration is the **Extended Table Format** (**ETF**) [Blom et al., 2010]: It is the symbolic labeled Kripke structure format that PINS (see Subsec. 5.5.2) can read and write. Because of PINS's design, ETF needs only enumerate all short vectors for all groups, resulting in efficient compression, e.g. 0.57 billion states in a 1.6 KiB ETF file.

Another example for an implicit enumeration is a symbolic transition system, used frequently in this thesis and described in detail in the next subsection.

Symbolic Transition Systems, a Specification Language Based on Enumeration

A symbolic transition system (**STS**) extends an LTS with variables and conditions in FOL [Frantzen et al., 2006; Frantzen, 2016] (cf. Fig. 9.1 on page 232 and Fig. B.1 on page 386). The variables, respective conditions, explicitly add data, respective data-dependent control flow, to LTSs. STSs can be considered a specification language for LTSs based on enumeration.

Syntax.

Def. 3.30 defines the syntax of STSs, with Σ being a signature for first order logic.

Definition 3.30. A **symbolic transition system** $(S, \rightarrow, L, \mathcal{V}, \mathcal{I}, S^0)$ has

- a nonempty, countable set $\boldsymbol{S}$ of **abstract states**, also called **locations**;
- a countable set $\boldsymbol{L}$ of **labels**, also called **gates**. We define $\boldsymbol{type} : L \to 2^{\mathcal{I}}$;
- a countable set $\boldsymbol{\mathcal{V}}$ of **state variables**, also called **location variables**;
- a countable set $\boldsymbol{\mathcal{I}}$ of **interaction variables**, with $\mathcal{V} \cap \mathcal{I} = \varnothing$ and $\boldsymbol{Var} := \mathcal{V} \dot\cup \mathcal{I}$;
- a relation $\rightarrow \subseteq S \times L \times \mathrm{FORM}_\Sigma(Var) \times \mathrm{TERM}_\Sigma(Var)^{\mathcal{V}} \times S$, called **switch relation**. An element $(s, l, F, \rho, s') \in \rightarrow$ is called **switch** and must meet $F \in \mathrm{FORM}_\Sigma(\mathcal{V} \dot\cup type(l))$ and $\rho \in \mathrm{TERM}_\Sigma(\mathcal{V} \dot\cup type(l))^{\mathcal{V}}$; we write $s \xrightarrow{l,F,\rho} s'$ or $s \xrightarrow{l<type(l)>,[F],\rho} s'$ or $s \xrightarrow{l<type(l)>.[F]\{\rho\}} s'$, and call F **switch restriction** or **guard** and ρ **update mapping** or just **update**. Tautological guards and empty updates can be omitted;
- a set of initial states $\boldsymbol{S^0} \subseteq S$.

$\boldsymbol{\mathbb{S}_{STS}}$ denotes the set of all STSs.

Notes. This thesis defines the codomain of $type(\cdot)$ as set, not as sequence, leaving open how arguments are mapped to interaction variables (e.g., via positional or named arguments [Rytz and Odersky, 2010]).

It is irrelevant whether bound variables are in Var; for simplicity and to be consistent with other literature [Frantzen, 2016], we define them to be.

Some literature deviates from Def. 3.30 when defining STSs, e.g., input output symbolic transition systems (IOSTS) in [Rusu et al., 2000], where the set S of locations must be finite.

Semantics.

Def. 3.31 defines the semantics of an STS $\mathcal{S}$ by expanding $\mathcal{S}$ to an LTS, depending on a variable initialization.

Definition 3.31. Let $\mathcal{S} = (S, \rightarrow, L, \mathcal{V}, \mathcal{I}, S^0) \in \mathbb{S}_{STS}$ and $\mathcal{D} = (D, I)$ a first order structure. Then:

- a **variable initialization** for $\mathcal{S}$ is a function $\mathcal{V}^0 : \mathcal{V} \to D$, which initializes all state variables in all states of S^0. $\boldsymbol{\mathcal{V}^d}$ is the **default variable initialization**, which initializes each variable with null (e.g., 0 if $D \subseteq \mathbb{N}$ and ϵ if D are strings);
- the **interpretation** (also called **expansion**) of $\mathcal{S}$ for the variable initialization $\mathcal{V}^0$ is the (usually infinite) LTS $[[\boldsymbol{\mathcal{S}_{\mathcal{V}^0}}]] := (S \times D^{\mathcal{V}}, L_{LTS}, \to_{LTS}, S^0_{LTS})$ with
  - $L_{LTS} := \bigcup_{l \in L} \{l\} \times D^{|type(l)|}$;
  - $\to_{LTS} \subseteq (S \times D^{\mathcal{V}}) \times L_{LTS} \times (S \times D^{\mathcal{V}})$, with $\to_{LTS} := \{(s, \xi, l, \zeta, s', \xi') \mid s, s' \in S,\ \xi, \xi' \in D^{\mathcal{V}},\ (l, \zeta) \in L_{LTS},\ s \xrightarrow{l,F,\varrho} s',\ \mathcal{D}, \xi \dot\cup \zeta \models F,$ and $\xi' = x \mapsto val_{\mathcal{D},\xi\dot\cup\zeta}(\rho(x))\}$;
  - $S^0_{LTS} := \{(s_0, \mathcal{V}^0) | s_0 \in S^0\}$.

  A state $(s, \xi) \in S \times D^{\mathcal{V}}$ is called an **instantiated state**.

Many methods and tools that use STSs expand them into LTSs as described in Def. 3.31 (cf. Sec. 9.4). Hence STSs can be considered as specification language for LTSs based on implicit enumeration. They are the main specification language used for model-based testing in this thesis (cf. Part II).

Specification Languages Based on Process Algebra

Formal methods (especially model checking) often analyze concurrent behaviors, which are hard for humans to understand, but are becoming more and more ubiquitous, and therefore error-prone [Groote and Mousavi, 2014]. **Process algebraic languages** (aka **process calculi**) are able to concisely formalize concurrent systems. A process algebraic expression can be defined via and translated into an LTS [Pierce, 1997; Palamidessi, 2003; Tretmans, 2008; Gorla, 2008, 2010]. Such an expression is a precise and abstract description of how concurrent components communicate and interact with one another. As basic construct for a component, a **process** is defined, which is a free-standing computational activity, running in parallel with other processes and possibly containing many independent sub-processes (cf. [Pierce, 1997]). As basic construct for communication and interaction, **channels** are defined, which are used for sending and receiving **messages**. Channels are **named**, i.e., messages are passed over named locations (so-called interaction points). This communication via messages is called **message passing** [El-Rewini and Abd-El-Barr, 2005; Siegel and Gopalakrishnan, 2011; Raynal, 2013]. Channels with **capacity** greater zero are called **buffered channels** and **transmit asynchronously**, i.e., temporally decoupled. Messages are stored in a channel, usually in a first in, first out (FIFO) manner. Channels with capacity zero do not have a buffer and therefore provide **synchronous transmission**, i.e., sending and receiving is executed concurrently in one atomic synchronous communication step. Some but not all process algebras support **mobility**, i.e., dynamic evolutions of the systems [Gorla, 2010], e.g., **link mobility** where the topology of communication is dynamic [Milner et al., 1992]. This can for instance be achieved by allowing channels to be passed over channels.

The following list shows the basic operators for process algebras, with P, Q being two processes. The basic operators and their syntax varies between process algebras; we focus on a variant of the π-calculus [Milner et al., 1992] that is small (yet Turing complete) [Pierce, 1997]:

- the **inert process**, often written as **0**, does nothing;

- the **input prefix**, often written $\boldsymbol{x(y)}$, waits to read a value y from named channel x;
- the **output prefix**, often written $\boldsymbol{\overline{x}\langle y\rangle}$, waits to send value y along named channel x;
- the **parallel composition** of P and Q, often written $\boldsymbol{P \mid Q}$, results in simultaneous computation of P and Q: If an execution step of a process has synchronous communication, the processes synchronize, i.e., execute concurrently in one atomic step. Otherwise, the execution step is independent of all other processes, so parallel execution of processes with independent execution steps can yield any interleaving, i.e., any possible order of these independent steps;
- the **replication** of P, often written $!P$, denotes an infinite number of copies of P, all running in parallel;
- the **restriction** of P for x, often written as $\boldsymbol{(\nu x); P}$ restricts the visibility of x to P, therefore guaranteeing that communication within P can be performed in isolation.

Besides these basic operators, there are several other operators. Whether they increase expressivity depends on the requirements on encoding these operators with the basic operators, and on the precise definition of the operators and of equivalence [Palamidessi, 2003; Gorla, 2008, 2010]. With P, Q being two processes, the most common further operators are:

- the **sequential composition** of P and Q, often written as $\boldsymbol{P; Q}$, means that Q is executed only after successful termination of P;
- the **polyadic communication prefix** $\overline{x} < y_1, \ldots, y_n >; P$ helps send the tuple $< y_1, \ldots, y_n >$ over the channel with interaction point x. This is the same as $(\nu p)\overline{x}p; \overline{p}; y_1; \ldots; \overline{p}y_n; P$. Likewise, $x(y_1, \ldots, y_n); Q$ helps receive the tuple $< y_1, \ldots, y_n >$;
- the **choice** (or **sum**) between P and Q, often written $\boldsymbol{P + Q}$, behaves like either P or Q;
- the **silent prefix**, often written $\boldsymbol{\tau}$ performs the silent action, i.e., does not send or receive a value;
- the replication of P by **renaming**, often written as $P[y/z]$;

The following parallel composition operators are secondary operators implemented with the help of restriction and renaming. They are particularly helpful for our thesis since they determine the degree of synchronization for the parallel composition:

- the **restricted parallel composition** of P and Q on channels $c_1, \ldots, c_n$, often written as $\boldsymbol{P \mid [c_1, \ldots, c_n] \mid Q}$ is the parallel composition where P and Q only synchronize on the channels $c_1, \ldots, c_n$ but not on other channels, i.e., input and output actions over the channels $c_1, \ldots, c_n$ synchronize between P and Q;
- the **synchronous parallel composition** (aka **full parallel composition** or **synchronous product** or **cross product**) of P and Q, often written as $\boldsymbol{P \mid\mid Q}$, is the parallel composition where P and Q synchronize on all common channels, i.e., $P|[C]|Q$ with C being all common channels of P and Q;
- the **asynchronous product** (aka **asynchronous parallel composition**) of P and Q, often written as $\boldsymbol{P \mid\mid\mid Q}$, is the parallel composition where P and Q do not synchronize on any channel, i.e., $P|[\varnothing]|Q$, so the execution steps of P and Q are all interleaved.

Composition is used frequently since it reduces complexity by modularization, i.e., splitting up a problem into components, e.g., with each one dealing with a different concern (see also Sec. 3.6). This modularization enables abstraction (cf. Sec. 3.7), flexible scheduling of tasks (cf. Sec. 3.6), parallelization (cf. Sec. 3.5), fault-tolerance (cf. Note 8.36 and [Cristian, 1993; Lynch, 1996; Engel et al., 2010; Dubrova, 2013]) and is thus helpful to verification and safety-critical applications.

Example. As example of a parallel composition, we define the synchronous parallel composition of two LTSs (also called component automata) in Def. 3.32. This is for instance used by SPIN for verification, with one component automata for the system specification and one for the property to verify (cf. PROMELA below and Subsec. 5.5.1), or for test execution, with a component automata for the system specification and one for the test case (cf. Subsec. 8.7.1).

Definition 3.32. Let $\mathcal{S}_1 = (S_1, \mathfrak{T}_1, L_1, S_1^0)$, $\mathcal{S}_2 = (S_2, \mathfrak{T}_2, L_2, S_2^0) \in \mathbb{S}_{LTS}$.
Then the synchronous parallel composition $\mathcal{S}_1 || \mathcal{S}_2 := (S, \mathfrak{T}, L, S^0)$ with

- $S := S_1 \times S_2$;
- $S^0 := S_1^0 \times S_2^0$;
- $L := L_1 \cup L_2$;
- $\mathfrak{T} := \{((s_1, s_2), l, (s_1', s_2')) \in S \times L \times S \,|\, (s_1, l, s_1') \in \mathfrak{T}_1 \text{and } (s_2, l, s_2') \in \mathfrak{T}_2,$
  $\text{or } (s_1, l, s_1') \in \mathfrak{T}_1 \text{ and } s_2 = s_2' \text{ and } l \notin L_2,$
  $\text{or } (s_2, l, s_2') \in \mathfrak{T}_2 \text{ and } s_1 = s_1' \text{ and } l \notin L_1\}.$

Example. Most prominent process algebraic languages are CSP [Hoare, 1985], the π-calculus [Milner et al., 1992; Pierce, 1997], mCRL2 [Cranen et al., 2013; Groote and Mousavi, 2014] and LOTOS [SC 7, JTC 1, 1989; Bowman and Gómez, 2006]. The core concepts, however, are also applied in many other languages and tools, for instance the programming language occam (cf. [URL:occam]) and APIs for Java (cf. URL:JCSP) and C++ (cf. URL:C++CSP). Many specification languages for model checking integrate process algebraic concepts, for instance the languages for PRISM (cf. [URL:PRISM]) and SPIN (cf. Subsec. 5.5.1), which even supports link mobility. They will be described below in this subsection.

Specification Languages Based on Programming Languages

Using a different paradigm for specification and implementation can increase design diversity [Avizienis, 1995] and thus reduce the probability of having the same fault in the specification and implementation [Avizienis, 1995]; but it is more convenient for developers and software engineers to stick to a paradigm they are familiar with. Hence specification languages based on programming languages are very popular.

For specification languages describing transition systems, imperative languages are a good match since they are designed to efficiently describe state changes, which can be used to describe transitions. Most specification languages are based on procedural programming languages. Another class of imperative languages are based on the **guarded command language** [Dijkstra, 1975]: Its main construct is the **guarded command**, which has the syntax ⟨guard⟩ → ⟨statement⟩, where ⟨guard⟩ is a Boolean expression without side effects and ⟨statement⟩ is a command that may have side effects and is

executed only if its ⟨guard⟩ evaluates to **true**. The guarded command is often used within control flow constructs and enables nondeterminism.

Example. A specification languages based on C is PROMELA (see below in the subsection), specification languages based on Java are JML [Leavens et al., 2008; URL:JML] and Conformiq Designer (cf. Subsec. 10.3.4).

The model checkers PRISM and DiVinE (cf. Sec. 5.5) bring their own specification languages based on guarded commands (and process algebras). PROMELA, which is based on the C language as well as guarded commands (and process algebra) is described below in this subsection.

PROMELA, a Specification Language Based on Process Algebra as well as Programming Languages

PROMELA [Holzmann, 2004; URL:PROMELAHP; URL:PromelaDatabase; Faragó, 2007] stands for *pro*cess *me*ta *la*nguage and is based on process calculi, the guarded command language and the C programming language. It is the system specification description language of the model checker SPIN (cf. Subsec. 5.5.1).

System specification descriptions in PROMELA consist of variable declarations and process declarations. To give an overview, the language constructs are given here, with details only where it deviates from the C language. For compactness, we describe syntax and semantics together, not in separate paragraphs.

Process type declarations have the form

proctype *proc_name* (*formal_parameters*)
{ *sequence* }

where *sequence* contains local variable declarations and commands in imperative style, as described below.

If the declaration **never** { *sequence* } is present in the model specification, a special process called **never claim** is instantiated once at the beginning. It is used for monitoring every execution step of the system (mainly via user-supplied assertions or acceptance cycle detections, see Subsec. 5.5.1). *sequence* can reference special functions and variables (e.g., np_, see below), and contains commands that should not have side effects.

If the declaration **init** { *sequence* } is present, the **special process init** is instantiated once at the beginning.

Variable declarations outside of all process declarations have a global scope, variable declarations within a process declaration or as formal parameters have a scope local to that process. Variables are declared as bounded integers via primitive types or as complex, composed types. The **primitive types** are **bit**, **bool**, **byte**, **short**, **unsigned** of given bit-width, **pid** (see below), or **enumerations** (i.e., symbolic constants declared by **mtype**). A bounded integer value is equivalent to **false** iff it is 0, and otherwise equivalent to **true**, also called **executable** (see below). **Complex types** are:

- structs (i.e., records), using a globally defined **typedef** type;
- **arrays**, with their size given as constant;
- **channels** of some primitive or complex type, i.e., what messages the channel can store (including channels, see process calculi above).

Variable declarations may contain an **initializer** expression:

- for a primitive type, it evaluates to a value of the same primitive type;
- for an array, it evaluates to the same primitive type that the array is based on, in which case all array elements are assigned that value;
- for a channel, it evaluates to its capacity and an appropriate channel contents.

If no initializer is given, all types but channels are initialized with the value zero (for each of its primitive types). For details and exceptions, see [Holzmann, 2004].

The **basic commands** of PROMELA (called **basic statements** in Holzmann [2004]) are expressions that evaluate to a bounded integer value. An expression blocks if it evaluates to **false**. Otherwise it is enabled ([Holzmann, 2004] uses the notion **executable** instead. The main basic commands are:

- a **variable assignment** with *var* **=** *expr*, which always evaluates to **true** and has as side effect that variable *var* is assigned the evaluation of the expression *expr*;
- **skip**, which always evaluates to **true** and has no side effect;
- **else**, which can be the first expression in at most one choice of a control flow construct and evaluates to **true** iff all other choices evaluate to **false** (see sequences and guards below);
- **run** *proc_name* (*actual_parameters*) , which evaluates to **false** and has no side effect if the maximum number of processes is already running; otherwise, it has the side effect of dynamically creating a new process p of type *proc_name* and evaluates to p's process instantiation number. p's local variables are instantiated and initialized;
- **println ()** , which always evaluates to **true** without any side effects for model checking (during simulation, however, it has the side effect of printing a string on the standard output stream, interpreting its parameters similarly to the programming language C);
- a **send** or **receive** command on a channel c:
  For default settings (in SPIN), the **standard send** command **c!message** evaluates to **false** and has no side effect if c is full or not initialized; otherwise, it evaluates to **true** with the side effect of inserting message at the end of the FIFO buffer of c. If c is a buffered channel, a **sorted send** command inserts the new message according to its (numerical and lexicographical) ordering.
  The **standard receive** command **c?x** evaluates to **false** and has no side effect if c is empty or not initialized; otherwise, it evaluates to **true** and has the side effect of fetching the oldest message from the FIFO buffer of c and assigning it to the variable x. If c is a buffered channel, variations of the command can fetch an arbitrary element from the buffer, which is called **random receive** command, and read a message without removing it from the buffer.

PROMELA's **compound commands** (named **compound statements** in Holzmann [2004]) are composed of other commands via one of the following **control flow constructs**:

- **sequential composition** (aka concatenation) of other commands via **;** to **sequences** of the form $\langle command \rangle \{; \langle command \rangle\}^*$. Sequences are also expressions that evaluate to the bounded integer value of their first basic command, called **guard**. This can be indicated more obviously by replacing the first ; by the equivalent $->$ operator. Thus a sequence is **executable** iff its guard is executable;
- **selection** of the form **if** $\{:: \langle sequence \rangle\}^+$ **fi** , which selects nondeterministically among all executable sequences;

- **repetition** of the form **do** {:: ⟨*sequence*⟩}$^+$ **od**, which repeatedly selects nondeterministically among all executable sequences;
- **break** within a repetition terminates the innermost repetition, i.e., jumps to the command directly after the repetition;
- **exception handling** of the form ⟨$sequence_1$⟩ **unless** ⟨$sequence_2$⟩, which successively executes each command from ⟨$sequence_1$⟩, but beforehand always checks whether ⟨$sequence_1$⟩ is executable. If it is, ⟨$sequence_1$⟩ is no longer executed and control transfers to ⟨$sequence_2$⟩. Otherwise execution of ⟨$sequence_1$⟩ continues. If ⟨$sequence_1$⟩ terminates, the second sequence is ignored.
- **goto** *label_name* to jump to a **label** set by *label_name*:.

Note. Besides the described essential features, PROMELA additionally offers a lot of further features [URL:PROMELAHP; Natarajan and Holzmann, 1997; Weise, 1997; Holzmann, 2004], which can be very helpful in certain situations, but are not frequently used and thus not further covered in this thesis.

The **LTS semantics of PROMELA** are very elaborate if all technical details are considered [URL:PROMELAHP; Natarajan and Holzmann, 1997; Weise, 1997; Holzmann, 2004; Suprapto and Pulungan, 2011; van der Berg and Laarman, 2012]; hence we will not go into those details. By transforming a PROMELA system specification description to an LTS, the overall semantics of PROMELA can be described. Similarly, a labeled Kripke structure can be constructed by using Boolean expressions in PROMELA's syntax, via user-supplied symbol definitions and predefined propositional variables that SPIN offers. For instance, PROMELA labels starting with the string "**end**", "**progress**", or "**accept**" are special, as they mark **valid end states**, **progress states** (encoded in SPIN by variable **np_**), or **acceptance states**, respectively.

For a process p of type *proc_name* (inclusive never claim), a **local LTS** is created: the values of the global variables and p's local variables form p's **local states**. In each local state, the enabled basic commands of *proc_name*, which manipulate the local state, form the labeled transitions in the local LTS. The flow of control in p's execution is guided by the control flow constructs in *proc_name*, so they help to structure the transitions.

Since PROMELA is imperative, p contains a local **program counter** variable (*pc_* that indicates the current position of p's execution, e.g., *pc_* is incremented when executing a basic command in a sequence. Consequently, p's basic commands are implicitly parametrized with its *pc_*.

To instantiate this local LTS the appropriate number of times, copies are created for each process that is an instance of type *proc_name*. p terminates when its program counter reaches the end of the body of *proc_name*. Only processes that have already terminated can be deconstructed, which undoes the construction steps (cf. command `run`) in reversed order.

All copies of the local LTSs are then combined to form the **global LTS**: the local variables of all copies are combined with the global variables (forming the so called state vector), resulting in **global states**. The global transitions update the state vector; they are constructed by restricted parallel composition of the local LTSs (exclusive never claim) on the unbuffered channels. If in a global state s of the LTS two processes p and q can execute the basic statement `println("foo")`, the result is not the same since the *pc_* being increased differs. Therefore, using solely the basic commands as statements for the

LTS can result in nondeterministic statements. To guarantee deterministic statements, we need to make each *pc_* an explicit argument for the transition labels, i.e., to index the basic commands accordingly, e.g., $\text{println}(\text{"foo"})_{\text{pc of } p}$ and $\text{println}(\text{"foo"})_{\text{pc of } q}$.

When a never claim is present, its declared *sequence* is interpreted as a series of conditions on the system state, which must all become true for a path in the computation tree to be reported as undesired behavior. Theoretically, the never claim and the global LTS are interpreted as labeled Kripke structures, transformed into FSMs (see Subsec. 3.4.2) and the synchronous parallel composition is taken (cf. Def. 4.33 for details). Practically, this is implemented by interleaving transitions from the never claim and from the system in lockstep, where the never claim has capabilities to inspect the systems current state. Consequently, a blocking never claim aborts the current exploration of the $\mathcal{S}$, i.e., prunes the state space.

Therefore, asynchronous execution of the processes is simulated by nondeterministically **scheduling** the running processes; in the LTS this corresponds to multiple transitions being enabled, and **interleaving** the transitions generated by the enabled statements in all possible orders. **d_step** or **atomic** restrict the interleaving by constraining execution elsewhere in the system: A sequence within d_step{ ... } is executed **indivisibly**, i.e., without any interleaving. A sequence within atomic{ ... } is executed with exclusive privilege whenever possible.

Note. In detail, many further technical tasks need to be performed, e.g., for run, timeout, user-supplied assertions, and for the atomic and deterministic execution mode [Natarajan and Holzmann, 1997; Suprapto and Pulungan, 2011].

3.5. Improving Formal Methods by Parallelization

3.5.1. Introduction

Until about the year 2000, the sequential processing speed of CPUs increased exponentially over time. This is related to **Moore's law**, which states that the number of transistors available in CPUs and memory doubles every 18 months [Moore, 1965]. Although Moore's law still applies, "the free lunch is over" [Sutter, 2005; Tzannes et al., 2014]: Moore's law stopped benefiting the sequential processing speed (via CPU's clock speed and straight-line instruction throughput) because of physical limitations. Consequently, to make use of current advances of CPUs, programs must be **parallelized**, i.e., modularized into multiple instances, called **processes** (cf. Subsec. 3.4.3), that work concurrently to solve a problem [Lynch, 1996]. Parallelization is also the trend in formal methods: many model checkers (cf. Sec. 5.5) and SAT solvers (cf. Subsec. 3.2.3 and [Hölldobler et al., 2011]), and now also SMT solvers (cf. Subsec. 3.3.3 and [Barrett et al., 2013; URL:SMTCOMPHP] are being parallelized. This thesis also considers parallel computing for both model checking and model-based testing.

3.5.2. Foundation of Parallel Computing

This subsection shortly introduces the main aspects of parallel computing and difficulties they cause.

The architecture of the hardware determines how parallelization can be implemented:

- at the finest granularity, we have **multi-core computing**, i.e., CPUs have multiple cores. We focus on multiple instruction streams, multiple data streams [Flynn, 1972] since "the vast majority of modern parallel systems use the MIMD architecture" [URL:IntelMIMDHP; URL:ParCompHP];
- on a coarser granularity, we have **multiprocessing**, i.e., multiple CPUs connected over a bus;
- on the coarsest granularity, we have **distributed computing** [Raynal, 2013] where the processes are distributed and communicate over a network.

Many hardware architectures mix the granularities of parallelism, e.g., multi-core multiprocessing with a non-uniform memory architecture (NUMA) [Herlihy and Shavit, 2008], which is an increasing trend [Sutter, 2009].

A main aspect for the implementation of parallelism is whether **shared memory architecture** is used, i.e., memory that is simultaneously accessible by multiple processes. Shared memory programming is often performed in multi-core computing and multiprocessing, using **multi-threading**, i.e., the processes are asynchronous independent sequential programs, so-called **threads**. In contrast, systems that do not share memory (or more strictly share no common state in memory or on disk, called **shared-nothing architecture** (**SNA**)) are often used in distributed computing and communicate via message passing [El-Rewini and Abd-El-Barr, 2005; Siegel and Gopalakrishnan, 2011; Raynal, 2013] (formalized by process calculi, cf. Subsec. 3.4.3, or the Actor Model [Hewitt, 2012]). So each process owns its private resources (memory, disk, and input output devices), is independent, and is usually called **distributed component** (or **parallel instance** or **distributed node** or **node** for short). So the memory architecture determines how processes can **interact**, i.e., how they communicate and synchronize. This has a consequence on the paradigm for parallel programming and on performance.

For shared memory, own implementations or APIs such as **OpenMP** [Dagum and Enon, 1998] can be used for multi-threading. For message passing, the **message passing interface** (**MPI**) [Message Passing Interface Forum, 2012] is a popular choice, but there are also own implementations. For mixed hardware architectures, MPI and OpenMP can be combined [Rabenseifner et al., 2009]. Since accessing shared memory is a convenient programming paradigm, **distributed shared memory** (**DSM**) hides the message passing and the fact that the memory is physically separate. So DSM is convenient for programming, but the programmer looses control. DSM is implemented (in hardware or software) with a **shared memory abstraction** on top of message passing, called **distributed shared memory system** (**DSM system** for short) [Raynal, 2013]. So data is stored on the local (i.e., physically separate) memories of the distributed nodes, but addressed as one. Messages are passed by the DSM system for distribution and communication. In an object-based DSM system, objects can be stored in the DSM and used concurrently, called **concurrent objects**, e.g., using Hazelcast (see below).

To measure the consequence of the architecture on the **performance of interaction**, three main performance measures are used [Goetz et al., 2006; Lewis and Berg, 2000; McCool et al., 2012]:

- **latency**: the response time, i.e., the time between when a request is submitted (e.g., a method is called) and when the request is fulfilled (e.g., the method has fully executed and returned);

- **throughput**: the rate of progress, e.g., the amount of method calls fully executed within a given time period;
- **power consumption**: the power consumed by all processes to compute a solution.

The overall parallel performance of parallel computing is the degree to which a problem is solved in parallel (cf. Amdahl's law [Amdahl, 1967]). It is often measured in:

- **parallel speedup**, i.e., the ratio $\boldsymbol{S_P} := t_1/t_P$ between the sequential time t_1 and the parallel time t_P when $\boldsymbol{P}$ processes are used; so for **linear speedup**, $S_P = P$ (and for **super-linear speedup** $S_P > P$). The **relative speedup** takes t_1 from the parallel algorithm, the **absolute speedup** of the fastest sequential algorithm [Bader et al., 2000];
- **parallel efficiency** (sometimes called **parallel scalability** or **horizontal scalability**), which is the ratio S_P/P between the parallel speedup and the number P of processes.

More generally, speedup can be any dependent metric S_P with the explicit influencing variable P, e.g., S_P being WC time, CPU time, or number of test execution steps (cf. Subsec. 14.3.7); likewise, efficiency can be any dependent metric $E = S_P/P$ [Reussner and Firus, 2005]. These parallel performance metrics show how good a parallel program **scales**, i.e., how effectively an increase of parallelism can handle larger problems [Nicol and Willard, 1988; Hager and Wellein, 2011]. In high performance computing [Hager and Wellein, 2011; Eijkhout, 2014; Kaminsky, 2015] the two standard ways to measure scalability are:

- **strong scaling**, where the parallel runtimes (or speedups or efficiencies) are measured for increasingly large parallelization degrees P, for one fixed (and sufficiently large [Bader et al., 2000]) problem size. This approach focuses on achieving better runtimes and is suitable for parallel programs that are CPU bound, i.e., where the CPU is the bottleneck (which is usually the case in this thesis);
- **weak scaling**, where the parallel runtimes (or speedups or efficiencies) are measured for increasingly large parallelization degrees P, for a fixed workload, i.e., a fixed problem size per process, so the overall problem size increases proportionally to P. This approach focuses on achieving higher problem sizes and is suitable for parallel programs that are memory bound, i.e., where the memory is the bottleneck.

The parallel program is **scalable** (or **scales well** or **scales linearly**) when it sustains a constant efficiency. So to be strong scalable, the runtime must decrease proportionally to P; to be weak scalable, the runtime must stay fixed. Scalability becomes hard for large P since the communication cost often increases proportionally to p and eventually the workload per process becomes too small for the process to operate efficiently [Culler et al., 1998; Bader et al., 2000; El-Rewini and Abd-El-Barr, 2005; Müller-Hannemann and Schirra, 2010; Hager and Wellein, 2011; Eijkhout, 2014]. Hence one often talks about **scalability up to a certain** P and has to choose problems that are sufficiently large to outweigh constant parallel overhead, especially for strong scaling.

To increase portability and parallel performance without overburdening the software developers, several general techniques have evolved [Herlihy and Shavit, 2008; McCool et al., 2012]:

- flexible and high-level synchronization constructs, e.g., futures and promises [Baker Jr and Hewitt, 1977; Herlihy and Shavit, 2008; URL:FuturesPromisesScalaHP;

Mohr et al., 1991]: a **future** F is a read-only proxy for a result not yet available, which is computed potentially in parallel (synchronously or asynchronously) via a corresponding **promise**, a single assignment container that sets the value of F, changing F's state from **promise pending** to **promise kept** or **promise broken**. Futures and promises are concise language constructs for parallelism, and are flexible because they allow efficient blocking and non-blocking communication, can be used for multi-core computing, multiprocessing, and distributed computing, and make little restrictions on parallel schedulings (see Sec. 3.6);

- many variants and performance optimizations exist for parallel schedulings: In this thesis, the focus is on **task parallelism** (aka **function parallelism** or **control parallelism**), which schedules tasks (e.g., processes, promises) onto parallel instances. For instance, when a thread pool is used, tasks are distributed onto a limited pool of long-lived threads to reduce overhead [Herlihy and Shavit, 2008; Lee, 2006]. Another important optimization is **load balancing**, i.e., performing **work distribution** such that ready tasks are efficiently assigned to idle instances [Herlihy and Shavit, 2008; Tzannes et al., 2014, 2010];
- lock-free and wait-free data structures (cf. Subsec. 6.7.3 and [Herlihy and Shavit, 2008]) lead to better work distribution and less **contention**, which is the amount that processes compete for a resource (e.g., try to acquire a lock at the same time). Contention is the main impediment to parallel scalability and predictability [Sutter, 2009; Al-Bahra, 2013], so it is important to avoid it.

For parallel programs with a large amount of global communication, shared memory programming has much higher performance of interaction than message passing programming [Chorley, 2007]. Contrarily, distributed computing in a shared-nothing architecture has low coupling where all nodes only coordinate via asynchronous message passing, leading to less complexity of the communication, less nondeterminism [Lee, 2006] and better reliability and predictability.

Multi-threaded Shared Memory Programming

Two main aspects of interaction in multi-threaded shared memory programming are its use for synchronization and optimizations.

Non-blocking synchronization often yields higher performance than blocking synchronization, and is often based on some primitive synchronization operation, called **universal synchronization primitive** if it is sufficiently expressive (cf. universal construction in Subsec. 6.7.3 and [Herlihy, 1991]). This thesis uses **Compare-And-Swap** (**CAS**) operations, which atomically updates a memory location only conditionally by comparing values [Herlihy, 1990]. CAS is currently the most established universal synchronization primitive (e.g., via Intel's CMPXCHG [Millind Mittal and Eval Waldman, 1999]).

Note. This thesis uses CAS for synchronization because it is the prevalent universal synchronization primitive, even though it has to deal with the ABA problem, i.e., that a process assumes no change when the synchronization primitive compares A to A, when in fact other processes have changed the value A to the value B and back to A [Michael, 2004]. Transactional memory [Herlihy and Moss, 1993] will likely replace CASs eventually, e.g., the hardware transactional memory primitives Load-Linked, Store-Conditional,

Validate (LL/SC/VL) [Michael, 2004] or Intel's Transactional Synchronization Extensions (TSX) [Intel, 2012; Maciej Swiech, Kyle C. Hale, 2012]), or software transactional memory as in the GNU Compiler Collection (gcc) since version 4.7 [Boehm et al., 2012], possibly via easy conversions such as Hardware Lock Elision (HLE) [Rajwar et al., 2013; Maciej Swiech, Kyle C. Hale, 2012].

Since accessing shared memory is slow, optimizations are installed:

- CPUs and compilers cause **out-of-order execution** (often called **reordering optimizations** for compilers), i.e., the instructions are not execute in the order given by the program code. Instead, an instruction may be executed lazily when its input data is available and the result of the instruction is required, hereby increasing throughput. Reordering is performed according to the specified memory consistency model [Berg, 2014; Adve and Gharachorloo, 1996] (causing further combinatorial and state space explosion during verification);
- the hardware architecture uses **caching**, i.e., it additionally stores data locally, to achieve lower latency. Usually, memory is stacked over many levels, resulting in steep memory hierarchies.

Out-of-order execution and reordering optimizations of multi-threaded programs can break **memory consistency**, e.g., sequential consistency (or some other consistency, depending on the memory model), i.e., exhibit linear time behavioral properties (cf. Chapter 4) differing from any interleaving of the threads' sequential programs [Herlihy and Shavit, 2008]. To avoid this, we need to add **memory barriers** (also called **memory fences**) to inhibit reorderings that break memory consistency; memory barriers are instructions that guarantee that certain memory operations are performed before the barrier, and certain others after the barrier.

Note. An exemplary memory barrier is the **full fence**, which forbids any memory operation to cross it. An **acquire memory barrier** only forbids any following memory operation to cross it (also called **read-acquire** since it is used for read operations); conversely, a **release memory barrier** forbids any preceding memory operation to cross it (also called **write-release** since it is used for write operations). Memory barriers are strongly hardware or compiler dependent; for instance, `volatile` in C++ does not guarantee sequential consistency, whereas in Java, it does by preventing certain compiler as well as hardware reorderings.

Caching in shared memory architectures calls for a **cache coherency protocol** to propagate a change of a value in one cache throughout the system, so that the value is coherently updated everywhere. A **cache line** is the finest-grained memory block in cache at which coherency is maintained. Its size, often 64 byte, is called **coherency granularity**.

Caching in multi-threaded programs can lead to contention, with a major cause being **false sharing**, which occurs when the cache coherency protocol forces a core to reload a cache line albeit only irrelevant data in the cache line has been altered by another core. So the effect on reloads is the same as for truly shared data [Al-Bahra, 2013]. To avoid false sharing, we can use **padding**, i.e., we add dummy variables so that each cache line only comprises logically related data.

Distributed Computing with Message Passing

Two main aspects of distributed computing is its architecture and the implementation of message passing.

Distributed architectures follow one of the two fundamental communication models [Subramanian and Goodman, 2005; Vogel et al., 2011]:

- in a **master/slave network** (aka **client/server network** or **centralized network**), a dedicated node, called **master** has the authority over the network and hence initiates communication with all other nodes, called **slaves**. Often the master controls behavior of the slaves and slaves do not communicate with each other;
- in a **peer-to-peer network** (aka **decentralized network**), all nodes have equal authority, functionality and behavior, and can potentially communicate with any other instance. Each node is self-sustaining and coordinates with other via asynchronous message passing, to supply as well as consume messages and tasks. Thus nodes need no initial knowledge of the distributed environment other than the communication address.

Example. A prominent programming model that uses a master/slave network is **MapReduce** [Dean and Ghemawat, 2004], with Hadoop [URL:HADOOP; White, 2012] being an established implementation. A MapReduce Master (called NameNode in Hadoop) distributes each task as a job to a slave, via a map function, and a reduce function collects and combines the results.

Master/slave networks have some disadvantages:

- complexity of efficient task distribution (cf. work distribution above and lazy techniques for parallelization in Subsec. 3.6.2);
- a single point of failure, i.e., they are not fault-tolerant;
- a considerable overhead [Duarte et al., 2006] by the communication for coordination since the server has to send all tasks to clients and receive all their results. This can lead to contention for the network resource, i.e., network congestion.

These problems can be avoided by using a peer-to-peer network: It is more flexible and has no single point of failure or contention. But under heavy communication load, they have decreased communication performance. Hence communication must be implemented efficiently.

Many **implementations of message passing** exist, directly on some standard protocol, or more elaborate implementations with higher level interfaces and more features.

The Internet Protocol Version 4 (IPv4) offers **broadcasts** via the User Datagram Protocol (UDP), which is a lightweight, real-time, one-to-all communication [Mogul, 1984a,b]: it efficiently sends a datagram via a single Internet Protocol (IP) transmission (i.e., over each link of the network only once) to all receivers (i.e., network-attached hosts). Since the receivers are not known to the sender and no handshake is made, broadcasts via UDP are unreliable, i.e., datagrams may be lost or delivered out of order.

A **multicast** via UDP is a lightweight, scalable, real-time, one-to-many or many-to-many communication [Deering, 1989; Deering and Hinden, 1998]: like broadcasts, it efficiently sends a datagram via a single IP transmission to all receivers that have joined the corresponding multicast group. Like broadcasts, multicasts via UDP are unreliable. The Internet Protocol Version 6 (IPv6) no longer offers IPv4's (layer 3) broadcasts; but

multicasts subsume broadcasting and are more selective, scalable and efficient [Graziani, 2012]. Therefore, multicasts are future-proof for the Internet [Deering, 1989; Deering and Hinden, 1998; Graziani, 2012], for cloud computing [Rothenberg, 2010; URL:GOGRID; URL:RACKSPACE; URL:DIMENSIONDATA] and for data centers [McBride, 2013].

Hazelcast [Johns, 2013; URL:Hazel] is an open source object-based DSM system enabling shared memory programming for distributed systems. It offers collections (e.g., lists, sets, and maps), observers, synchronization primitives, concurrency utilities (AtomicNumber and IdGenerator), and a scheduler, called **distributed executor service**, to create and run tasks anywhere on the cluster (see below). It is also called an in-memory data grid to reflect its database capabilities, especially persistence and transactions. It is a peer-to-peer embedded, in-memory database that is based on distributed hash tables and is self-discovering, self-clustering and elastic: contributing nodes automatically and dynamically form a cluster to almost evenly distribute all data across all nodes for high parallel scalability [Zhang et al., 2013]. For fault-tolerance, data is stored redundantly, so there is no single point of failure. The cost for these powerful features is high runtime and communication cost for setup and cleanup of the Hazelcast cluster, especially for discovery and for repartitioning and new backups when an instance leaves the Hazelcast cluster (cf. Appendix B.1.4). To avoid repetitive setup and cleanup, Hazelcast's distributed executor service and elasticity can be used.

A major application of distribution on the Internet is via **web services** (**WSs**), which are software systems for interoperable machine-to-machine interaction over a network [URL:WS] (cf. Subsec. 14.3.2). There are many architectures (covered by service-oriented architecture) and protocols (like SOAP and REST) that can be used for WSs. For machines to automatically determine what operations are available on some server, the interfaces of its web services are formally specified, e.g., in the Web Services Description Language (WSDL) [URL:WSDL], and their behavior, e.g., in the WSDL-S [URL:WSDLS] or WS-BPEL [URL:WSBPEL]. Since only the specifications of the web services are available, but not their implementation, web services are black-box systems. Thus the machines depending on black-box systems that have formal specifications. Therefore, much work in formal methods have been dedicated to WSs [Bozkurt et al., 2010; Wang, 2013].

3.5.3. Roadmap

The last subsection has shown that parallel programming is difficult. This thesis considers multiple aspects of parallel computing and ways to cope with these difficulties:

- Subsec. 3.4.3 shortly introduced models for parallelism via composition, helping to cope with the complexity by modularization;
- Chapter 5 describes the parallel computing capabilities of several model checkers, to improve their feasibility. Model checkers also help prove correctness of parallel programs;
- Chapter 6 covers the parallelization of our model checking algorithm DFS_{FIFO} to further improve the feasibility of livelock detection, which is an important property in parallel programs. Since DFS_{FIFO} requires a large amount of global communication, multi-threaded shared memory programming is used;
- Chapter 7 covers the application of software BMC on a distributed system of low

energy wireless sensor networks. It copes with the complexity by abstraction and selection of exemplary parallel scenarios;
- Chapter 11, Chapter 13, and Chapter 14 cover the parallelization of our MBT algorithm LazyOTF for integrating on-the-fly and offline MBT. As for model checking, MBT and parallelization benefit from each other. Since the parallelization of LazyOTF requires little communication and decouples processes via own private resources for test execution, distributed computing with message passing is used.

3.6. Improving Formal Methods by Lazy Techniques

3.6.1. Introduction

Often problems are solved by splitting them up into smaller problems (for instance modularization [Parnas, 1972; Hughes, 1989; Abelson et al., 1996] by divide and conquer [Bentley, 1980; Smith, 1985; McCool et al., 2012], or for parallelization [Milner et al., 1992; Baker Jr and Hewitt, 1977; Tzannes et al., 2014]). This yields multiple tasks to be processed, e.g., the execution of multiple function calls or the evaluation of multiple expressions.

The simplest approach is to execute these tasks as soon as they arise. This scheduling of tasks is called **eager** or **greedy** and is the only possible scheduling iff all currently unfinished tasks cannot advance the moment a new task t arises, until the result of t is available, i.e., until t terminates. But often this is not required, so a task can be executed at a later point in time than it arises, i.e., it can be executed lazily. Hence schedulings that differ from eager scheduling are called **lazy**. The most extreme lazy technique postpones the execution of each task until no longer possible, i.e., until the result is demanded by some other task.

Example. The most prominent lazy techniques are from the field of programming, e.g., lazy initialization and lazy evaluation, but other areas of software development can also benefit from lazy techniques, for instance modeling and testing.

Lazy initialization of a variable v initializes v lazily only when it is read in an execution. So the initialization expression is not evaluated when it is bound to v, but only the moment v is read by some later expression. If the initialization is costly but v is read only in few execution paths, lazy initialization of v can strongly spare initialization tasks and memory on average. But lazy initialization is complex and thus error-prone, especially in concurrent systems [Bloch, 2008]. Therefore stronger checks are now often offered by static typing [Dietl et al., 2011; Klepinin and Melentyev, 2011; URL:Lombok; URL:LazyInitCSharpHP].

More generally, functional programming languages often have **lazy evaluation** [Watt and Findlay, 2004; Hughes, 1989; Asperti and Guerrini, 1998], which also covers expressions for actual function parameters [URL:ScalaByNameHP] besides expressions for assignments. Lazy evaluation can avoid some evaluations altogether and avoid repeating an evaluation if expressions are shared [Asperti and Guerrini, 1998]. Besides decreasing time and space requirements, lazy evaluation offers expressivity, e.g., for modularity and to efficiently operate on (potentially) **infinite data structures** using recursive types [Bird and Wadler, 1988; Watt and Findlay, 2004; Hughes, 1989; Abelson et al., 1996].

For agile software development (cf. Sec. 14.2), modeling should also be agile, i.e., avoid a big design up front, but evolve lazily, e.g., spread over sprints. Many terms are used for this: **lazy specification** [Simons, 2007; Simons et al., 2008], **document late** and **iteration modeling** in agile modeling [Ambler, 2002]. Refinement of specifications (e.g., via the refines relation, cf. Sec. 14.2) is one example of lazy specification. Since tests are often used as specification in agile software development, test-driven development is a prominent example for lazy specification and lazy testing, since tests are only developed in the same sprint as the code for each user story. Another example of lazy specification and lazy testing is **lazy systematic unit testing** [Simons, 2007; Simons et al., 2008], where the specification evolves lazily by inferring it on-the-fly by dynamic analysis with interaction from the test engineer: the state space is explored systematically by automatically generating test drivers up to a given depth, test oracles are generated by user interaction as well as automatic, heuristic rules over previous oracles. So for software engineering processes, various tasks are scheduled, and the examples above are lazy schedulings.

3.6.2. Current Lazy Techniques for Formal Methods

Lazy techniques are often used for formal methods, for instance in SAT solvers, SMT solvers, model checking, model-based testing and in the parallel algorithms they apply.

One lazy technique that many SAT solvers use is two literal watching: literal assignments are updated lazily for checking unit/conflicting clauses, improving performance [Mahajan et al., 2004]. In its core, SAT solvers often use further lazy techniques, e.g., for the data they generate [Alsinet et al., 2004; Ohrimenko et al., 2007].

SMT solvers also use eager or lazy techniques, depending on whether theory information is processed eagerly or lazily (cf. Subsec. 3.3.3 or [Sebastiani, 2007; Biere et al., 2009]): The tasks for SMT solving are partitioned into processing specific theories and processing the propositional part, i.e., SAT solving. **Eager SMT** employs **eager encoding** for some theory T, i.e., all the expressions in T are translated from the beginning into SAT problems before SAT solving starts; then the SAT solver is invoked once with no further interaction with a decision procedure of theory T. **Lazy SMT** employs **lazy encoding** for some theory T, i.e., each expression t in T is processed lazily by a **T-solver** during SAT-solving, i.e., dynamically whenever the SAT solver demands the evaluation of t. If different theories are present and handled differently, an interesting mix of lazy and eager techniques may occur. For instance in [Falke et al., 2013c], the theory of uninterpreted functions and lambdas are eagerly encoded into either theory of quantifiers or arrays and bitvectors. The used SMT solvers STP and Booleactor in turn use an eager encoding of the theory of bitvectors into SAT using bitblasting. Arrays are mostly solved lazily [Brummayer and Biere, 2009].

Many formal methods algorithms in this thesis use a transition system $\mathcal{S}$ as specification and perform checks in certain states or transitions of $\mathcal{S}$. Then tasks and their (sequential or parallel) scheduling can be divided as defined in Def. 3.33.

Definition 3.33. For algorithms performing checks in certain states or transitions of a transition system, their tasks can be partitioned into:

- **transition tasks**: taking a transition during graph traversal of the specification;
- **check tasks**: performing some check for a state in the graph of the specification.

Then schedulings of tasks can be partitioned into:

- **offline techniques**, which firstly perform all transition tasks and thereafter all check tasks;
- **on-the-fly techniques** (**OTF** for short), which perform the transition tasks lazily with respect to the check tasks, i.e., both kinds of tasks are intertwined.

Note. Def. 3.33 shows that at one point in time, several new tasks can arise (e.g., one check task and many transition tasks). Furthermore, the order that constitutes "greedy" depends on the design of the algorithm. Offline techniques are the default design, so executing the transition tasks later is called "lazy".

Similar to Def. 3.33, the tasks of other algorithms can also be partitioned into two classes, e.g., for SMT solvers as described above. Then offline techniques schedule the tasks of these two classes separately, while on-the-fly techniques pick the tasks on-the-fly, i.e., intertwine the tasks of both classes.

For model checking, offline techniques separate tasks into state space construction and the actual checks on the state space. So for on-the-fly model checking, also called lazy model checking, the state space is only constructed on demand, i.e., dynamically during the checks. Both offline and on-the-fly schedulings are applied broadly (cf. Chapter 5). Some eager check tasks (e.g., via the synchronous product of the system specification and property to be checked, cf. Note 4.34) help to prune the parts of the state space irrelevant for the property to be checked (cf. Subsec. 3.4.3 and Subsec. 5.3.2). This is sometimes called **level 0 OTF** [Barnat et al., 2009]. If more check tasks are scheduled earlier, model checking can achieve **early termination**: the model checking algorithm

- may terminate before the state space is fully constructed, for **level 1 OTF** [Barnat et al., 2009];
- terminates as soon as an error is reached during traversal (i.e., no further state is traversed), for **level 2 OTF** [Barnat et al., 2009].

This categorization was in relation to the state space. If the property to be checked is described as transition system $\mathcal{P}$, model checking that performs traversal tasks on $\mathcal{P}$ lazily is sometimes called **truly OTF** [Hammer et al., 2005].

The degree of laziness for on-the-fly techniques (especially for model checking) is often measured by its time and space requirements in case a fault is found, since these requirements are strongly influenced by the amount of transition tasks that have been performed; this performance measure is often called **on-the-flyness** (cf. Subsec. 6.8.6). On-the-flyness is quite relevant in practice: Checking a property P is reasonable only if there is the possibility that P does not hold, in which case strong on-the-flyness detects that P does not hold significantly faster than weaker on-the-flyness [Barnat, 2010]. So often on-the-flyness for level 0 OTF is much smaller than for level 1 OTF, and for level 1 OTF it is much smaller than for level 2 OTF.

Bounded model checking with a bound check (cf. Subsec. 5.2.3) also applies lazy scheduling: a completeness threshold is not computed a priori for a Kripke structure and property; instead, bound checks are performed lazily.

For model-based testing, checks are usually performed dynamically during runtime; then offline techniques perform transition tasks statically and check tasks dynamically. Separation of these phases can cause high time and space requirements (e.g., high test

case complexity, cf. Def. 10.4). The early algorithms for model checking and model-based testing use offline techniques; newer algorithms often use on-the-fly techniques.

If multiple phases are involved, the situation becomes even more complex. Examples are the mix of lazy and eager SMT solving described above, and the MBT tool TGV (cf. Subsec. 10.3.1), which uses offline techniques with respect to model-based testing, but on-the-fly techniques with respect to model checking.

Many lazy techniques exist for parallelization, which can also be used by formal methods if they are parallelized (cf. Sec. 3.5). Exemplary lazy techniques are:

- using a future F when its state is promise pending, e.g., as an actual parameter; this corresponds to lazy evaluation or lazy computation. F's promise can be computed eagerly or lazily, depending on the available resources and use of F;
- for parallel schedulings: lazy task creation, which inlines a future F by default and creates a costly task for F only lazily when processing resources become available [Mohr et al., 1991]. Similarly, lazy threads only expose an entry point for work that can be performed remotely. When the work ends up being performed locally, it is simply inlined into the local thread as a sequential call, i.e., remote work is generated only lazily [Goldstein et al., 1996]. Modern work distribution is supported by implicit synchronization, implicit scheduling, and task-parallel languages, which offer high-level constructs for task parallelism via nested parallelism, i.e., more parallelism created from an already parallel context. This necessitates dynamic scheduling for work distribution, which is implemented mainly by **work stealing** [Herlihy and Shavit, 2008; Faxén, 2008; Tzannes et al., 2014, 2010; McCool et al., 2012; van Dijk et al., 2012; Laarman and Faragó, 2013]: each instance has a local queue to add new tasks and work off tasks; an idle instance lazily tries to steal a task from the queue of another instance, usually chosen randomly (cf. Subsec. 6.7.3 and [Tzannes et al., 2010]). Alternatives to work stealing are: Firstly, **synchronous random polling** [Sanders, 1997; Laarman et al., 2010; Laarman, 2014], where a thread that becomes idle lazily polls another random thread for work, which splits its problem into two sub-problems. This results in low overhead tree shaped computations. Secondly, **lazy scheduling** [Tzannes et al., 2014] firstly exposes all available parallelism and only lazily restricts it to not exposing too short and too many tasks for the current load conditions and target platform. This minimizes scheduling overheads without manual tuning. Similar lazy techniques for dynamic load-balancing, to adapt dynamically to load conditions, also exist for work stealing [Tzannes et al., 2010];
- lock-free and wait-free data structures (e.g., for work distribution) also profit from lazy techniques, for instance lazy removal of entries from a list (cf. Subsec. 6.7.3 and [Herlihy and Shavit, 2008; Heller et al., 2005]).

3.6.3. Advantages and Disadvantages of Lazy Techniques

Some disadvantages of lazy techniques, which are also present for formal methods, are:

- the problem needs to be modularized into fine-grained tasks;
- lazy schedulings of these tasks is more complex and thus error-prone. Complexity is caused by having to decide how long a task should be postponed. Furthermore, phases that are separate in eager scheduling often get intertwined in lazy

schedulings;
- laziness impedes debugging because tasks that were originally performed statically are postponed to dynamic phases (i.e., the fail fast design concept is often no longer followed [Gray, 1986; Bloch, 2006]).

Note. When applying formal methods, laziness rarely impedes debugging because the tasks that move from static to dynamic phases rarely fail. One exception is lazy processing of specifications (e.g., for on-the-fly MBT) since specifying is error-prone. Consequently, test engineers should perform on-the-fly MBT as soon as they formulated the specification. If no SUT is yet available, it can be simulated (cf. Subsec. 10.3.3).

Some of the common advantages of lazy techniques, which are also applicable to formal methods, are:

1. avoiding performing tasks that are not required later on, leading to better performance in space and time; or even rendering some method feasible, e.g., processing of infinite data structures;
2. avoid performing tasks that have already been performed, by reusing the results of old tasks;
3. avoid space and time required for decoupling the phases;
4. the availability of new, dynamic information that just arose (e.g., some nondeterministic choice). The information can help avoid tasks (see item 1), or prioritize tasks, or increase comprehension;
5. comprehension is also improved by higher expressiveness and simplicity;
6. ease of use by higher expressiveness, better comprehension and higher feasibility;
7. reduce parallel contention and improve work distribution and parallel performance.

Further advantages that laziness yields for our algorithms DFS_{FIFO} and LazyOTF are mentioned in the following subsection and are investigated in later chapters.

3.6.4. Roadmap

In this thesis, new algorithms with lazy techniques are designed for model checking and model-based testing: while on-the-fly model checking and on-the-fly model-based testing intertwine transition tasks and check tasks, they do so rigidly (cf. Sec. 6.3 and Subsec. 10.2.5). Our new algorithms will intertwine the tasks more elaborately to achieve many advantages, those from the previous subsection and more:

In Chapter 6, the algorithm DFS_{FIFO} checks livelocks on-the-fly in one pass. For this, it schedules the transition tasks that make progress maximally lazily, i.e., only after all transition tasks without progress, as well as their check tasks, have been scheduled. DFS_{FIFO} is a level 2 truly OTF algorithm and shows two further advantages of lazy schedulings: The order of tasks entails **correctness** of DFS_{FIFO}, i.e., a sound and complete livelock detection, while other schedulings do not. Furthermore, DFS_{FIFO} enables strong optimizations, especially partial order reduction, where the lazy scheduling allows weaker provisos, and parallelization, where low contention leads to almost linear parallel speedup.

In Chapter 11, lazy on-the-fly MBT is a level 1 OTF algorithm that leads to feasible testing of large and nondeterministic systems. For this, it offers a framework to schedule some transition tasks eagerly and some lazily. This mix yields another advantage of lazy

schedulings: The transitions that are eagerly scheduled need not be executed on the SUT, enabling **backtracking**, i.e., undoing of choices, such that better choices can be executed on the SUT. To decide which choices are better, so-called guidance heuristics are used, which exploit dynamic information, enabled by lazy scheduling of transition tasks.

Chapter 12 introduces heuristics for scheduling of tasks, to decide which transition tasks to schedule lazily. Thereafter, guidance heuristics are investigated, which are on-the-fly heuristics that process static and dynamic information.

3.7. Improving Formal Methods by Abstraction

3.7.1. Introduction

Abstractions are methods that discard or hide details (called Verkürzungsmerkmal in general model theory [Stachowiak, 1973]) to get results that are simpler, uniform, more formal. These results are called **models**, or also abstractions. So models simplify a system for a certain goal [Prenninger and Pretschner, 2005].

The general goals of the simpler, uniform, and more formal structures are: to understand, communicate, validate and manage facts, to analyze the system, or to generate further artifacts, e.g., tests or other checks for the system, or to generate the system itself. Without abstractions, which reduce complexity, these goals would not be feasible at all, e.g., due to state space explosion [Fraser et al., 2009], or at least more costly, e.g., with a lower degree of automation. For formal methods, models are used for uniform and automated checks on a system to analyze it – mainly its correctness using system specifications as models (cf. Subsec. 3.4.3). In model checking, models are used to describe the functional behavior of the SUT and to specify the desired abstract behavior of the SUT. In model based testing, models are used to describe the desired functional behavior of the SUT, and to specify what kind of tests should be performed. Furthermore, to formalize conformance testing, a mental model is used to directly abstract the SUT.

We differentiate abstractions by two aspects: what details are omitted, and how:

The first aspect differentiates whether the omitted details are functional or non-functional. **Functional abstraction** leaves out details of the behavior, e.g., special cases or the order of events or by comprising implementation details. **Non-functional abstraction** leaves out other details, mainly on data, e.g., implementation details and bounds for some data type, or data no longer required due to functional abstraction. Other non-functional abstractions are often related to performance or resource consumption.

The second aspect differentiates whether the omitted details are discarded or hidden: **Lossy abstractions** (also called abstractions with **actually missing information** [Prenninger and Pretschner, 2005]) are abstractions where the details are discarded, e.g., details that cannot be processed (e.g., too many possibilities), cannot be controlled (e.g., behavior of lower layers), or cannot be determined yet (e.g., implementation details of future work). Hence the missing information cannot be inserted automatically to reconstruct the original situation. The strongest benefits are reduced complexity and hence less resources for processing the models. **Lossless abstractions** are faithful,

so the details are still available somewhere or somehow, e.g., moved or compressed, so that the original system can be reconstructed [Prenninger and Pretschner, 2005]. This mainly increases understandability, but can also improve performance. The reconstruction, i.e., the opposite of abstraction, is called **refinement** [van der Bijl et al., 2005], or **contraction** in this context.

If some check does not take all relevant details into account, e.g., some relevant detail was lossy abstracted, the check can become unsound or incomplete.

Example. Compilers and macro mechanisms such as keywords in keyword driven testing [Brandes et al., 2015] are lossless abstractions, usually functional.

Abstraction from concrete entries in an underlying database is lossy data abstraction.

Abstracting from floating points to real numbers is lossy data abstraction on implementation details and bounds, and often results in unsound or incomplete checks.

UML models that describe the architecture are usually lossy functional and non-functional abstractions.

Multiple levels of abstraction are often helpful [Faragó, 2010; Ulbrich, 2014; Abrial, 2010]. A popular example are the ISO-OSI levels: an abstraction hierarchy where higher levels are lossless abstractions of lower levels, replacing some functional behavior of communication into one more abstract element.

This thesis contains many more examples, enumerated in the roadmap below.

3.7.2. Roadmap

In this thesis, several lossy abstractions are performed:

- in this chapter, systems are described formally by mathematical structures, loosing (mainly non-functional) details from real systems;
- the interface abstractions and testing hypothesis (cf. Sec. 8.1) formalize the functional and non-functional gap (i.e., the lost details) that always exists between a real system and a formal description. For instance, abstractions from the precise timing of certain events is a non-functional abstraction that is lossy and often results in unsound or incomplete checks on performance properties;
- this thesis frequently uses various kinds of nondeterminism (cf. Subsec. 8.2.5); **nondeterminism** abstracts by comprising multiple possibilistic choices – functional or non-functional. These choices are made at **nondeterministic choice points**, the choices are called **nondeterministic choices**. A sequence that determines the choices for all encountered choice points is called **nondeterministic resolution** [Bienmüller et al., 2000; Lawrence, 2002; Bowman and Gómez, 2006]. Often nondeterminism covers more variants than occur in reality and it is not possible to reconstruct exactly the real variants, i.e., the abstraction is **truly underspecified** and hence lossy. Nondeterminism in the specification of the SUT is uncontrollable for the test engineer, whereas nondeterminism in the specification of the SUT's environment is controllable (cf. Subsec. 8.2.5), which is utilized in the next item;
- nondeterminism is often applied to abstract from the full system: by only considering a module of the full system, we modularize and leave the remaining modules (e.g., the user, network or library) open, as **environment**. Thus we only have to specify an open systems. But if a formal method should process such a systems rigorously and automatically, no control and input to the formal method must

be supplied via interaction with the environment. Thus the specification for the formal method should describe a closed system, which includes some environment to contain all information required for automatic processing. The specification can transform the **open** system into a closed system by
  - either taking the **maximal environment**, i.e., nondeterministically allowing all possibilities (cf. Subsec. 7.1, demonic completion in Def. 8.24, [Godefroid et al., 2005; Cadar et al., 2008a]);
  - or taking only expected environments, i.e., allowing only certain possibility that expected environments exhibit, by generating a more restrictive environment specification (cf. angelic completion in Def. 8.23, [Parizek et al., 2009]), e.g., using a tool like Java Pathfinder [URL:JPF]). This aims to make the environment specification sufficiently restrictive for the formal method to be more feasible, yet still general enough to cover relevant cases.

  So for environment abstractions, we reduce details by making certain assumptions on the environment that the software is embedded into, e.g., that it follows the assumed interfaces. Another example of environment abstraction is our abstract behavior model (cf. Subsec. 7.2.3), which uses nondeterminism to consider multiple behaviors within one scenario, and to abstract from details such as probabilistic values and hardware;
- underspecification and nondeterminism (via our *refines* relation, cf. Sec. 9.3) to postpone implementation decisions for iterative software development (cf. Sec. 14.2);
- the refinement hierarchy via our *refines* relation (cf. Sec. 9.3 and Sec. 14.2), the V model (cf. Sec. 2.4, [Borgida et al., 1982]), and the abstractions from the SUT to the model to the specification to the test cases (cf. Chapter 8) are examples for multiple levels of abstraction;
- in Chapter 5, several reductions with lossy abstractions are introduced: mainly partial order reduction, which groups together interleavings that have the same property. Furthermore, symmetry reduction, program slicing, abstract interpretation and statement merging are introduced, and finally bitstate hashing and hash compaction, which are often incomplete or unsound;
- in Chapter 12, weights as lossy abstraction from meaningfulness.

In this thesis, also lossless abstractions are used:

- in Sec. 5.4, several state space compression techniques for model checking are introduced: symbolic techniques, tree compression and collapse compression;
- for test adapters (cf. Subsec. 8.7.2), an abstract test step can comprise multiple concrete test steps;
- many programming APIs and frameworks we use offer simplifications, e.g., Hazelcast to abstract from distributed memory and from message passing, and task parallel languages to abstract from dynamic scheduling and creating parallelism dynamically (cf. Subsec. 3.5.2).

3.8. Conclusion

3.8.1. Summary

This chapter introduced formal methods in a general way, applicable to model checking and model-based testing, covering propositional logic, first order logic, automata theory, parallelization for formal methods, and abstractions for formal methods (cf. positioning in Fig. 15.1 on page 377.

3.8.2. Contributions

Sec. 3.6 derived a model to describe lazy techniques in formal methods, considered their advantages and disadvantages, as well as various instances.

Part II.

Model Checking

4. Temporal Logics

4.1. Introduction

This chapter discusses temporal properties and temporal logics. They can be used as basis for model checking, covered in this part of this thesis, and several related formal methods, e.g., model-based testing, covered in Part III.

Having such a broad view, this thesis deviates in some aspects from standard literature [Clarke et al., 1999b; Baier and Katoen, 2008]: Since software is frequently and effectively specified as Kripke structures, we will use them as semantic structure as much as possible, e.g., also as basis for behavioral properties. Having the same structures improves interchangeability and reuse amongst techniques, as well as their comparison and understanding by verification engineers.

We also restrict their kind as little as possible, e.g., by allowing infinitely many states wherever possible, as well as final states, i.e., end states, and adapting the semantics of temporal logics accordingly. This improves how termination as well as non-termination can be handled. Behavioral properties, temporal logics and their relations are also introduced in a more general, formal and consistent way. Therewith, we can better compare various classes of behavioral properties, temporal logics and their expressiveness.

Roadmap. To be able to talk about behavior, Sec. 4.2 defines the semantics of behavioral properties over Kripke structures (cf. Subection 3.4.1) and classifies them. Sec. 4.3 shows how these classes of properties can be described formally by defining several temporal logics. Sec. 4.4 further investigates the relationships between behavioral properties and various temporal logics.

4.2. Behavioral Properties

Subsec. 3.4.1 introduced properties of single states in isolation by extending a transition system (S, T) to a Kripke structure (S, T, Σ, I) (cf. Def. 3.18).

Behavioral properties usually do not only consider a single state s in isolation, but the whole Kripke structure (e.g., the complete path that led to s). They are sometimes also called temporal properties. In practice, even the simplest ones that do only consider s in isolation, additionally investigate whether s is reachable from *init*, and are hence called **reachability properties**.

For properties of states in isolation, we did not consider the states themselves, but their interpretation I over Σ. We take the same model theoretical approach for behavioral properties, as defined in Def. 4.2. For this, Def. 4.1 firstly abstracts from a state s to $I(\cdot, s)$, then lifts the abstraction to paths and finally to $\mathbb{S}_{Kripke}$ (resulting in a kind of complete trace equivalence [Bonchi et al., 2012] in relation to $I(\cdot, s)$).

Definition 4.1. Let both $\mathcal{S}_1 = (S_1, T_1, \Sigma, I_1)$ and $\mathcal{S}_2 = (S_2, T_2, \Sigma, I_2)$ be in $\mathbb{S}_{Kripke}$, $S = S_1 \cup S_2$ (wlog, $S_1 \cap S_2 = \varnothing$).

Then $\boldsymbol{\approx_\Sigma}$ is the equivalence relation defined

- over S with $\forall s, s' \in S : s \approx_\Sigma s' :\Leftrightarrow I(\cdot, s) = I(\cdot, s')$, with I being either I_1 or I_2, i.e.,$\forall s \in S : I(\cdot, s) = I_i(\cdot, s)$ for $s \in S_i$;
- over $paths(\mathcal{S}_1) \cup paths(\mathcal{S}_2)$ with $\forall (s_i)_i, (s'_i)_i \in paths(\mathcal{S}_1) \cup paths(\mathcal{S}_2)$: $(s_i)_i \approx_\Sigma (s'_i)_i :\Leftrightarrow |(s_i)_i| = |(s'_i)_i|$ and $\forall j \in [0, \ldots, 1 + |(s_i)_i|) : s_j \approx_\Sigma s'_j$;
- over $\mathbb{S}_{Kripke}$ with $\mathcal{S}_1 \approx_\Sigma \mathcal{S}_2 :\Leftrightarrow paths_{max}(\mathcal{S}_1)/\approx_\Sigma = paths_{max}(\mathcal{S}_2)/\approx_\Sigma$.

Definition 4.2. Let the signature Σ be given. Then

- a **behavioral property** P is a subset of $\mathbb{S}_{Kripke}/\approx_\Sigma$;
- $[\mathcal{S}] \in \mathbb{S}_{Kripke}/\approx_\Sigma$ **satisfies** $P :\Leftrightarrow [\mathcal{S}] \in P$ (written $[\mathcal{S}] \models P$);
- $\mathcal{S} \in \mathbb{S}_{Kripke}$ **satisfies** P $:\Leftrightarrow [\mathcal{S}] \models P$ (written $\mathcal{S} \models P$)
  $\Leftrightarrow$: $\mathcal{S}$ is a **model** for P.

Roadmap. This section covers the classification of behavioral properties as depicted in Fig. 4.1: Subsec. 4.2.1 defines branching time behavioral properties, Subsec. 4.2.2 linear time behavioral properties. The focus is on linear time properties, which are the most suitable properties for software verification (cf. Subsec. 5.2.5). Therefore Fig. 4.1 already depicts further details: the corresponding temporal logics LTL (introduced in Subsec. 4.3.2) and Büchi automata (introduced in Subsec. 4.3.2).

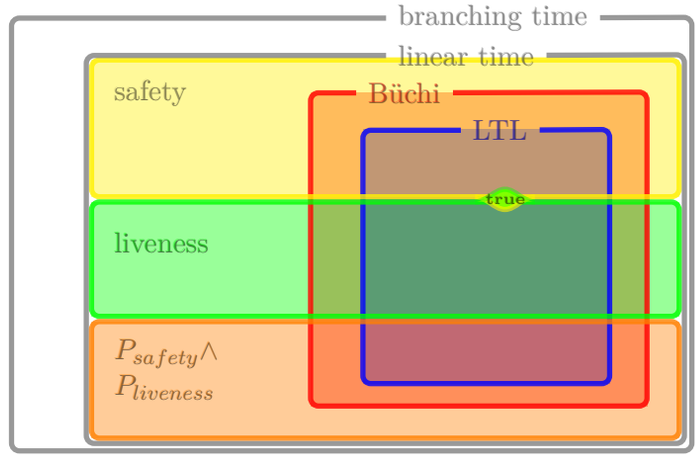


Figure 4.1.: Classification of linear time properties

4.2.1. Branching time properties

Definition 4.3. A **branching time property** is an arbitrary behavioral property in $2^{\mathbb{S}_{Kripke}/\approx_\Sigma}$.

Note. By Def. 4.3, branching time properties cover structures that can branch, i.e., need not be deterministic. The kind of Kripke structures $\mathcal{S} = (S, T, \Sigma, I)$ on which

satisfaction of branching time properties are evaluated is not constrained in this thesis, in contrast to most literature, where T is restricted to transitive, asymmetric, or total relations, for example. Totality, which is the only constraint in [Clarke et al., 1999b] and [Baier and Katoen, 2008], is not required in this thesis, where S is only restricted to the reachable states where necessary (cf. Def. 3.17). Therefore a real end state s can be used without the workaround of self-loops (i.e., $s \to s$, cf. page 71).

If some state $s \in S$ needs to be distinguished on different paths, copies of s can be used on the different paths, i.e., $\mathcal{S}$ can be unwound. For instance if q_3 of $\mathcal{S}$ on the left in Fig. 11.1 (page 255) should be reachable from q_1 only for $i > 0$, but i should not be observable, a copy of q_1 can be used, leading to $\mathcal{S}'$ on the right. Completely **unwinding** $\mathcal{S}$ this way results in its computation tree, as defined in Def. 4.4.

Definition 4.4. Let $\mathcal{S} \in \mathbb{S}_{Kripke}$ with $S^0 = \{init\}$.

Then the **computation tree** of $\mathcal{S}$ (***computationTree*($\mathcal{S}$)**) is the unique tree with the root *init* and the children of a node s are exactly a copy of each element in $dest(s, \to)$ of $\mathcal{S}$.

Therefore, $\forall \pi_1, \pi_2 \in paths^{fin}(computationTree(\mathcal{S})) : dest(\pi_1) = dest(\pi_2) \implies \pi_1 = \pi_2$ and $\mathcal{S} \approx_\Sigma computationTree(\mathcal{S})$. $\mathbb{S}_{Kripke}/\approx_\Sigma$ can be represented with the (potentially) infinite data structure $tree(node) := (node \times tree(node)^{\mathbb{N}}) \dot\cup \{\varnothing\}$, with each *node* from $S/\approx_\Sigma$.

4.2.2. Linear time properties

Definition 4.5. A Kripke structure (S, T, Σ, I, S^0) is **linear** iff it is deterministic and $|S^0| = 1$. $\mathbb{S}_{Kripke,linear}$ denotes the set of all linear Kripke structures.

Definition 4.6. A **linear time property** is a behavioral property in $2^{\mathbb{S}_{Kripke,linear}/\approx_\Sigma}$.

Notes. Since $\mathcal{S}_{lin} \in \mathbb{S}_{Kripke,linear}$ is deterministic, $computationTree(\mathcal{S}_{lin})$ (i.e., $\mathcal{S}_{lin}$'s complete unwinding) is graph isomorphic to the unique path in $paths_{max}(\mathcal{S}_{lin}, \{init\})$, denoted $\boldsymbol{\pi_{(\mathcal{S}_{lin},\{init\})}}$ (or $\boldsymbol{\pi_{\mathcal{S}_{lin}}}$ for short). $\mathcal{S}_{lin}$ has **finite length** $:\Leftrightarrow |\pi_{\mathcal{S}_{lin}}| \in \mathbb{N}_{\geq 0}$ (which is not equivalent to S being finite). If, conversely, $\mathcal{S}_{arbitrary} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke}$, $s \in S$ and a path $\pi = (s_i)_i \in paths(\mathcal{S}_{arbitrary}, s)$ are given, a linear Kripke structure $\mathcal{S}_{(\mathcal{S}_{arbitrary},\pi)}$ (or $\mathcal{S}_\pi$ for short) with $\pi_{(\mathcal{S}_\pi)}$ graph isomorphic to π can be constructed: $\boldsymbol{\mathcal{S}_\pi} := (S', T', \Sigma, I')$ with $S' := [0, \ldots, 1 + |\pi|)$, $T' := \{(i-1, i) | 0 < i < 1 + |\pi|\}$ and $\forall p \in \Sigma : I'(i, p) := I(s_i, p)$. For a lasso $\pi = \pi_{prefix} \cdot (\pi_{loop})^\omega$, we can alternatively choose the finite linear Kripke structure with S' restricted to $[0, \ldots, |\pi_{prefix}| + |\pi_{loop}| - 1]$ and T' additionally containing $(|\pi_{prefix}| + |\pi_{loop}| - 1, |\pi_{prefix}|)$. As specialization from the previous section, we then have $\mathcal{S}_{lin} \approx_\Sigma \mathcal{S}_{(\pi_{\mathcal{S}_{lin}})}$, and $\mathbb{S}_{Kripke,linear}/\approx_\Sigma$ can be represented with the (potentially) infinite data structure $sequence(node) := node^* \dot\cup node^\omega$, with each *node* from $S/\approx_\Sigma$. Hence, such a sequence can be considered as finite or infinite word with letters from 2^Σ. With this and $k \in \mathbb{N}_{\geq 0}$, $\boldsymbol{(\mathcal{S}_{lin})_{\geq k}} := \mathcal{S}_{(\pi_{\mathcal{S}_{lin}})_{\geq k}}$ and $\boldsymbol{(\mathcal{S}_{lin})_{\leq k}} := \mathcal{S}_{(\pi_{\mathcal{S}_{lin}})_{\leq k}}$ can be defined.

This thesis uses $\mathbb{S}_{Kripke,linear}/\approx_\Sigma$ as default structure for linear time properties, since this is a subset of $\mathbb{S}_{Kripke}/\approx_\Sigma$ and abstracts from irrelevant details of other representations (e.g., whether unwounded), and the thesis represents as much as possible via Kripke structures.

Definition 4.7. A **counterexample** to a linear time property P is an element $\mathcal{S} \in \mathbb{S}_{Kripke,linear}/\approx_\Sigma$ (or $\mathcal{S} \in \mathbb{S}_{Kripke,linear}$ or its path $\pi_\mathcal{S}$) with $\mathcal{S} \not\models P$.

A **witness** to a linear time property P is an element $\mathcal{S} \in \mathbb{S}_{Kripke,linear}/\approx_\Sigma$ (or $\mathcal{S} \in \mathbb{S}_{Kripke,linear}$ or its path $\pi_\mathcal{S}$) with $\mathcal{S} \models P$.

As depicted in the Euler diagram in Fig. 4.1, linear time properties (except for $P = \mathbb{S}_{Kripke,linear}/\approx_\Sigma$, denoted in temporal logic as **true**, cf. Sec.4.3) are partitioned into safety properties, liveness properties, and $P_{safety} \wedge P_{liveness}$, with the following definitions, similar to [Baier and Katoen, 2008]:

Definition 4.8. A **safety property** P is a linear time property for which all counterexamples have a prefix causing refutation, i.e., for all counterexamples to P, it can be determined after finitely many transitions that P does not hold: With Σ given, $\forall \mathcal{S} \in \mathbb{S}_{Kripke,linear} : (\mathcal{S} \not\models P \implies \exists i \in \mathbb{N}\ (\forall \mathcal{S}' \in \mathbb{S}_{Kripke,linear} \text{ with } \mathcal{S}'_{\leq i} \approx_\Sigma \mathcal{S}_{\leq i}) :\ \mathcal{S}' \not\models P)$.

Definition 4.9. A **liveness property** P is a linear time property for which no counterexample has a prefix causing refutation: With Σ given, $\forall \mathcal{S} \in \mathbb{S}_{Kripke,linear}\ \forall i \in \mathbb{N}\ \exists \mathcal{S}' \in \mathbb{S}_{Kripke,linear} : (\mathcal{S}'_{\leq i} \approx_\Sigma \mathcal{S}_{\leq i}$ and $\mathcal{S}' \models P)$.

Example. A descriptive safety property is P_1 :=*the program does not terminate within 42 steps.* A safety property often describes the reachability property that some bad state property, p, never happens ($[]!p$, cf. Subsec. 4.3). A liveness property often describes that some good state property, p will eventually happen ($<> p$, cf. Subsec. 4.3), e.g., P_2 :=*the program eventually terminates.*

Safety and liveness properties are disjoint (except for **true**), but there are also other linear time properties: those for which some but not all refuting traces have a prefix causing refutation, e.g., the property $P_1 \wedge P_2$. Because of the following lemma, given in Baier and Katoen [2008], this thesis describes such properties as $\boldsymbol{P_{safety} \wedge P_{liveness}}$.

Lemma 4.10. *A linear time property that is neither a safety nor a liveness property can be specified as a conjunction of a safety and a liveness property.*

4.3. Temporal Logics

A behavioral property is usually not given explicitly as subset of $\mathbb{S}_{Kripke}/\approx_\Sigma$, but implicitly using some temporal logic formalism, e.g., a formula or an automaton.

Definition 4.11. Temporal logics are formalisms that specify behavioral properties. $\boldsymbol{TL}$ denotes the set of all **property descriptions in temporal logics** for behavioral properties.

Def. 4.12 describes when a system specification satisfies a property description, Def. 4.13 relates property descriptions to behavioral properties. Def. 4.14 and Def. 4.15 define concise formalisms in relation to semantics.

Definition 4.12. Let $\mathcal{S} = (S, T, \Sigma, I, S^0) \in \mathbb{S}_{Kripke}$, $s \in S$, temporal logic $\mathcal{L} \subseteq TL$ and $F \in \mathcal{L}$. Then:

- $val_{(\mathcal{S},s)} : \mathcal{L} \to \mathbb{B}$ is the **evaluation function**, which extends I to $\mathcal{L}$, describing $\mathcal{L}$'s semantics;
- $(\mathcal{S}, s)$ **satisfies** $F :\Leftrightarrow val_{(\mathcal{S},s)}(F) = \textbf{true}$ (written $(\mathcal{S}, s) \models\!\!\!\models F$);
- $\mathcal{S}$ **satisfies** $F :\Leftrightarrow \forall s \in S^0 : (\mathcal{S}, s) \models F$ (written $\mathcal{S} \models F$)
  $\Leftrightarrow$: $\mathcal{S}$ is a **model** for F.

Definition 4.13. Let temporal logic $\mathcal{L} \subseteq TL, F \in \mathcal{L}$ and behavioral property $P \subseteq \mathbb{S}_{Kripke}/\approx_\Sigma$. Then:

$$P \text{ is } \textbf{specified} \text{ by } F :\Leftrightarrow \forall \mathcal{S} \in \mathbb{S}_{Kripke} : (\mathcal{S} \models P \Leftrightarrow \mathcal{S} \models F).$$

Definition 4.14.

$$\begin{aligned}
\boldsymbol{Prop}: &\quad TL \to 2^{\mathbb{S}_{Kripke}/\approx_\Sigma}, F \mapsto Prop(F) := \\
&\qquad \{[\mathcal{S}] \in \mathbb{S}_{Kripke}/\approx_\Sigma | \mathcal{S} \models F\}; \\
\boldsymbol{Prop_{lin}}: &\quad TL \to 2^{\mathbb{S}_{Kripke,linear}/\approx_\Sigma}, F \mapsto Prop_{lin}(F) := \\
&\qquad Prop(F) \cap \mathbb{S}_{Kripke,linear}/\approx_\Sigma; \\
\boldsymbol{Models}: &\quad TL \to 2^{\mathbb{S}_{Kripke}}, F \mapsto Models(F) := \\
&\qquad \{\mathcal{S} \in \mathbb{S}_{Kripke} | \mathcal{S} \models F\}; \\
\boldsymbol{Models_{lin}}: &\quad TL \to 2^{\mathbb{S}_{Kripke,linear}}, F \mapsto Models_{lin}(F) := \\
&\qquad Models(F) \cap \mathbb{S}_{Kripke,linear}.
\end{aligned}$$

Definition 4.15. Let $F_1, F_2 \in TL$.
Then $F_1 \equiv F_2 :\Leftrightarrow \forall \mathcal{S} \in \mathbb{S}_{Kripke} : (\mathcal{S} \models F_1 \Leftrightarrow \mathcal{S} \models F_2)$.
We say F_1 and F_2 are **equivalent**.

Notes. Since a temporal logic $\mathcal{L} \subseteq TL$ specifies behavior, its semantics must be determined by extending I to $val_{(\mathcal{S},s)}$ by only considering $I(\cdot, s)$ and not s itself. So the requirement on temporal logics is that $\approx_\Sigma$ is a congruence relation with respect to $\models$, i.e., f.a. $F \in \mathcal{L}$ f.a. $\mathcal{S}_1, \mathcal{S}_2 \in \mathbb{S}_{Kripke}$ with $\mathcal{S}_1 \approx_\Sigma \mathcal{S}_2 : (\mathcal{S}_1 \models F \Leftrightarrow \mathcal{S}_2 \models F)$. Therefore $\models$ is well defined on $\mathbb{S}_{Kripke}/\approx_\Sigma$ (i.e., $[\mathcal{S}_1]_{\approx_\Sigma} \models F \Leftrightarrow \mathcal{S}_1 \models F$) and so is $Prop$ and $Models$.

Put differently, P is specified by F iff $P = Prop(F)$ and $F_1 \equiv F_2 :\Leftrightarrow Prop(F_1) = Prop(F_2)$.

Originating from philosophy and linguistics, [Pnueli, 1977] transferred temporal logics to computer science to specify behavioral properties for reactive systems, and to reason about them.

Though property descriptions in temporal logics are often formulas (e.g., CTL$^* \subseteq TL$, cf. Subsec. 4.3.1), this is not necessarily so (e.g., Büchi automata $\subseteq TL$ are also allowed, cf. Subsec. 4.3.2).

Closure of a temporal logic, see Def. 4.16, is an interesting and useful property since it simplifies some tasks and proofs (e.g., Lemma 5.8).

Definition 4.16. Let temporal logic $\mathcal{L} \subseteq TL$ and n-ary operator O be given. Then:

- $\mathcal{L}$ is **closed under** O $:\Leftrightarrow \forall F_1, \ldots, F_n \in \mathcal{L} : \exists F \in \mathcal{L}$ with $F \equiv \text{O}(F_1, \ldots, F_n)$;
- $\mathcal{L}$ is **closed** $:\Leftrightarrow$ $\mathcal{L}$ is closed under all its operators;

- the smallest set $S \subseteq TL$ that is closed under some operators is called the **closure** of those operators.

The semantics of temporal logics are defined differently here than in the standard literature, e.g., [Clarke et al., 1999b]: they can be interpreted in Kripke structures with finitely many states and end states, too (cf. page 71).

Roadmap. Subsec. 4.3.1 describes temporal logics for branching time properties (**branching time logics** for short), and its most prominent representatives CTL* and CTL. Subsec. 4.3.2 considers temporal logics for linear time properties, and gives LTL and (extended) Büchi automata as examples.

The Euler diagram in Fig. 4.3 gives an overview over these and further temporal logics and their relationships (discussed in Sec. 4.4).

4.3.1. Temporal Logics for Branching Time Properties

CTL*

CTL* is the propositional temporal logic that extends propositional logic with the existential path quantifier E, the **unary modality** X, called next time operator, and the **binary modality** U, called until operator, to express properties about the whole Kripke structure, i.e., about behavior.

Syntax. The distinction between linear time and branching time properties is reflected in the syntactic structure of CTL* and its subsets CTL (see Subsec. 4.3.1) and LTL (see Subsec. 4.3.2): they are all defined by linear time and branching time formulas.

To handle branching time aspects, **branching time formulas** (called state formulas in [Clarke and Draghicescu, 1988]) are able to existentially quantify over $paths_{max}(\mathcal{S}, s)$ for a given state s using the **existential path quantifier E** (also denoted by $\exists$):

$$\begin{aligned} \langle \text{branching} \rangle ::= {} & \langle \text{atomic prop} \rangle \mid (\langle \text{branching} \rangle \vee \langle \text{branching} \rangle) \mid \\ & \neg \langle \text{branching} \rangle \mid \text{E } \langle \text{linear} \rangle; \\ \langle \text{atomic prop} \rangle ::= {} & p; \qquad\qquad \text{where } p \in \Sigma \end{aligned}$$

To handle linear time aspects, **linear time formulas** (called path formulas in [Clarke and Draghicescu, 1988]) use the **next time operator X** and the **until operator U**:

$$\begin{aligned} \langle \text{linear} \rangle ::= {} & \langle \text{branching} \rangle \mid (\langle \text{linear} \rangle \vee \langle \text{linear} \rangle) \mid \neg \langle \text{linear} \rangle \mid \\ & (\langle \text{linear} \rangle \text{ U } \langle \text{linear} \rangle) \mid \text{X } \langle \text{linear} \rangle; \end{aligned}$$

Definition 4.17. Computation Tree Logic∗ (CTL*) is the set of all branching time formulas.

Note. For all formulas, parentheses can be inserted for disambiguation.

The syntax of CTL* shows that it is closed under all its operators.

Semantics. To specify the semantics of CTL* , Def. 4.18 inductively defines $\models$ over its syntactic structure.

Definition 4.18. Let $\mathcal{S} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke}$, $s \in S, \pi \in paths_{max}(\mathcal{S}, s)$, b_i branching time formulas and l_i linear time formulas. Then:

- $(\mathcal{S}, s) \models p \quad :\Leftrightarrow \quad I(p, s) = \mathbf{true} \qquad (p \in \Sigma)$
- $(\mathcal{S}, s) \models \neg b_1 \quad :\Leftrightarrow \quad (\mathcal{S}, s) \not\models b_1$
- $(\mathcal{S}, s) \models b_1 \vee b_2 \quad :\Leftrightarrow \quad (\mathcal{S}, s) \models b_1$ or $(\mathcal{S}, s) \models b_2$
- $(\mathcal{S}, s) \models \mathrm{E}\ l_1 \quad :\Leftrightarrow \quad \exists \pi \in paths_{max}(\mathcal{S}, s) : (\mathcal{S}, \pi) \models l_1$
- $(\mathcal{S}, \pi) \models b_1 \quad :\Leftrightarrow \quad (\mathcal{S}, source(\pi)) \models b_1$
- $(\mathcal{S}, \pi) \models \neg l_1 \quad :\Leftrightarrow \quad (\mathcal{S}, \pi) \not\models l_1$
- $(\mathcal{S}, \pi) \models l_1 \vee l_2 \quad :\Leftrightarrow \quad (\mathcal{S}, \pi) \models l_1$ or $(\mathcal{S}, \pi) \models l_2$
- $(\mathcal{S}, \pi) \models \mathrm{X}\ l_1 \quad :\Leftrightarrow \quad |\pi| > 0$ and $(\mathcal{S}, \pi_{\geq 1}) \models l_1$
- $(\mathcal{S}, \pi) \models l_1 \mathrm{U}\ l_2 \quad :\Leftrightarrow \quad \exists k \in [0, \ldots, 1 + |\pi|) : ((\mathcal{S}, \pi_{\geq k}) \models l_2$ and $\forall j \in [0, \ldots, k) : (\mathcal{S}, \pi_{\geq j}) \models l_1)$

Notes. For CTL*, linear time formulas can express more than linear time properties: because of the syntactic definition $\langle \text{linear} \rangle ::= \langle \text{branching} \rangle \mid \ldots$, the linear time aspects and branching time aspects are intertwined and the linear time formulas contain all of CTL*.

Both the syntax and semantics of CTL* allow formulas with directly cascaded path quantifiers, i.e., without linear time operators between them (e.g., A E A X p). Since all but the last quantifier are irrelevant for the semantics, they can be ignored. Hence this thesis does not consider such formulas, but also does not forbid them.

The semantics of CTL* show that the propositional logic operators have **set theoretic semantics** for $\mathbb{S}_{Kripke,1}$: Let $b_1, b_2 \in$ CTL*, then

- $Models(b_1 \wedge b_2) = Models(b_1) \cap Models(b_2)$
- $Models(b_1 \vee b_2) = Models(b_1) \cup Models(b_2)$
- $Models(\neg b_1) = \mathbb{S}_{Kripke,1} \setminus Models(b_1)$.

Termination. Let $\mathbb{S}_{\boldsymbol{Kripke,\geq\omega}} := \{\mathcal{S} \in \mathbb{S}_{Kripke} | \forall \pi \in paths_{max}(\mathcal{S}) : |\pi| = \omega\}$ and $\mathbb{S}_{\boldsymbol{Kripke,<\omega}} := \{\mathcal{S} \in \mathbb{S}_{Kripke} | \forall \pi \in paths_{max}(\mathcal{S}) : |\pi| < \omega\}$.

Usually, temporal logics use **infinite trace semantics**, i.e., paths with finite length are not allowed. Instead, a **self-loop** is added to an originally final end state $s \in S$ (cf. [Clarke et al., 1999b] or [Holzmann, 2004]), resulting in an infinite path where the last state is repeated (called **stuttering** in [Lamport, 1983]) infinitely. So instead of having an **end state**, i.e., $s \not\rightarrow$, $dest(s, \rightarrow) = \{s\}$ is used, and only $\mathbb{S}_{Kripke,\geq\omega}$ is considered. Avoiding finite paths in this way results in fewer case distinctions in proofs about some temporal logic properties (e.g., Theorem 4.46). But the approach has multiple disadvantages:

1. $\mathcal{S}$ needs to be modified, i.e., the kind of Kripke structures allowed are restricted (e.g., by requiring totality, cf. Subsec. 4.2.1);
2. termination is not directly expressible in such a temporal logic. Instead, termination needs to be specified individually for each Kripke structure $\mathcal{S}$: each valid terminating state is specified manually, for instance in [Manna and Pnueli, 1995, Properties of Terminating Programs, page 61ff] by introducing a propositional variable e_i that is set in exactly that state; if $\mathcal{S}$ has n valid terminating states, $e := e_1 \vee \cdots \vee e_n$ specifies being in a valid terminating state. This approach quickly becomes infeasible, especially since some optimizations and reductions (e.g., program slicing, see Subsec. 5.4.3) are incompatible with this solution;
3. an end state with a self-loop is indistinguishable from an originally already present self-loop. If distinction is required, e.g., for several safety checks and for livelock detection (cf. Chapter 6), originally final end states again need to be specified individually, as in Item 2, or handled by some workaround;
4. it is not suitable when traces need to stay finite (e.g., for testing and bounded model checking);
5. unintuitive next-time operator, e.g., X p might hold in an end state;
6. counterexamples need not be shortest because of the self-loops.

To avoid item 2 and 3 in infinite trace semantics, a state $\bot \notin S$ with $dest(\bot, \to) = \{\bot\}$ can be added into $\mathcal{S}$, and a transition $s \to \bot$ from every originally final end state $s \in S$ (similarly to Def. 4.44, see also [Baier and Katoen, 2008]. But the other disadvantages still hold, and

- modularity becomes more difficult, since $\bot$ must either be shared when modules are combined, or multiple $\bot$ exist, which must be handled or checked for;
- the stronger the Kripke structures must be modified, the more error-prone and costly it gets to create and handle them;
- formulas become more complex, since they need to take $\bot$ with its meaning in all situations into account. If they do not, item 5 from above still holds.

Hence we avoid the workaround of transforming Kripke structures and rather take the more general approach of allowing finite paths, too. So we cover arbitrary Kripke structures (but can still restrict ourselves in certain situations or use self-loops if we want to). Our semantics, which we call finite $\cup$ infinite trace semantics (explained below), do not have the disadvantages listed above. Simultaneously allowing both finite and infinite traces were also covered in [Kamp, 1968] for the linear case, where the finite case is investigated on the last few pages, and in [Baier and Katoen, 2008] for CTL. We use it consistently throughout this chapter, up to ECTL* [Dam, 1994].

With finite $\cup$ infinite trace semantics, the semantics of the operator X differs for finite paths from the usual approach: If $s \in S$ with $s \not\to$, then in this thesis $(\mathcal{S}, s) \not\models \mathrm{X}\ p$, which models X more intuitively, closer to the diamond operator of modal logics (cf. [Fitting and Mendelsohn, 1999]), and makes corner cases more explicit. These semantics were introduced for the linear case by the philosophical dissertation [Kamp, 1968], used for CTL in [Baier and Katoen, 2008] and adapted in [Manna and Pnueli, 1995; Bauer et al., 2010] by introducing the **runtime trace semantics FLTL** (by adding the weak next operator X_w, cf. Def. 4.24) for runtime verification (cf. Sec. 2.2), i.e., when considering finite traces only. They are sometimes also referred to as **finite trace semantics**, e.g., in [Bauer and Haslum, 2010], where several variants exist [Manna and Pnueli,

1995; Eisner et al., 2003; Fraser and Wotawa, 2006]. Finite trace semantics only cover finite but extensible traces, i.e., only consider $\mathbb{S}_{Kripke,<\omega}$. In this thesis, however, we allow both finite traces for termination as well as infinite traces, which we call **finite ∪ infinite trace semantics**. This is more than the finite and infinite cases combined, i.e., $\mathbb{S}_{Kripke} \supsetneq \mathbb{S}_{Kripke,<\omega} \dot\cup \mathbb{S}_{Kripke,\geq\omega}$. For instance, a reactive system that is waiting for input in a potentially endless loop and only terminates upon a specific input exposes self-loops, end states and infinite traces. These cases all having different meaning and are treated appropriately with finite ∪ infinite trace semantics.

As consequence of our semantics, the formula $F_{fin} = \neg \mathrm{X}\ \mathbf{true}$ can be used to detect end states. Using a self-loop in s results in $(\mathcal{S}, s) \not\models F_{fin}$. Furthermore, when a formula with top-level operator E or U is interpreted in an end state, the algorithms for our semantics may only choose the shortest path, of length 0, as solution or counterexample.

On the mathematical level, this chapter shows how the general approach of allowing finite paths can be carried through consequently up to Büchi automata (cf. Def. 4.29), translations from LTL to Büchi (cf. Def. 4.44), and statements about the expressiveness of temporal logics (cf. Theorem 4.49): some of the following proofs about temporal logic properties have to consider more cases, but definitions and structures become simpler and more general, leading to richer statements. In practice, though, many formulas are free of the next time operator (cf. page 76), for which the semantics are the same for finite, infinite and finite ∪ infinite trace semantics.

Secondary Operators. Up to now, we have only considered the operator basis $\{\vee, \neg,\ \mathrm{E},\ \mathrm{X},\ \mathrm{U}\}$. Just like other binary Boolean operators, the following temporal operators can be expressed within our operator basis, with l_i being linear time formulas.

- the **universal path quantifier A** l_1 (or $\forall\ l_1$), as abbreviation for $\neg\,\mathrm{E}\,\neg l_1$. So the semantics is:
  $(\mathcal{S}, s) \models \mathrm{A}\ \ l_1 \ :\Leftrightarrow\ \forall \pi \in paths_{max}(\mathcal{S}, s) : (\mathcal{S}, \pi) \models l_1$;

- the **sometimes operator F** l_1 (or $\Diamond\ l_1$, also called **diamond** or **eventually operator**), as abbreviation for $(\mathbf{true}\,\mathrm{U}\,l_1)$. So the semantics is:
  $(\mathcal{S}, \pi) \models \mathrm{F}\,l_1 \ :\Leftrightarrow\ \exists k \in [0, \ldots, 1 + |\pi|) : ((\mathcal{S}, \pi_{\geq k}) \models l_1$;

- the **always operator G** l_1 (or $\Box\ l_1$, also called **box operator** or **globally operator**), as abbreviation for $\neg\,\mathrm{F}\,\neg\,l_1$. So the semantics is:
  $(\mathcal{S}, \pi) \models \mathrm{G}\,l_1 \ :\Leftrightarrow\ \forall k \in [0, \ldots, 1 + |\pi|) : ((\mathcal{S}, \pi_{\geq k}) \models l_1$;

- the **release operator** $(l_1 \mathbf{R}\ l_2)$, as abbreviation for $\neg(\neg l_1\,\mathrm{U}\,\neg l_2)$. So the semantics is:
  $(\mathcal{S}, \pi) \models l_1\ \mathrm{R}\ l_2 \ :\Leftrightarrow\ min\{i \in [0, \ldots, 1 + |\pi|) | (\mathcal{S}, \pi_{\geq i}) \models l_1\} \leq min\{i \in [0, \ldots, 1 + |\pi|) | (\mathcal{S}, \pi_{\geq i}) \models \neg l_2\}$, or G l_2;

- the **weak until operator** $(l_1 \mathbf{W}\ l_2)$, or $(l_1 \mathbf{U}_{weak}\ l_2)$ (also called **unless operator**), as abbreviation for $(l_1\,\mathrm{U}\,l_2) \vee \mathrm{G}\ l_1$. So the semantics is:
  $(\mathcal{S}, \pi) \models l_1\ \mathrm{W}\ l_2 \ :\Leftrightarrow\ \exists k \in [0, \ldots, 1 + |\pi|) : ((\mathcal{S}, \pi_{\geq k}) \models l_2$ and $\forall j \in [0, \ldots, k) : (\mathcal{S}, \pi_{\geq j}) \models l_1)$, or $\forall k \in [0, \ldots, 1 + |\pi|) : (\mathcal{S}, \pi_{\geq k}) \models l_1$.

Computation Tree Logic

Computation Tree Logic (**CTL**) is the main representative for a logic describing branching time properties.

Syntax. CTL is a subset of CTL* that strongly restricts CTL*'s linear time aspects by only allowing one linear time aspect (operator X or U) per branching time aspect (operator E). So CTL restricts CTL*'s linear time formulas, but retains CTL*'s branching time formulas:

$$\begin{aligned}\langle \text{branching} \rangle ::= {} & \langle \text{atomic proposition} \rangle \mid (\langle \text{branching} \rangle \vee \langle \text{branching} \rangle) \mid \\ & \neg \langle \text{branching} \rangle \mid \text{E } \langle \text{linear} \rangle; \\ \langle \text{linear} \rangle ::= {} & \text{ X } \langle \text{branching} \rangle \mid (\langle \text{branching} \rangle \text{ U } \langle \text{branching} \rangle) \mid \\ & \neg(\text{ X } \langle \text{branching} \rangle) \mid \neg(\langle \text{branching} \rangle \text{ U } \langle \text{branching} \rangle)\end{aligned}$$

Definition 4.19. Computation Tree Logic (**CTL**) is the set of all branching time formulas.

This results in the following **temporal operators of CTL** (**CTL operators** for short), which intertwine branching time and linear time aspect (and are sometimes confusingly denoted as **branching time operators**): $\mathbf{EX}\ b_1$, $\mathbf{E}\neg\ \mathbf{X}\ b_1$, $\mathbf{E}(b_1\ \mathbf{U}\ b_2)$, $\mathbf{E}(\neg\ (b_1\ \mathbf{U}\ b_2))$.

Semantics. Since CTL is a subset of CTL*, its semantics is determined by the definition of $\models$ in Def. 4.18. So a CTL formula is interpreted over arbitrary Kripke structures.

Note. The syntax of CTL shows that it is closed under propositional logic operators and CTL operators.

The semantics of CTL show that the propositional logic operators have set theoretic semantics for $\mathbb{S}_{Kripke,1}$.

Secondary Operators. We may use the following abbreviations in CTL, with b_i being branching time formulas: $\mathbf{EF}\ b_1$, $\mathbf{E}\neg\ \mathbf{F}\ b_1$, $\mathbf{EG}\ \ b_1$, $\mathbf{E}\neg\ \mathbf{G}\ b_1$, $\mathbf{E}(b_1\ \mathbf{R}\ b_2)$, $\mathbf{E}(\neg\ (b_1\ \mathbf{R}\ b_2)$, and all CTL operators with the universal instead of the existential path quantifier. This is possible since substituting A, F, G, R with their definition (given in the previous subsection) in these abbreviations directly results in formulas that are conform to CTL's syntax.

We can also use the weak until operator in CTL:

- $\mathbf{E}(b_1\ \mathbf{W}\ b_2)$ as abbreviation for $\text{E}\,(b_1\ \text{U}\ b_2)\ \vee\ \text{EG}\ b_1$;
- $\mathbf{A}(b_1\ \mathbf{W}\ b_2)$ as abbreviation for $\neg\text{E}\,(\neg b_2\ \text{U}\ \neg(b_1 \vee b_2))$ since $\neg\text{E}\ \neg(b_1\ \text{W}\ b_2) = \neg\text{E}\,(\neg(b_1\ \text{U}\ b_2) \wedge \neg(\text{G}\ b_2)) = \neg\text{E}\,((\text{G}\ b_2 \vee (\neg b_2\ \text{U}\ \neg(b_1 \vee b_2))) \wedge \neg(\text{G}\ b_2))$.

4.3.2. Temporal Logics for Linear Time Properties

Linear Time Logic

Linear Time Logic (**LTL**) restricts CTL* to purely linear time properties (cf. Subsec. 4.4) and is the main representative of a logic for linear time properties.

Syntax. LTL strongly restricts CTL*'s branching time formulas, but retains CTL*'s linear time formulas:

$$\begin{aligned}\langle\text{branching}\rangle &::= \langle\text{atomic proposition}\rangle;\\ \langle\text{linear}\rangle &::= \langle\text{branching}\rangle \mid (\langle\text{linear}\rangle \vee \langle\text{linear}\rangle) \mid \neg\langle\text{linear}\rangle \mid\\ &\quad (\langle\text{linear}\rangle \text{ U } \langle\text{linear}\rangle) \mid \text{X } \langle\text{linear}\rangle;\end{aligned}$$

Definition 4.20. Linear Time Logic (**LTL**) is the set of all linear time formulas.

Semantics. An LTL formula L is interpreted over linear Kripke structures (cf. Def. 4.5) as L describes a linear time property. The semantics of L is determined by $\models$ for CTL*, defined in Def. 4.18, because for $\mathcal{S}_{lin} \in \mathbb{S}_{Kripke,linear} : \mathcal{S}_{lin} \approx_\Sigma \mathcal{S}_{(\pi_{S_{lin}})}$, and CTL*'s semantics does not consider the states themselves but their interpretation I. Therefore $(\mathcal{S}_{lin}, init) \models L :\Leftrightarrow (\mathcal{S}_{lin}, \pi_{\mathcal{S}_{lin}}) \models L$ in CTL*.

An LTL formula L can be identified with $\neg E \neg L$ (i.e., with A L, see abbreviations in Subsec. 4.3.1). This does not change the semantics of L for linear Kripke structures, because of Lemma 4.21. But with this universal path quantification, LTL formulas can also be interpreted over arbitrary Kripke structures, as defined in Def. 4.22. Therefore LTL is really a subset of CTL*.

Lemma 4.21. *Let $\mathcal{S} = (S, T, \Sigma, I)$ be a deterministic Kripke structure, state $s \in S$, formula $L \in$ LTL and path quantifier $Q \in \{E, A\}$.*
Then $(\mathcal{S}, s) \models Q\ L \Leftrightarrow (\mathcal{S}, \pi_{\mathcal{S}}) \models L$.

Proof. $(\mathcal{S}, s) \models Q\ L$
$\Leftrightarrow$ for the unique path $\pi \in paths_{max}(\mathcal{S}, s) : (\mathcal{S}, \pi) \models L$
$\Leftrightarrow (\mathcal{S}, \pi_{(\mathcal{S},s)}) \models L$. □

Definition 4.22. Let $\mathcal{S} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke}, s \in S$ and $L \in$ LTL.
Then $(\mathcal{S}, s) \models L :\Leftrightarrow \forall \pi \in paths_{max}(\mathcal{S}, s) : (\mathcal{S}_\pi, s) \models L$.

Notes. The syntax of LTL shows that it is closed under all its operators.

For temporal logics for linear time properties, the propositional logic operators do not have set theoretic semantics for $\mathbb{S}_{Kripke,1}$ (cf. Def. 4.22): A counterexample is the CTL* formula A $\neg$ X $q \in$ LTL. Its complement is the CTL* formula E X q, which is not expressible in LTL (cf. Lemma 4.52). But LTL's semantics show that its propositional logic operators have set theoretic semantics for $\mathbb{S}_{Kripke,linear}$.

Secondary Operators. The secondary linear-time operators F, G, R and W of CTL* can also be used in LTL since they can all be reduced to until operators, which can be used in LTL in arbitrary nesting.

Sometimes, we restrict the set of formulas we consider: only formulas in negation normal form, as defined in Def. 4.23, or next-free formulas, as defined in Def. 4.25.

Definition 4.23. A formula in **negation normal form** contains the negation only directly in front of atomic propositions.

To transform a formula L into an equivalent formula L_{NNF} in negation normal form, we need dual operators, i.e., the ability to move negation outside in (i.e., top down from the highest level to the lowest in L's abstract syntax tree). Since $\neg$ X $\neg p \not\equiv$ X p for end states, we need to introduce another operator, X_w, defined in Def. 4.24.

Definition 4.24. The **weak next operator** is defined as

$$\boldsymbol{X_w}\langle\text{linear}\rangle := \neg X \neg\langle\text{linear}\rangle.$$

Thus L_{NNF} contains the operators U, R, X, $X_w, \wedge, \vee$, and in front of atomic propositions also the operator $\neg$.

Definition 4.25. $\mathbf{LTL_{-X}}$ is the set of **next-free LTL formulas**, i.e., constructed from the operator basis $\{\vee, \neg,$ U $\}$.

LTL_{-X} is an important subclass of LTL since LTL_{-X} is sufficiently expressive in practice and those formulas exactly expresses the linear time properties that are closed under stuttering (cf. Def. 4.26 and Lemma 4.27), i.e., those formulas are insensitive to stuttering: finitely repeating a state s (or $[s]/{\approx_\Sigma}$) on a path [Clarke et al., 1999b; Holzmann, 2004; Baier and Katoen, 2008].

Definition 4.26. Let $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_{Kripke,linear}, \pi := \pi_\mathcal{S}, \pi' := \pi_{\mathcal{S}'}$. Then

- $\pi \rightarrow_{stutter} \pi' :\Leftrightarrow \exists k \in \mathbb{N}_{\geq 0} : [\pi_{\leq k}]/{\approx_\Sigma} = [\pi'_{\leq k}]/{\approx_\Sigma}$ and $[\pi_{>k}]/{\approx_\Sigma} = [\pi'_{>k+1}]/{\approx_\Sigma}$ and $source([\pi'_{>k}]) = source([\pi'_{>k+1}])$; we say $[\pi']$ **stutters** at k;
- $\boldsymbol{\approx_{st}}$, called **stutter equivalence**, is the reflexive, transitive and symmetric closure of $\rightarrow_{stutter}$ over paths;
- likewise, $\approx_{st}$ is an equivalence relation over $\mathbb{S}_{Kripke,linear}$ and $\mathbb{S}_{Kripke,linear}/{\approx_\Sigma}$;
- F is **invariant under stuttering** $:\Leftrightarrow \forall \mathcal{S}_1, \mathcal{S}_2 \in \mathbb{S}_{Kripke,linear}$ with $\mathcal{S}_1 \approx_{st} \mathcal{S}_2 : (\mathcal{S}_1 \in Models_{lin}(F) \Leftrightarrow \mathcal{S}_2 \in Models_{lin}(F))$.

Lemma 4.27. *Let $F \in LTL_{-X}$. Then $Prop_{lin}(F)$ is closed under stuttering (i.e., $P \in Prop_{lin}(F)$, $P' \in [P]/{\approx_{st}} \Rightarrow P' \in Prop_{lin}(F)$).*

Proof. The proof is a simple induction on the complexity of F, cf. [Baier and Katoen, 2008]. □

Note. Lemma 4.27 is important for partial order reduction on LTL_{-X} (cf. Subsec. 5.4.1).

Though this thesis makes no use of it, the converse of Lemma 4.27 also holds: If linear time property $P \in 2^{\mathbb{S}_{Kripke,linear}/{\approx_\Sigma}}$ is closed under stuttering, then there is $F \in \text{LTL}_{-X}$ with $Prop_{lin}(F) = P$ [Clarke et al., 1999b].

Büchi Automata

More complex linear time properties are either not practically expressible in LTL and CTL*, or not at all. Hence we use an automaton $\mathcal{A}$ to specify linear time properties: The structure of $\mathcal{A}$ is identical to an FSM, but the condition of a path in $paths_{max}(\mathcal{A})$ being accepting is modified, resulting in ω-**automata**, defined in Def. 4.28.

Definition 4.28. Let signature Σ be fixed and $\mathcal{A} = (Q, \Delta, A, Q^0, F) \in \mathbb{S}_{FSM}$ with $A = 2^\Sigma$.

Then $\mathcal{A}$ is an ω-**automaton**. $\mathbb{S}_{\boldsymbol{\omega\text{-automaton}}}$ denotes the set of all ω-automata.

Depending on how acceptance is defined, different classes of ω-automata arise. Büchi automata are one of the most prominent ω-automata. They use the acceptance condition given in Def. 4.29. This definition deviates from the usual one (cf. [Clarke et al., 1999b; Baier and Katoen, 2008]) in that maximal finite paths may be accepted, too.

Definition 4.29. Let the signature Σ be fixed, $\mathcal{A} \in \mathbb{S}_{\omega-\text{automaton}}$ and $\pi \in paths_{max}(\mathcal{A})$.
Then $\mathcal{A}$ is a **Büchi automaton** iff the following acceptance is used: $\mathcal{A}$ **accepts** $\pi :\Leftrightarrow$
$\begin{cases} dest(\pi) \in F & \text{if } \pi \in paths^{fin}_{max}(\mathcal{A}); \\ |\{i \in \mathbb{N} | s_i \in F\}| = \omega & \text{if } \pi \in paths^{\omega}(\mathcal{A}). \end{cases}$
Büchi denotes the set of all Büchi automata.

Def. 4.30 lifts acceptance for an ω-automaton $\mathcal{A}$ from $paths_{max}(\mathcal{A})$ to $\mathbb{S}_{Kripke,linear}$, defining satisfiability.

Definition 4.30. Let the signature Σ be fixed, $\mathcal{S} \in \mathbb{S}_{Kripke,linear}$ and $\mathcal{A} \in \mathbb{S}_{\omega-\text{automaton}}$. Then
$(\mathcal{S}, init)$ **satisfies** $\mathcal{A}$ (written $(\mathcal{S}, init) \models \mathcal{A}$) $:\Leftrightarrow$
$\exists \pi \in paths_{max}(\mathcal{A}) : \mathcal{A}$ accepts π and both π and $\mathcal{S}$ describe the same linear behavior: $|\pi| = 1 + |\pi_{\mathcal{S}}|$ and for $(s_i \xrightarrow{l_i} s_{i+1}) := \pi$ and $(s'_i)_i := \pi_{\mathcal{S}}$ we have $\forall i \in [0, \ldots, 1 + |\pi_{\mathcal{S}}|) : l_i = supp(I(\cdot, s'_i))$.

Def. 4.31 in turn lifts satisfiability from $\mathbb{S}_{Kripke,linear}$ to $\mathbb{S}_{Kripke}$ by universal path quantification, similarly to Def. 4.22 for LTLs. With this and the more general definition of Büchi acceptance in Def. 4.29, this thesis again allows arbitrary Kripke structures, as for the previous temporal logics.

Definition 4.31. Let $\mathcal{S} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke}$, $s \in S$ and $\mathcal{A} \in \mathbb{S}_{\omega-\text{automaton}}$.
Then $(\mathcal{S}, s) \models \mathcal{A} :\Leftrightarrow \forall \pi \in paths_{max}(\mathcal{S}, s) : (\mathcal{S}_\pi, s) \models \mathcal{A}$.

For some classes of ω-automata and acceptance conditions, the FSM (Q, Δ, A, Q^0, F) must be generalized, for instance such that F is a subset of Q^2 (Streett automata, Rabin automata), a subset of 2^Q (Muller automata), cf. [Farwer, 2001], or a finite set of subsets of Q (extended Büchi automata), cf. Def. 4.32. Except for the acceptance conditions, all properties of $\mathbb{S}_{\omega-\text{automaton}}$ apply for these automata, too.

Definition 4.32. Let the signature Σ be fixed, $\mathcal{A} = (Q, \Delta, 2^\Sigma, Q^0, \mathcal{F})$ with $n \in \mathbb{N}_{>0}, \mathcal{F} = (F_i)_{i \in [1,\ldots,n]}$ and $\forall i \in [1, \ldots, n] : F_i \subseteq Q$. Let $\pi \in paths_{max}(\mathcal{A})$.
Then $\mathcal{A}$ is an **extended Büchi automaton** iff the following acceptance is used: $\mathcal{A}$ **accepts** $\pi :\Leftrightarrow$
$\forall j \in [1, \ldots, n] : \begin{cases} dest(\pi) \in F_j & \text{if } \pi \in paths^{fin}_{max}(\mathcal{A}); \\ |\{i \in \mathbb{N} | s_i \in F_j\}| = \omega & \text{if } \pi \in paths^{\omega}(\mathcal{A}). \end{cases}$
Extended-Büchi denotes the set of all extended Büchi automata.

For given Büchi automata $\mathcal{A}_1$ and $\mathcal{A}_2$, a new one can be constructed that reflects a combination of $\mathcal{A}_1, \mathcal{A}_2$; the intersection (cf. Def. 4.33) of $\mathcal{A}_1, \mathcal{A}_2$ is particularly important, since this construction can be used to emulate extended Büchi automata (cf. Lemma 4.43), to emulate LTL formulas (cf. Lemma 4.46) and to perform LTL model checking (cf. Subsec. 5.3.2).

Definition 4.33. Let Σ be fixed and $\mathcal{A}_1 = (Q_1, \Delta_1, 2^\Sigma, Q_1^0, F_1), \mathcal{A}_2 = (Q_2, \Delta_2, 2^\Sigma, Q_2^0, F_2) \in$ Büchi.

Then the **conjunctive Büchi automaton** is $\mathcal{A}_1 \cap \mathcal{A}_2 := (Q, \Delta, 2^\Sigma, Q^0, F)$ with

- $Q := Q_1 \times Q_2 \times \{1,2\}$;
- $Q^0 := Q_1^0 \times Q_2^0 \times \{1\}$;
- $\Delta := \{((q_1,q_2,x), a, (q_1',q_2',x')) \in Q \times 2^\Sigma \times Q | (q_1,a,q_1') \in \Delta_1,$
  $(q_2,a,q_2') \in \Delta_2$ and $x' = \begin{cases} 2 & \text{if } x = 1 \text{ and } q_1 \in F_1; \\ 1 & \text{if } x = 2 \text{ and } q_2 \in F_2; \\ x & \textit{otherwise}; \end{cases} \}$;
- $F := (\{q_1 \in F_1 | q_1 \not\to\} \times \{q_2 \in F_2 | q_2 \not\to\} \times \{1,2\}) \dot\cup$
  $(\{q_1 \in F_1 | q_1 \to\} \times \{q_2 \in Q_2 | q_2 \to\} \times \{1\})$.

Notes 4.34. For the special case of $F_1 = Q_1$ (or $F_2 = Q_2$), the construction of $\mathcal{A}_1 \cap \mathcal{A}_2$ can be simplified by collapsing the third dimension $\{1,2\}$ of Q, resulting in the synchronous product (cf. Def. 3.32), in this case:

- $Q := Q_1 \times Q_2$;
- $Q^0 := Q_1^0 \times Q_2^0$;
- $\Delta := \{((q_1,q_2), a, (q_1',q_2')) \in Q \times 2^\Sigma \times Q | (q_1,a,q_1') \in \Delta_1$
  and $(q_2,a,q_2') \in \Delta_2\}$;
- $F := (\{q_1 \in F_1 | q_1 \not\to\} \times \{q_2 \in F_2 | q_2 \not\to\}) \dot\cup$
  $(\{q_1 \in F_1 | q_1 \to\} \times \{q_2 \in F_2 | q_2 \to\})$.

Since Δ uses the synchronous product, i.e., requires corresponding transitions in both Δ_1 and Δ_2 with the same label, a restriction of the enabled labels in $\mathcal{A}_i$ also restricts the enabled labels in $\mathcal{A}_1 \cap \mathcal{A}_2$, hence its state space often has much fewer states than Q.

Restrictions of the enabled labels can yield new end states, which necessitate workarounds [Baier and Katoen, 2008] unless our finite $\cup$ infinite trace semantics are used (cf. page 71).

Lemma 4.35. *Let $\mathcal{A}_1, \mathcal{A}_2 \in$ Büchi.*
Then $Models_{lin}(\mathcal{A}_1 \cap \mathcal{A}_2) = Models_{lin}(\mathcal{A}_1) \cap\ Models_{lin}(\mathcal{A}_2)$.

Proof. Let $\mathcal{S} \in Models_{lin}(\mathcal{A}_1 \cap \mathcal{A}_2)$ with $\pi \in paths_{max}(\mathcal{A}_1 \cap \mathcal{A}_2)$ the path accepted by $\mathcal{A}_1 \cap \mathcal{A}_2$ with the same linear behavior as $\mathcal{S}$. Let $((q_i^1, q_i^2, x_i) \xrightarrow{l_i} (q_{i+1}^1, q_{i+1}^2, x_{i+1}))_i := \pi$.

Then by construction, $(q_i^1 \xrightarrow{l_i} q_{i+1}^1)_i$ is a path accepted by $\mathcal{A}_1$ with the same linear behavior as $\mathcal{S}$, and $(q_i^2 \xrightarrow{l_i} q_{i+1}^2)_i$ is a path accepted by $\mathcal{A}_2$ with the same linear behavior as $\mathcal{S}$: If these paths are finite, they all end in accepting end states; if the paths are infinite, they must all visit infinitely many accepting states since π infinitely often alternates between traversing a state in $\{q_1 \in F_1 | q_1 \to\} \times \{q_2 \in Q_2 | q_2 \to\} \times \{1\}$ and one in $\{q_1 \in Q_1 | q_1 \to\} \times \{q_2 \in F_2 | q_2 \to\} \times \{2\}$.

Let now $\mathcal{S} \in Models_{lin}(\mathcal{A}_1) \cap Models_{lin}(\mathcal{A}_2)$ with $(q_i^1 \xrightarrow{l_i} q_{i+1}^1)_i$ the path accepted by $\mathcal{A}_1$ with the same linear behavior as $\mathcal{S}$, and $(q_i^2 \xrightarrow{l_i} q_{i+1}^2)_i$ the path accepted by $\mathcal{A}_2$ with the same linear behavior as $\mathcal{S}$.

Then $((q_i^1, q_i^2, x_i) \xrightarrow{l_i} (q_{i+1}^1, q_{i+1}^2, x_{i+1}))_i := \pi$ with $x_0 = 1$ and all other x_i determined by Δ is a path accepted by $\mathcal{A}_1 \cap \mathcal{A}_2$ with the same linear behavior as $\mathcal{S}$. □

The proof also shows that for the special cases of $F_1 = Q_1$ or $F_2 = Q_2$, the third dimension of Q can really be collapsed.

Notes. Besides the conjunctive Büchi automaton, we define Büchi automata for other propositional logic operators for $\mathbb{S}_{Kripke,linear}$: The disjunctive Büchi automaton (cf. Def. 4.36 and Lemma 4.37) and the complement Büchi automaton (cf. Lemma 4.38). These are far less important than the conjunctive Büchi automaton since they are not required in practice, but still interesting and theoretically helpful (e.g., for defining ECTL* below). These automata show that Büchi automata are closed under propositional logics operators.

Being a temporal logics for linear time properties, the propositional logic operators do not have set theoretic semantics for $\mathbb{S}_{Kripke,1}$ (cf. Def. 4.31), only for $\mathbb{S}_{Kripke,linear}$. As for LTL, the CTL* formula A $\neg$ X q is a counterexample for $\mathbb{S}_{Kripke,1}$ (cf. Corollary 4.50).

Definition 4.36. Let Σ be fixed and $\mathcal{A}_1 = (Q_1, \Delta_1, 2^\Sigma, Q_1^0, F_1), \mathcal{A}_2 = (Q_2, \Delta_2, 2^\Sigma, Q_2^0, F_2) \in$ Büchi.

Then the **disjunctive Büchi automaton** is $\mathcal{A}_1 \cup \mathcal{A}_2 := (Q, \Delta, 2^\Sigma, Q^0, F)$ with

- $Q := Q_1 \dot\cup Q_2 \dot\cup \{init\}$;
- $Q^0 := \{init\}$;
- $\Delta := \Delta_1 \dot\cup \Delta_2 \dot\cup \{(init, a, q) \in Q \times 2^\Sigma \times Q \mid \exists i \in \{1,2\} \; \exists q_0 \in Q_i^0 : (q_0, a, q) \in \Delta_i\}$;
- $F := F_1 \dot\cup F_2$.

Lemma 4.37. *Let $\mathcal{A}_1, \mathcal{A}_2 \in$ Büchi.*

Then $Models_{lin}(\mathcal{A}_1 \cup \mathcal{A}_2) = Models_{lin}(\mathcal{A}_1) \cup Models_{lin}(\mathcal{A}_2)$.

Proof. An accepting path in $paths_{max}(\mathcal{A}_1 \cup \mathcal{A}_2)$ has accepting states either in F_1 or in F_2, and $F = F_1 \dot\cup F_2$. Furthermore, paths in $paths(\mathcal{A}_1 \cup \mathcal{A}_2)$ are equal to paths in $paths(\mathcal{A}_i) \dot\cup paths(\mathcal{A}_i)$ except for the first state. Therefore, the linear behaviors described by accepting paths in $paths_{max}(\mathcal{A}_1 \cup \mathcal{A}_2)$ are equal to the linear behaviors described by accepting paths in $paths_{max}(\mathcal{A}_1) \cup paths_{max}(\mathcal{A}_2)$. □

Lemma 4.38. *Let $\mathcal{A} \in$ Büchi.*

*Then there exists a **complement Büchi automaton** $\mathcal{A}^c$ such that $\mathcal{A}^c$ is the complement to $\mathcal{A}$ in relation to $\mathbb{S}_{Kripke,linear}$: $Models_{lin}(\mathcal{A}^c) = \mathbb{S}_{Kripke,linear} \setminus Models_{lin}(\mathcal{A})$.*

Proof. Let $\mathcal{A} \in$ Büchi and $A \subseteq paths_{max}(\mathcal{A})$ be its accepting paths. Since we use infinite and finite trace semantics, we transform $\mathcal{A}$ to nondeterministically distinct at the beginning between finite and infinite accepting paths; for this, we construct two Büchi automata:

- $\mathcal{A}_\omega$ is a copy of $\mathcal{A}$ with $F_\omega := \{f \in F | f\rightarrow\}$, so its accepting paths are exactly $A \cap paths^\omega(\mathcal{A})$;
- $\mathcal{A}_{fin}$ is a copy of $\mathcal{A}$ with $F_{fin} := \{f \in F | f\not\rightarrow\}$, so its accepting paths are exactly $A \cap paths_{max}^{fin}(\mathcal{A})$.

Thus $\mathcal{A} \equiv \mathcal{A}_{fin} \cup \mathcal{A}_\omega$. We construct complements $\mathcal{A}'_\omega$ for the infinite accepting paths and $\mathcal{A}'_{fin}$ for the finite accepting paths, such that $\mathcal{A}^c = \mathcal{A}'_\omega \cup \mathcal{A}'_{fin}$: As in [Sistla et al., 1985], we construct $\mathcal{A}'_\omega = (Q', \Delta', 2^\Sigma, Q^{0\prime}, F')$ with $\forall f \in F' : f\rightarrow$ and $Models_{lin}(\mathcal{A}'_\omega) = \mathbb{S}_{Kripke,linear,\geq\omega} \setminus Models_{lin}(\mathcal{A})$. As in [Hopcroft and J.D. Ullman, 1979], we determinize $\mathcal{A}_{fin}$ and then complement it, resulting in $\mathcal{A}'_{det} = (Q'_{det}, \Delta'_{det}, 2^\Sigma, Q^{0\prime}_{det}, F'_{det})$.

We duplicate each state $f \in F'_{det}$ together with all its incoming and outgoing transitions, but do not put the duplicate in F'_{det}. We remove all outgoing transition of f. The result is a Büchi automaton $\mathcal{A}'_{fin}$ with $\forall f \in F'_{fin} : f \not\to$ and $Models_{lin}(\mathcal{A}'_{fin}) = \mathbb{S}_{Kripke,linear,<\omega} \setminus Models_{lin}(\mathcal{A})$. Thus $\mathcal{A}^c = \mathcal{A}'_{fin} \cup \mathcal{A}'_{\omega}$ has the required properties: $Models_{lin}(\mathcal{A}^c) = Models_{lin}(\mathcal{A}'_{fin}) \cup Models_{lin}(\mathcal{A}'_{\omega}) = (\mathbb{S}_{Kripke,linear,\geq\omega} \setminus Models_{lin}(\mathcal{A})) \cup (\mathbb{S}_{Kripke,linear,<\omega} \setminus Models_{lin}(\mathcal{A})) = \mathbb{S}_{Kripke,linear} \setminus Models_{lin}(\mathcal{A})$. □

4.4. Relationships

Definition 4.39. Let temporal logics $\mathcal{L}_1, \mathcal{L}_2 \subseteq TL$ and $F_2 \in \mathcal{L}_2$. Then:

- F_2 is **expressible** in $\mathcal{L}_1 :\Leftrightarrow \exists F_1 \in \mathcal{L}_1 : F_1 \equiv F$;
- $\mathcal{L}_1 \geq \mathcal{L}_2 :\Leftrightarrow \forall F_2 \in \mathcal{L}_2 : F_2$ is expressible in $\mathcal{L}_1$, we say $\mathcal{L}_1$ is **at least as expressive as** $\mathcal{L}_2$;
- $\mathcal{L}_1 \gneq \mathcal{L}_2 :\Leftrightarrow \mathcal{L}_1 \geq \mathcal{L}_2$ and $\mathcal{L}_2 \not\geq \mathcal{L}_1$, we say $\mathcal{L}_1$ is **more expressive than** $\mathcal{L}_2$;
- $\mathcal{L}_1 \equiv \mathcal{L}_2 :\Leftrightarrow \mathcal{L}_1 \geq \mathcal{L}_2 \geq \mathcal{L}_1$, we say $\mathcal{L}_1$ and $\mathcal{L}_2$ are **equally expressive**.

Definition 4.40. Let temporal logic $\mathcal{L} \subseteq TL$ and property $P \subseteq \mathbb{S}_{Kripke/\approx_\Sigma}$. Then:

- P is **specifiable** in $\mathcal{L} :\Leftrightarrow \exists F \in \mathcal{L} : P = Prop(F)$;
- $\boldsymbol{Prop(\mathcal{L})} \subseteq 2^{\mathbb{S}_{Kripke/\approx_\Sigma}}$ is the set of properties that are specifiable in $\mathcal{L}$.

Lemma 4.41. *Let temporal logics $\mathcal{L}_1, \mathcal{L}_2 \subseteq TL$. Then:*
$\mathcal{L}_1 \geq \mathcal{L}_2 \Leftrightarrow Prop(\mathcal{L}_1) \supseteq Prop(\mathcal{L}_2)$.

Proof. Let $\mathcal{L}_1 \geq \mathcal{L}_2, P \in Prop(\mathcal{L}_2)$. Thus $\exists F \in \mathcal{L}_2 : \forall \mathcal{S} \in \mathbb{S}_{Kripke} : (\mathcal{S} \models P \Leftrightarrow \mathcal{S} \models F)$ and $\exists F' \in \mathcal{L}_1 : F' \equiv F$. Hence $\forall \mathcal{S} \in \mathbb{S}_{Kripke} : (\mathcal{S} \models P \Leftrightarrow \mathcal{S} \models F')$, so P is specified by F' and $P \in Prop(\mathcal{L}_1)$.
Let $Prop(\mathcal{L}_1) \supseteq Prop(\mathcal{L}_2)$ and $F \in \mathcal{L}_2$. Thus $Prop(F) \in Prop(\mathcal{L}_2) \subseteq Prop(\mathcal{L}_1)$ and so $\exists F' \in \mathcal{L}_1 : F' \equiv F$. □

Lemma 4.42. *Let $\Sigma = \{p\}$ and $\boldsymbol{even(p)} := \{\mathcal{S} \in \mathbb{S}_{Kripke,linear} | (s_i)_i := \pi_{\mathcal{S}}$ and $|\pi_{\mathcal{S}}| = \omega$ and $\forall j \in \mathbb{N}_{\geq 0} : (j$ is even $\Rightarrow I(p, s_j) = \textbf{true})\} / \approx_\Sigma$.*
Then even(p) $\notin$ Prop(LTL), but even(p) $\in$ Prop(Büchi).

Proof. [Wolper, 1983, Corollary 4.2] proved that $even(p) \notin Prop(\text{LTL})$ by showing that any $F \in \text{LTL}$ over $\Sigma = \{p\}$ with n next operators has the same truth value for all $\{\mathcal{S} \in \mathbb{S}_{Kripke,linear} | (s_i)_i := \pi_{\mathcal{S}}$ and $\exists j > n : supp(I(\neg p, \cdot)) = \{s_j\}\}$. Furthermore, $even(p)$ is specified by $\boldsymbol{\mathcal{A}_{even}}$, the Büchi automaton in Fig. 3.2 on page 37 with $l_0 = l_1 = \{p\}$ and $l_2 = \varnothing$, which exactly restricts every even step to be p. □

Lemma 4.43. *Extended-Büchi $\equiv$ Büchi.*

Proof. Since every Büchi automaton is also an extended Büchi automaton, Extended-Büchi $\geq$ Büchi. Let $n \in \mathbb{N}_{>0}, \mathcal{A}_{ext} = (Q, \Delta, 2^\Sigma, Q^0, (F_i)_{i \in [1,\ldots,n]}) \in$ Extended-Büchi and $\pi \in paths_{max}(\mathcal{A}_{ext})$.

Then $\mathcal{A}_{ext}$ accepts π
$\Leftrightarrow \forall j \in [1,\ldots,n] : \begin{cases} dest(\pi) \in F_j & \text{if } \pi \in paths^{fin}_{max}(\mathcal{A}); \\ |\{i \in \mathbb{N} | s_i \in F_j\}| = \omega & \text{if } \pi \in paths^{\omega}(\mathcal{A}). \end{cases}$
$\Leftrightarrow \bigcap_{i\in[1,\ldots,n]} (Q, \Delta, 2^\Sigma, Q^0, F_i)$ accepts π (cf. Def. 4.33).
Thus Büchi $\geq$ Extended-Büchi, too. □

For Theorem 4.46 about the relationship between Büchi and LTL, we construct a special Büchi automaton $\mathcal{A}_{LTL}$ as defined in Def. 4.44, which has the property given in Lemma 4.45. Theorem 4.46 is a generalization of [Wolper, 2000; Schmitt, 2012b], as it allows Kripke structures with end states, too. It is more general than the German [Walther, 2011], as it also allows Kripke structures with infinite length, and is more thorough since it proves its propositions, which [Walther, 2011] forgets to do, besides the fact that for his propositions, a normal form for LTL and the adapted transformation from Extended-Büchi automata to Büchi automata are missing. Theorem 4.46 is also more general than [Giannakopoulou and Havelund, 2001], which only considers the temporal operator U, not X, and finite traces, which are then translated to FSMs.

Definition 4.44. Let signature Σ and $L_{NNF} \in$ LTL in negation normal form (cf. Def. 4.24) be given.

Then we define $\mathcal{A}_{L_{NNF}} := (Q, \Delta, 2^\Sigma, Q^0, \mathcal{F})$ with:

- $\boldsymbol{subF}(L_{NNF}) :=$ the set of all sub-formulas of L_{NNF};
- $Q := \{q \subseteq subF(L_{NNF}) | ((C_1 \vee C_2) \in q \implies C_1 \in q \text{ or } C_2 \in q)$ and $((C_1 \wedge C_2) \in q \implies C_1, C_2 \in q)$ and $\mathbf{false} \notin q\}$;
- $Q_\perp := \{\perp\} \dot\cup Q$ (with $\forall F \in \text{LTL} : F \notin \perp$);
- $Q^0 := \{q \in Q | L_{NNF} \in q\}$;
- $\Delta \subseteq Q \times 2^\Sigma \times Q_\perp$ with$(q_1, a, q_2) \in \Delta :\Leftrightarrow$
  $\forall A, B \in subF(L_{NNF})$ all 5 cases apply :
  1. $\forall p \in \Sigma : (p \in q_1 \implies p \in a)$ and $(\neg p \in q_1 \implies p \notin a)$;
  2. $XA \in q_1 \implies A \in q_2$;
  3. $X_w A \in q_1 \implies (A \in q_2 \text{ or } q_2 = \perp)$;
  4. $A \mathbf{U} B \in q_1 \implies (B \in q_1 \text{ or } (A \in q_1 \text{ and } A \mathbf{U} B \in q_2))$;
  5. $A \mathbf{R} B \in q_1 \implies (A \in q_1 \text{ or } A \mathbf{R} B \in q_2 \text{ or } q_2 = \perp)$ and $B \in q_1$;
- $(A_i \mathbf{U} B_i)_i :=$ the finite sequence of all $A \mathbf{U} B \in subF(L_{NNF})$ with $k := |(A_i \mathbf{U} B_i)_i|$ and $\forall i \in [1,\ldots,k] : U_i := A_i \mathbf{U} B_i$;
- $\mathcal{F}_i := \{q \in Q_\perp | \text{X } \mathbf{false} \notin q \text{ and } (U_i \notin q \text{ or } U_i \in q \text{ and } B_i \in q)\}$ with $i \in [1,\ldots,k]$;
- $\mathcal{F} := \{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$ if $k > 0$, otherwise $\mathcal{F} := \{\{q \in Q_\perp | \text{X } \mathbf{false} \notin q\}\}$.

Lemma 4.45. *Let signature Σ and $L_{NNF} \in$ LTL in negation normal form be given.*
Then $\mathcal{A}_{L_{NNF}} \equiv L_{NNF}$.

Proof. To prove $\forall \mathcal{S} \in \mathbb{S}_{Kripke,linear} : \mathcal{S} \models \mathcal{A}_{L_{NNF}} \implies \mathcal{S} \models L_{NNF}$, let $\mathcal{S} \in Models_{lin}(\mathcal{A}_{L_{NNF}})$ and $(s_i \xrightarrow{l_i} s_{i+1})_i \in paths_{max}(\mathcal{A}_{L_{NNF}})$ the accepting path describing the same linear behavior (cf. Def.4.30).

We prove $\forall C \in subF(L_{NNF}) \forall i \in [0, \ldots, 1 + |(s_i)_i|)$:

$$C \in s_i \Rightarrow \mathcal{S}_{\geq i} \models C \tag{4.1}$$

via induction on the complexity of C:

- C is a literal l :
  Because of case 1 of Δ's definition, $l \in$ label l_i. Since $\mathcal{S}$ and $(s_i \xrightarrow{l_i} s_{i+1})_i$ describe the same linear behavior, $\mathcal{S}_{\geq i} \models C$.
- $C = (C_1 \wedge C_2)$ or $C = (C_1 \vee C_2)$: By Q's definition, $(C_1 \wedge C_2) \in s_i \implies C_1$ and $C_2 \in s_i$ (respectively $(C_1 \vee C_2) \in s_i \implies C_1$ or $C_2 \in s_i$). $\mathcal{S}_{\geq i} \models C$ follows by the induction hypothesis.
- $C = XC_1$:
  Then $C_1 \in s_{i+1}$ and, by the induction hypothesis, $\mathcal{S}_{\geq i+1} \models C_1$, so $\mathcal{S}_{\geq i} \models C$.
- $C = X_w C_1$:
  Then $C_1 \in s_{i+1}$ or $s_{i+1} = \bot$. If $C_1 \in s_{i+1}$, then $\mathcal{S}_{\geq i+1} \models C_1$, so $\mathcal{S}_{\geq i} \models C$. If $s_{i+1} = \bot$, then $\mathcal{S}$ ends with $s_i \xrightarrow{l_i} s_{i+1}$, so $\mathcal{S}_{\geq i} \models X_w C_1$ because of X_w's semantics.
- $C = C_1 \mathbf{U} C_2$:
  Then by Δ's definition, only two cases may occur: either $C_2 \in s_i$, so $\mathcal{S}_{\geq i} \models C_2$ by the induction hypothesis, and thus $\mathcal{S}_{\geq i} \models C_1 \mathbf{U} C_2$. In the second case, $C_1 \in s_i$ and $C_1 \mathbf{U} C_2 \in s_{i+1}$, so $s_{i+1} \neq \bot$. The same argumentation can now be applied inductively to s_{i+j} for subsequent $j \in \mathbb{N}$: If this induction on $i + j$ would not terminate, the result were the infinite sequence $(s_j)_{j \in [i, \ldots, \omega)}$ with $\forall j \in [i, \ldots, \omega) : C_1 \in s_j, C_1 \mathbf{U} C_2 \in s_j$ and $C_2 \notin s_j$. But this contradicts $(s_i \xrightarrow{l_i} s_{i+1})_i$ being an accepting path because of the definition of $\mathcal{F}$: with $U_h = C_1 \mathbf{U} C_2$, the infinite sequence must eventually reach an element in $\mathcal{F}_h$, which may not contain $C_1 \mathbf{U} C_2$ without containing C_2. If the induction on s_{i+j} terminates for $k > i$, the result is: $\forall j \in [i, \ldots, k) : C_1 \in s_j$ and $C_2 \in s_k$, so $\forall j \in [i, \ldots, k) : \mathcal{S}_{\geq j} \models C_1$ and $\mathcal{S}_{\geq k} \models C_2$ by our induction hypothesis. Hence $\mathcal{S}_{\geq i} \models C_1 \mathbf{U} C_2$.
- $C = C_1 \mathbf{R} C_2$:
  Then by Δ's definition, $C_2 \in s_i$ and only three cases may occur: Firstly, $C_1 \in s_i$, so $\mathcal{S}_{\geq i} \models C_1$ by the induction hypothesis, thus $\mathcal{S}_{\geq i} \models C_1 \mathbf{R} C_2$. Secondly, if $s_{i+1} = \bot$, $\mathcal{S}_i \models \mathrm{G}\ C_2$ by the induction hypothesis, so $\mathcal{S}_i \models C_1 \mathbf{R} C_2$. Finally, if $C_1 \mathbf{R} C_2 \in s_{i+1}$, the same argumentation can now be applied inductively to $i + j$ for subsequent $j \in \mathbb{N}$: If this induction on $i + j$ does not terminate, then $\forall s \in (s_j)_{j \in [i, \ldots, \omega)} : C_2 \in s$. Hence, $\mathcal{S}_i \models \mathrm{G}\ C_2$ by the induction hypothesis, so $\mathcal{S}_i \models C_1 \mathbf{R} C_2$. If this induction on $i + j$ terminates for $k > i$ with $s_k = \bot$, then likewise $\forall (s_j)_{j \in [i, \ldots, k)} : C_2 \in s$ leads to $\mathcal{S}_i \models C_1 \mathbf{R} C_2$. If this induction on $i + j$ terminates for $k > i$ with $C_1 \in s_k$, then $\forall (s_j)_{j \in [i, \ldots, k]} : C_2 \in s$. Therefore by the induction hypothesis, $\forall j \in [i, \ldots, k] : \mathcal{S}_{\geq j} \models C_2$ and $\mathcal{S}_{\geq k} \models C_1$, so $\mathcal{S}_{\geq i} \models C_1 \mathbf{R} C_2$.

Since $L_{NNF} \in s_0 \in Q^0$, equation 4.1 implies $\mathcal{S} \models L_{NNF}$. To now prove $\forall \mathcal{S} \in \mathbb{S}_{Kripke,linear} : \mathcal{S} \models L_{NNF} \implies \mathcal{S} \models \mathcal{A}_{L_{NNF}}$, let $\mathcal{S} \in Models_{lin}(L_{NNF})$, $k := |\pi_{\mathcal{S}}|$, $(s'_i)_i :=$

$\pi_{\mathcal{S}}$ and $j \in [0, \ldots, 1+k)$.

We construct an accepting path $(s_i \xrightarrow{l_i} s_{i+1})_{i\in[0,\ldots,1+k)} \in paths_{max}(\mathcal{A}_{L_{NNF}})$ that describes the same linear behavior as $\mathcal{S}$:

- l_j := $supp(I(\cdot, s'_j))$. Thus $\mathcal{S}$ and the accepting path describe the same linear behavior by construction.
- s_j := $\{C \in subF(L_{NNF}) | \mathcal{S}_{\geq j} \models C\}$; since $\mathcal{S}_{\geq j} \not\models$ **false** and $\mathcal{S}_{\geq j} \models C_1 \wedge C_2 \implies \mathcal{S}_{\geq j} \models C_1, \mathcal{S}_{\geq j} \models C_2$ and $\mathcal{S}_{\geq j} \models C_1 \vee C_2 \implies \mathcal{S}_{\geq j} \models C_1$ or $\mathcal{S}_{\geq j} \models C_2, s_j$ must be in Q.
- $s_{1+k} := \bot$ if $k \in \mathbb{N}$.

Since Δ's definition is according to LTL's semantics, $(s_j, l_j, s_{j+1}) \in \Delta$. If $k \in \mathbb{N}$, then $\pi := (s_j, l_j, s_{j+1})_{j\in[0,\ldots,1+k)}$ is accepting since $s_{1+k} = \bot$. Otherwise, $k = \omega$ and if π were not accepting, then $\exists h \exists n$ such that $\forall i > n : s_i \notin \mathcal{F}_h$. Since $k = \omega$ and $\forall C \in subF(L_{NNF}) : \mathcal{S}_{\geq j} \models C$, X **false** $\notin s_i$. Therefore $\forall i > n : U_h \in s_i$ and $B_h \notin s_i$, contradicting $\mathcal{S}_{\geq n+1} \models A_h$ **U** B_h. □

Note. If $q \in Q$ in Def. 4.44 is an accepting end state, then $q = \bot$ because $q \rightarrow$ according to Δ unless X **false** $\in q$, but then q is not accepting.

The function $\boldsymbol{subF_{lin}(\cdot)}$ can be used to indicate that linear subformulas are taken.

Theorem 4.46. *Büchi $\gtrsim$ LTL.*

Proof. By Lemma 4.42, LTL $\not\geq$ Büchi. To show that Büchi $\geq$ LTL, we construct $\mathcal{A} \in$ Büchi for a given signature Σ and $L \in$ LTL such that $\mathcal{A} \equiv L$. For this, we

1. transform L into a negation normal form L_{NNF} such that $L_{NNF} \equiv L$, by the use of X_w, see Def. 4.24;
2. construct an extended Büchi automaton $\mathcal{A}_{L_{NNF}}$ from L_{NNF} such that $\mathcal{A}_{L_{NNF}} \equiv L_{NNF}$, by the use of Lemma 4.45;
3. transform $\mathcal{A}_{L_{NNF}}$ into a Büchi automaton $\mathcal{A}$ such that $\mathcal{A} \equiv \mathcal{A}_{L_{NNF}}$, by the use of Lemma 4.43.

□

The following Def. 4.47 and Lemma 4.48 are required for Theorem 4.49 about the detailed relationship between CTL*, Büchi and LTL. It has (partly) been proved in [Clarke and Draghicescu, 1988] for LTL and simplified in [Schmitt, 2012a]. The proof in this thesis is a further simplification (caused by allowing infinite Kripke structures). Furthermore, the transfer to Büchi automata alternatively to LTL is new and results in Corollary 4.50.

Definition 4.47. For $F \in$ CTL*, F^d denotes the LTL formula obtained from F by deleting all its path quantifiers.

Note. For instance, (A F A G p)^d = F G p. Since F^d only has operators that are part of LTL, and LTL allows them in arbitrary nesting, F^d is really in LTL.

Lemma 4.48. *Let $\mathcal{S} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke}$ be deterministic, $s \in S$ and $F \in CTL^*$. Then $(\mathcal{S}, s) \models F \Leftrightarrow (\mathcal{S}, \pi_{(\mathcal{S},s)}) \models F^d$.*

Proof. Via Lemma 4.21 and a simple induction on the complexity of F. □

Theorem 4.49. *Let $F \in CTL^*$.*
Then F is expressible in LTL $\Leftrightarrow$ $F \equiv F^d$ $\Leftrightarrow$ F is expressible as Büchi automaton.

Proof. Assuming for $F \in \text{CTL}^*$ that $F \equiv F^d$, then Def. 4.47 and Theorem 4.46 show the two required expressibilities.

Assuming $F \in \text{CTL}^*$ is expressible in LTL or as Büchi automaton, let $B \in \text{LTL} \cup \text{Büchi}$ such that $F \equiv B$, $\mathcal{S} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke}$, $s \in S$.

$$
\begin{aligned}
\text{Then } (\mathcal{S}, s) \models F \quad &\overset{F \equiv B}{\Longleftrightarrow} \quad (\mathcal{S}, s) \models B \\
&\overset{\text{Def. 4.22 or 4.31}}{\Longleftrightarrow} \quad \forall \pi \in paths_{max}(\mathcal{S}, s) : \ (\mathcal{S}_\pi, s) \models B \\
&\overset{F \equiv B, \mathcal{S}_\pi \text{ det.}}{\Longleftrightarrow} \quad \forall \pi \in paths_{max}(\mathcal{S}, s) : \ (\mathcal{S}_\pi, s) \models F \\
&\overset{\text{Lemma 4.48}, \mathcal{S}_\pi \text{ det.}}{\Longleftrightarrow} \quad \forall \pi \in paths_{max}(\mathcal{S}, s) : \ (\mathcal{S}_\pi, s) \models F^d \\
&\overset{\text{semantics of LTL}}{\Longleftrightarrow} \quad \forall \pi \in paths_{max}(\mathcal{S}, s) : \ (\mathcal{S}, \pi) \models F^d \\
&\overset{\text{Def. 4.22}}{\Longleftrightarrow} \quad (\mathcal{S}, s) \models F^d.
\end{aligned}
$$

Applying this equivalence chain to all $s \in S^0$ proves the theorem. □

To relate CTL* to LTL and Büchi automata, one could argue that CTL*'s path quantifiers are "orthogonal" to the constraints that Büchi automata can pose on paths to be accepted. With Theorem 4.49, a more formal corollary and proof are possible:

Corollary 4.50. *Prop(Büchi) ∩ Prop(CTL^*) = Prop(LTL).*

Proof. For the non-trivial direction: Let property $P \in Prop(\text{Büchi}) \cap Prop(\text{CTL}^*)$. Hence $\exists F \in \text{CTL}^*, \exists \mathcal{A} \in \text{Büchi}$: F and $\mathcal{A}$ specify P, so $F \equiv \mathcal{A}$. By using Theorem 4.49 twice, F is equivalent to F^d and expressible in LTL. □

Lemma 4.51. *Property $even(p) \in Prop(\text{Büchi}) \backslash Prop(CTL^*)$.*

Proof. Lemma 4.42 shows that $even(p) \in Prop(\text{Büchi}) \backslash\ Prop(\text{LTL})$. Because of Corollary 4.50, $even(p) \notin Prop(\text{CTL}^*)$. □

Lemma 4.52. *LTL $\not\geq$ CTL.*

Proof. $F_{CTL} := (\text{EX } q) \in \text{CTL}$ but $Prop(F_{CTL}) \notin Prop(\text{LTL})$: If it were, then $\exists L \in$LTL: $Prop(L) = Prop(F_{CTL})$. So the Kripke structure $\mathcal{S}$ in Fig. 4.2 satisfies L. Because of LTL's semantics Def. 4.22, $\mathcal{S}' := \mathcal{S}$ without s_2 also satisfies L, contradicting $[\mathcal{S}']_{\approx_\Sigma} \notin Prop(F_{CTL})$. □

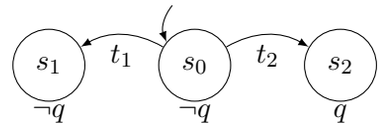


Figure 4.2.: Kripke structure $\mathcal{S}$ showing LTL $\not\geq$ CTL

Lemma 4.53. *CTL $\not\geq$ LTL.*

Proof. $F_{LTL} := (\text{FG } q) \in \text{LTL}$ but $Prop(F_{LTL}) \notin Prop(\text{CTL})$. This is shown in [Clarke and Draghicescu, 1988] using fairness constraints (cf. Subsec. 5.2.5). □

Lemma 4.54. $Prop(CTL^*) \supsetneq Prop(CTL) \cup Prop(LTL)$.

Proof. $F_{\text{CTL}^*} := F_{CTL} \vee F_{LTL} = \text{A}(\text{FG } q) \vee \text{EX } q \in \text{CTL}^*$, but $Prop(F_{\text{CTL}^*}) \notin Prop(\text{CTL})$ and $Prop(F_{\text{CTL}^*}) \notin Prop(\text{LTL})$: Assuming $Prop(F_{\text{CTL}^*}) \in Prop(\text{CTL})$, then $\exists C \in \text{CTL}$: $Prop(C) = Prop(F_{\text{CTL}^*})$. Therefore, $Prop(C \wedge \neg F_{CTL}) = Prop(F_{LTL}) \in Prop(\text{CTL})$, contradicting Lemma 4.53. Likewise assuming $Prop(F_{\text{CTL}^*}) \in Prop(\text{LTL})$, then $\exists L \in \text{LTL}$: $Prop(L) = Prop(F_{\text{CTL}^*})$. Therefore, $Prop(L \wedge \neg F_{LTL}) = Prop(F_{CTL}) \in Prop(\text{LTL})$, contradicting Lemma 4.52. □

Fig. 4.3 gives an overview over the temporal logics that we introduced in this chapter, and their relationships.

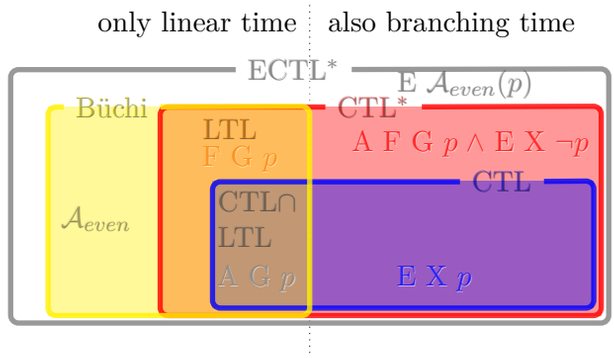


Figure 4.3.: Classification of temporal logics

Since CTL* does not contain Büchi, nor the other way around (Corollary 4.50, Lemmas 4.51 and 4.52), Fig. 4.3 additionally shows the closure of Büchi and CTL*: the **Extended Computation Tree Logic∗** (**ECTL***) [Dam, 1994].

Syntax. ECTL*'s syntax allows automata as temporal operators and is defined by:

$$
\begin{aligned}
\langle \text{branching} \rangle ::= \; & \langle \text{atomic prop} \rangle \mid (\langle \text{branching} \rangle \vee \langle \text{branching} \rangle) \mid \\
& \neg \langle \text{branching} \rangle \mid \text{A } \langle \text{Büchi} \rangle (\langle \text{branching} \rangle, \ldots, \langle \text{branching} \rangle); \\
\langle \text{atomic prop} \rangle ::= \; & p; \quad \text{where } p \in \Sigma \\
\langle \text{Büchi} \rangle ::= \; & \mathcal{A}; \quad \text{where } \mathcal{A} \in \text{ Büchi such that } \mathcal{A}(b_1, \ldots, b_n) \text{ uses} \\
& \text{the signature } \Sigma = \{b_1, \ldots, b_n\}.
\end{aligned}
$$

Definition 4.55. Extended Computation Tree Logic∗ (**ECTL***) is the set of all $\langle$branching$\rangle$ formulas.

Semantics. Our ECTL* semantics again deviates from the standard definition (cf. [Dam, 1994]) by allowing finite ∪ infinite trace semantics. It is determined by the semantics of CTL* and of Büchi automata over formulas, given in Def. 4.56 (similarly to Def. 4.30), resulting in Def. 4.57.

Definition 4.56. Let $\Sigma = \{b_1, \ldots, b_n\} \subsetneq \text{ECTL}^*$, $\mathcal{S} \in \mathbb{S}_{Kripke,linear}$ and $\mathcal{A}$ a Büchi automaton for signature Σ. Then:

- $\mathcal{A}$ is a **Büchi automaton over the formulas** $b_1, \ldots, b_n$, written $\boldsymbol{\mathcal{A}(b_1, \ldots, b_n)}$;
- $(\mathcal{S}, init)$ **satisfies** $\mathcal{A}(b_1, \ldots, b_n)$ (written $(\mathcal{S}, init) \models \mathcal{A}(b_1, \ldots, b_n)$) $:\Leftrightarrow$ $\exists \pi \in paths_{max}(\mathcal{A})$: $\mathcal{A}$ accepts π and both π and $\mathcal{S}$ describe the same linear behavior: $|\pi| = 1 + |\pi_{\mathcal{S}}|$ and for $(s_i \xrightarrow{l_i} s_{i+1}) := \pi$ and $(s'_i)_i := \pi_{\mathcal{S}}$ we have $\forall i \in [0, \ldots, 1 + |\pi_{\mathcal{S}}|) : l_i = \{b_j \in \Sigma \mid (\mathcal{S}, s'_i) \models b_j\}$.

Definition 4.57. Let $\mathcal{S} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke}$, $s \in S, p \in \Sigma, \mathcal{A} \in$Büchi and $b_i \in$ECTL*. Then:

- $(\mathcal{S}, s) \models p \quad :\Leftrightarrow \quad I(p, s) = \textbf{true}$
- $(\mathcal{S}, s) \models \neg b_1 \quad :\Leftrightarrow \quad (\mathcal{S}, s) \not\models b_1$
- $(\mathcal{S}, s) \models b_1 \vee b_2 \quad :\Leftrightarrow \quad (\mathcal{S}, s) \models b_1$ or $(\mathcal{S}, s) \models b_2$
- $(\mathcal{S}, s) \models \text{A } \mathcal{A}(b_1, \ldots, b_n) :\Leftrightarrow \quad (\mathcal{S}, s) \models \mathcal{A}(b_1, \ldots, b_n)$

Secondary Operator. As secondary operator for ECTL*, we define the **existential path quantifier E** $\mathcal{A}_1(b_1, \ldots, b_n)$ (or $\exists\, \mathcal{A}_1(b_1, \ldots, b_n)$) as abbreviation for $\neg\, \text{A}\, \mathcal{A}_1^c(b_1, \ldots, b_n)$, i.e., with the help of complementation for Büchi automata (cf. Lemma 4.38).

Notes. The syntax of ECTL* shows that it is closed under all its operators.

ECTL*'s semantics show that its propositional logic operators have set theoretic semantics for $\mathbb{S}_{Kripke,1}$.

The relationship of ECTL* with the other temporal logics of this chapter are shown in the following lemmas.

Lemma 4.58. *Let* $\Sigma = \{p\}$ *and* $\boldsymbol{Eeven(p)} := \{\mathcal{S} \in \mathbb{S}_{Kripke} | \exists \pi \in paths_{max}(\mathcal{S}) : [\mathcal{S}_\pi]_{\approx_\Sigma} \in even(p)\} / \approx_\Sigma$.

Then $Eeven(p) \notin Prop(CTL^*) \cup Prop(Büchi)$, *but* $Eeven(p) \in Prop(ECTL^*)$.

Proof. $Eeven(p) \in Prop(\text{ECTL}^*)$ since $Prop(\text{E } \mathcal{A}_{even}(p)) = Eeven(p)$.

Assume $\exists F \in \text{CTL}^*$: $Prop(F) = Eeven(p)$. Let $\mathcal{S} \in \mathbb{S}_{Kripke,linear}$. Because $\mathcal{S}$ is deterministic, $\mathcal{S} \models F$ iff $[\mathcal{S}]_{\approx_\Sigma} \in even(p)$ (analogously to Lemma 4.21). Lemma 4.48 shows that $\mathcal{S} \models F^d$ iff $[\mathcal{S}]_{\approx_\Sigma} \in even(p)$, contradicting Lemma 4.42 because $F^d \in$ LTL.

Similarly to the proof of Lemma 4.52, we assume $\exists \mathcal{A} \in$ Büchi: $Prop(\mathcal{A}) = Eeven(p)$ and give $\mathcal{S}_e$ that satisfies $\mathcal{A}$: see Fig. 4.4. Because of Def. 4.31, $\mathcal{S}'_e := \mathcal{S}_e$ without s_1 must also satisfy $\mathcal{A}$, contradicting $[\mathcal{S}'_e]_{\approx_\Sigma} \notin Eeven(p)$. □

Lemma 4.59. $ECTL^* \gneq Büchi$.

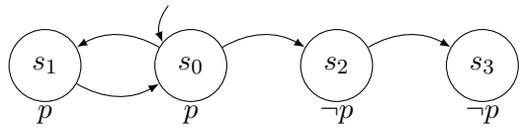


Figure 4.4.: Kripke structure $\mathcal{S}_e$ showing Büchie $\not\geq$ ECTL*

Proof. By Lemma 4.58, Büchi $\not\geq$ ECTL*. To show that ECTL$^* \geq$ Büchi, let $\mathcal{S} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke}$ with $\Sigma = \{p_1, \ldots, p_n\}, s \in S$ and $\mathcal{A} \in$ Büchi. Then $(\mathcal{S}, s) \models \mathcal{A} \Leftrightarrow (\mathcal{S}, s) \models$ A $\mathcal{A}(p_1, \ldots, p_n)$. Thus ECTL$^* \geq$ Büchi. □

Lemma 4.60. *ECTL$^* \gneq$ CTL*.*

Proof. By Lemma 4.58, CTL$^* \not\geq$ ECTL*. To show that ECTL$^* \geq$ CTL*, let $F \in$ CTL*.

Analogously to L_{NNF} for LTL (cf. Def. 4.23), F can be transformed into a normal form F_{NF} that contains no existential path quantifiers and no conjunctions, by replacing E with ¬ A ¬, $b_1 \wedge b_2$ with $\neg(\neg b_1 \vee \neg b_2)$), and collapsing multiple A into one.

Then F_{NF} can be transformed into an ECTL* formula F_{ECTL^*} such that $F_{NF} \equiv F_{\text{ECTL}^*}$: From the inside out (i.e., bottom up from the lowest level to the top-level in the abstract syntax tree), sub-formulas $F_{sub} =$ A F_{lin} (i.e. F_{sub} starts with top-level operator A followed by F_{lin} that does not start with a top-level operator A) are successively replaced with A $\mathcal{A}_{F_{lin}}(b_1, \ldots, b_n)$, where b_i is in Σ or started with top-level operator A (i.e., is an ECTL* sub-formula as result from a prior replacement). These Büchi automata exist because of Theorem 4.46. So all linear temporal parts of F_{NF} have been replaced by equivalent Büchi automata. Hence a simple induction on the complexity of F_{NF} shows that any CTL* formula can be written as an equivalent ECTL* formula; thus ECTL$^* \geq$ CTL*. □

Example. $F =$ E G $(p$ U A F $q) \wedge r$ results in $F_{NF} = \neg($ A ¬ G $(p$ U A F $q) \vee \neg r)$ and $F_{\text{ECTL}^*} = \neg($ A $\mathcal{A}_{\neg \text{ G } (b_1 \text{ U } b_2)}(p,$ A $\mathcal{A}_{\text{F } b_1}(q)) \vee \neg r)$.

4.5. Conclusion

4.5.1. Summary

This chapter has introduced a generalized theoretical foundation for behavioral properties and temporal logics for branching time properties as well as linear time properties. Finally, their relationships were investigated thoroughly (cf. Fig. 4.3 on page 85).

4.5.2. Contributions

The theoretical foundation was introduced in a generalized way by using the general automata theory from Chapter 3, so that new definitions are based on few common formalisms, and are not restricted. Therefore,

- temporal properties and temporal property descriptions like CTL*, Büchi and LTL can be compared more consistently since all have been based on Kripke structures;
- being able to compare them, their relationships have been thoroughly investigated, extending existent theorems and lemmas, and introducing new ones;

- since these Kripke structures were defined more flexibly with our finite ∪ infinite trace semantics, no workarounds are required, reductions (such as program slicing) are not restricted, and system specifications can simultaneously express termination, reactive systems that run forever, and self-loops (e.g., for waiting).

4.5.3. Future

Possible future work and open questions include:

- finding a more efficient translations from LTL to Büchi (cf. [URL:LTL2BUECHI; Wolper, 1987; Babiak et al., 2012]);
- finding and investigating a suitable case study for showing the benefits of finite ∪ infinite trace semantics. For instance, a protocol which exhibits practical use cases for all of the following situations: termination, running indefinitely, and performing a self-loop (e.g., for busy-waiting, i.e., repeatedly checking whether a condition is true, such as the availability of some resource);
- integrating further logics, such as the $\mu-$calculus, into this framework to enable further detailed comparisons between logics; it would be interesting to see whether and how easily finite ∪ infinite trace semantics can be transferred to those other logics.

5. Model Checking

5.1. Introduction

Model checking is a sound and complete formal method that checks whether a finite **system specification** $\mathcal{S}$ given as Kripke structure (or description thereof, cf. Subsec. 3.4.3) meets a **property description** F, usually given in some temporal logic. Diverse behavioral properties of the system can be verified by model checking; they depend on the kind of Kripke structure, temporal logic, and algorithm employed by the model checker.

Model checking can be used as the basis for several tasks for verifying correctness:

- pure model checking at the abstract level of the system specification to verify an algorithm or protocol, or on the implementation level to verify some source code or hardware design (cf. Chapter 6);
- checking some aspects of correctness for a piece of hardware design or source code using the more lightweight formal methods of bounded model checking (cf. Chapter 7);
- for model-based testing (cf. Part III), to generate test sequences automatically.

Example. Pure model checking at the abstract level is often used in the field of protocols and concurrent systems, since parallelization is complex and difficult to grasp and thus error-prone, calling for automated checks [Groote and Mousavi, 2014]. For example, deadlock freedom and livelock freedom are occasionally not met and hence prevalent properties to verify: A **deadlock** occurs when the whole system halts because each process is **blocked**, i.e., waits for an unsharable resource currently held by another process. Several countermeasures are possible to avoid deadlocks: For instance, a process can release the resources it is holding when it is blocked, and acquire it again later on. If this is enforced by the scheduler, it is called **preemption**. These countermeasures often become complex and lead to **livelocks**, i.e., executions where the processes are not all blocked, but they **starve**. This means no process ever acquires all resources it needs to make **progress**, i.e., to advance in the problem it has to solve. **Deadlock detection** is a safety property, which model checkers can check relatively efficiently; contrarily, **livelock detection** is a liveness property (cf. Subsec. 4.2.2) and thus harder to check (cf. Chapter 6).

Chapter 3 and Chapter 4 have introduced the theoretical foundations in a general way, allowing infinitely many states and end states in Kripke structures. For exhaustive model checking (cf. Subsec. 5.2.3), this and the following chapter mainly focus on what existing tools currently support: model checking of finite Kripke structures for infinite trace semantics with one initial state, i.e., $\mathcal{S} \in \mathbb{S}_{Kripke,1,finite,\geq\omega}$. Finite traces are ignored by using the classical approach of adding self-loops to end states. Thus, we get infinite traces but loose the information of which states have originally been end states. Subsec. 4.3.1 explained this on page 71 and showed that it has multiple disadvantages

(e.g., in Subsec. 5.4.3). Fortunately, for the most relevant model checking algorithms in this thesis (on-the-fly LTL, cf. Subsec. 5.3.2, and DFS_{FIFO}, cf. Chapter 6, without and with partial order reduction), finite $\cup$ infinite trace semantics can be integrated easily (cf. Note 5.11, Subsec. 5.6.3 and Subsec. 6.9.3); hence we use $\mathcal{S} \in \mathbb{S}_{Kripke,1,finite}$.

Unless mentioned otherwise, we only refer to sound, complete and computable model checking methods. Def. 5.1 refines the general Def. 3.2 of sound and complete formal methods to model checking. Model checking is defined in Def. 5.2, Listing 5.1 determines its input, output and contract.

Definition 5.1. We consider system specifications in $\mathbb{S}_{Kripke,1,finite}$, temporal logics $\mathcal{L} \subseteq TL$, and a model checker $\mathcal{M}$ for $\mathcal{L}$. Then:

$$\begin{aligned}
\mathcal{M} \text{ is } \textbf{sound} &:\Leftrightarrow \forall \mathcal{S} \in \mathbb{S}_{Kripke,1,finite} \forall F \in \mathcal{L} : \\
&\quad (\mathcal{M} \text{ states that } \mathcal{S} \not\models F) \Rightarrow (\mathcal{S} \not\models F) \\
\mathcal{M} \text{ is } \textbf{complete} &:\Leftrightarrow \forall \mathcal{S} \in \mathbb{S}_{Kripke,1,finite} \forall F \in \mathcal{L} : \\
&\quad (\mathcal{M} \text{ states that } \mathcal{S} \not\models F) \Leftarrow (\mathcal{S} \not\models F)
\end{aligned}$$

Definition 5.2. Let $\mathcal{S} \in \mathbb{S}_{Kripke,1,finite}$ be a system specification, $\mathcal{L} \subseteq TL$ a temporal logic and $F \in \mathcal{L}$ a property description.

Then **model checking** (**MC**) (for $\mathcal{L}$) is a (sound and complete) formal verification method that checks whether $\mathcal{S} \models F$.

For this, MC fully automatically and exhaustively (symbolically or explicitly) enumerates $\mathcal{S}$'s state space and all its behaviors that are relevant for F.

```
// PRE: S is a well-formed system specification in S_{Kripke,1,finite} (or
//      description thereof); F is a well-formed property description in L.
// POST: modelChecking(S, F) always terminates and gives as result
//      a Boolean output that is true iff S ⊨ F.
//      Optionally, a counterexample can be output if S ⊭ F.
𝔹 modelChecking(S, F)
```

Listing 5.1: Contract for model checking

The counterexample (cf. Def. 4.7) that MC optionally outputs is a path $\pi \in paths(\mathcal{S})$ that refutes the property described by F. If it is a safety property, π can be a finite path; if it is a liveness property, π can be a single unwinding of a lasso (cf. Lemma 5.9 and Note 5.10). π is also called an **error path** in $\mathcal{S}$ for the property description F. Consequently, if MC should generate a path that exhibits F, we check for the property $\neg F$, called **trap property** since it functions as a trap to catch such a path. This is also often used for model-based testing (cf. Subsec. 10.2.3).

Notes. So a false positive from an unsound model checker is a counterexample (i.e., a witness for the property $\neg F$) that is actually not a counterexample in $\mathcal{S}$. Conversely, a false negative from an incomplete model checker is the statement that $\mathcal{L}$ holds for $\mathcal{S}$ when in fact a counterexample to $\mathcal{L}$ does exist.

As for SAT solvers (cf. Note 3.7), soundness and completeness of $\models$ and $\not\models$ are related for terminating model checkers; soundness of both imply their completeness.

For linear time logics, universal path quantification (as in Def. 4.22 and Def. 4.31) is used to lift model checking from $\mathbb{S}_{Kripke,linear,finite}$ to full $\mathbb{S}_{Kripke,1,finite}$.

Note 5.3. Analogously to linear time property descriptions, branching time property descriptions in $\boldsymbol{A}$CTL* can be refuted: These are formulas whose negation normal form does not contain existential path quantifiers, so CTL$^* \supsetneq$ ACTL$^* \supsetneq$ LTL. Thus refutation can pick any one of the available paths.

Often MC operates on Kripke structures and does not require their transitions being labeled, hence this thesis defines MC on Kripke structures and only talks about labeled Kripke structures when the labels become relevant. When MC does use labels, the system specification description's basic commands, which update the state vector, are used (cf. PROMELA in Subsec. 3.4.3).

Usually states of the Kripke structure are not enumerated directly but given by a system specification description that uses a finite set of variables that resemble the data of the real world system (cf. PROMELA in Subsec. 3.4.3). These variables together form the **state vector**, which hence describes the states. For $\mathcal{S} \in \mathbb{S}_{Kripke,finite}$, the state vector and domain of the variables can always be chosen finite. In case of S being infinite, but its state space finite, we ignore unreachable states and thus identify S with its state space.

When MC uses a finite state vector, let $\boldsymbol{\Sigma_{sv}}$ be a set of propositional variables that encode it, and I_{sv} the corresponding interpretation. Then MC uses state vector semantic Σ_{sv}, as defined in Def. 5.4, since Σ_{sv} fully describes the states.

Definition 5.4. Let $\mathcal{S}_{sv} = (2^{\Sigma_{sv}}, T, \Sigma_{sv}, I_{sv}) \in \mathbb{S}_{Kripke,finite}$. Then:

- $\mathcal{S}_{sv}$ is called a **Kripke structure with state vector semantics** Σ_{sv}, with $I_{sv}(\cdot, s) = s$;
- $\boldsymbol{\mathbb{S}_{Kripke,\Sigma_{sv}}}$ is the set of all Kripke structures with state vector semantics Σ_{sv}.

Since $I_{sv}(\cdot, s) = s$, we have $\mathcal{S}_{sv} = [\mathcal{S}_{sv}]/{\approx_{\Sigma_{sv}}}$, so $\mathbb{S}_{Kripke,linear,\Sigma_{sv}} = \mathbb{S}_{Kripke,linear}/{\approx_{\Sigma_{sv}}}$, i.e., linear Kripke structures with state vector semantics are the linear time properties.

Unfortunately, modelChecking$(\mathcal{S}, F)$ often does not terminate normally because time or space runs out due to the **state space explosion**, i.e., the number of states that have to be examined becomes too large to handle: The state space grows exponentially in the number of components being specified because model checking uses the asynchronous product of the system's component automata to cover all possible interleavings, i.e., all execution orders (cf. Subsec. 3.4.3, [Peleska, 2013]). Therefore, state space explosion is caused by combinatorial explosion. State space explosion becomes particularly severe for domains with a high degree of parallelization (like protocols), where many components with little synchronization are involved.

If MC does not terminate due to state space explosion, the contract for model checking allows any behavior, and most model checkers do not give useful information. Hence **conditional model checking** (**CMC**) has been introduced [Beyer et al., 2012], which strengthens MC's contract, as given in Listing 5.2, to always terminate normally and summarize as much of the performed work as possible. For this, the **conditions** C_{input}, resp. C_{output}, describe for which parts of the state space the property description F has been proved to hold, for which parts F has been proved not to hold, and which parts have not yet been investigated – before, resp. after, the conditional model checking call (similar to pre- and post-conditions). For instance, state predicates can be used as conditions, with the special case $C_{\text{output}} = \textbf{true}$ iff $\mathcal{S} \models F$. If CMC detects $\mathcal{S} \not\models F$,

then C_{output} shows a counterexample to F and additionally for which parts of the state space CMC has verified F to hold. Therefore, if full MC fails, CMC still gives the user some information. C_{output} can be used as feedback for the user or as C_{input} for another CMC call, which can then prune parts of the state space for which F has already been verified. So for the first run, $C_{\text{input}} = \textbf{false}$ indicates that no part of the state space has been verified yet. But $C_{\text{input}} \neq \textbf{false}$ is also possible, to prune the state space (similarly to never claims, cf. Subsec. 3.4.3 on page 47) or as guidance for test case generation (cf. Chapter 11).

```
// PRE: S is a well-formed system specification in S_{Kripke,1,finite} (or description
//      thereof); F is a well-formed property description in L;
//      C_input is a well-formed description which part of S has already been verified;
// POST: modelChecking(S, F, C_input) always terminates normally
//      and gives as result a condition C_output that gives
//      the parts of the state space where F has been proved to hold,
//      the parts where F has been proved not to hold,
//      and the parts not yet proven.
// SIGNALS no exceptions.
C_output  modelChecking(S, F, C_input)
```

Listing 5.2: Contract for conditional model checking

Roadmap. Sec. 5.2 describes various classifications of model checking: explicit vs. implicit, on-the-fly vs. offline, exhaustive vs. bounded, finite vs. infinite, and ends with a practical comparison between CTL and LTL. Sec. 5.3 introduces various model checking algorithms: CTL* and CTL shortly, on-the-fly LTL in depth for later chapters. Sec. 5.4 describes reduction methods to cope with state space explosion: partial order reduction in depth for later chapters, symbolic techniques, and briefly other reductions. At the end, Sec. 5.5 introduces model checkers: SPIN, LTSmin, PRISM, and roughly DiVinE and PRISM.

5.2. Classifications of Model Checking

5.2.1. Explicit versus Implicit

MC can enumerate all states of the state space either **explicitly** (also called **explicit state model checking**) by traversing each one individually [Barnat, 2015], or **implicitly** (also called **implicit state model checking**) with the help of **symbolic techniques** (cf. Subsec. 5.4.2 or [Clarke et al., 1999b, Chapter 6]), where states and transitions are encoded in formulas (or representations thereof, like BDDs), which can be operated on and solved (e.g., using SAT or SMT solvers, cf. Chapter 3).

Note. Often symbolic MC is used synonymously for implicit state MC, but symbolic techniques can also be used for each state separately during explicit state enumeration, e.g., SPIN's minimized automaton compression (cf. Subsec. 5.4.2).

5.2.2. On-the-fly versus Offline

As opposed to **offline model checking**, **on-the-fly model checking** avoids an a priori construction of the entire state space, but rather builds it incrementally from a system specification description while checking the property description F (cf. Def. 3.33). This is done by starting in *init* and recursively traversing the transitions in $enabled(\cdot)$, which only requires states to be stored, not transitions. Since F is checked during traversal, a fault can be found while only parts of the entire state space have been constructed. The degree of saved space and time is called **on-the-flyness** (cf. Subsec. 6.8.6).

Notes. Explicit state MC is usually performed on-the-fly, since it is a big improvement and implemented easily. But implicit state MC can be implemented on-the-fly, too, cf. Subsec. 5.5.2.

Some offline MC algorithms can perform **global** MC: They not only determine whether $\mathcal{S} \models F$, but compute $\{s \in S | (\mathcal{S}, s) \models F\}$. Hence $\mathcal{S} \in \mathbb{S}_{Kripke,finite}$ with $|S^0| > 1$ can also be checked. Some applications require these global results (e.g., CTL* MC by employing a global LTL MC).

5.2.3. Exhaustive versus Bounded

Classical MC considers the full state space, i.e., does an **exhaustive** (explicit or implicit) search. Many current MC approaches still do; they aim at rigorously proving that any $\mathcal{S} \in \mathbb{S}_{Kripke,1,finite}$ meets some property F.

Alternatively, the state space can be pruned by some **bound b**, resulting in bounded model checking (**BMC**) [Biere et al., 1999; Clarke et al., 2004b], as defined in Def. 5.5. For small bounds, only part of the state space is considered, and the aim shifts to **bug finding** [Clarke et al., 2004b], i.e., not performing complete MC to prove correctness, but incomplete, lightweight checks that are more feasible and efficiently find some but not all bugs. This incomplete BMC is already useful in practice, especially since the **small scope hypothesis** usually holds when no complex data is involved [Jackson, 2006; Udupa et al., 2011], i.e., faults usually occur already with small values and after short runs. Another application of incomplete BMC is conditional model checking.

Definition 5.5. Let $\mathcal{S} \in \mathbb{S}_{Kripke,1,finite}$ be a system specification, $\mathcal{L} \subseteq TL$ a temporal logic for linear time properties, $F \in \mathcal{L}$ a property description and $b \in \mathbb{N}$.

Then **bounded model checking** (for $\mathcal{L}$) is a (computable, sound and within b complete) formal verification method that checks whether $\forall \pi \in paths_{max}(\mathcal{S}) : \pi \models_b F$, where

$$\pi \models_b F :\Leftrightarrow \begin{cases} \pi \models F & \text{if } \pi \textbf{ is within } \ b : |\pi| \leq b \text{ or } \pi = \pi_1 \cdot (\pi_2)^\omega \text{ with } |\pi_1| + |\pi_2| \leq b \\ \pi_{\leq b} \models F & otherwise. \end{cases}$$

For this, BMC fully automatically and exhaustively enumerates the state space and behaviors up to depth b.

Notes. There exist extensions of BMC to temporal logics for branching time properties [Tao et al., 2007].

For $\pi_{\leq b} \models F$, various semantics are possible: variations of finite trace semantics (cf. the paragraph on termination at page 71, or [Manna and Pnueli, 1995; Eisner et al., 2003; Fraser and Wotawa, 2006; Bauer et al., 2010]), or our finite $\cup$ infinite trace semantics, or infinite trace semantics by adding a self loop in s_b. Using finite trace semantics for

runtime [Biere et al., 1999] helps for some encodings (see first one in Subsec. 5.4.2) to detect too small bounds via counterexamples, as described below.

For the temporal logics $\mathcal{L}$ used for BMC (mainly $\mathcal{L} \subseteq$ Büchi), we have: $\forall \mathcal{S} \in \mathbb{S}_{Kripke,1,finite} \ \forall F \in \mathcal{L} \ \exists b \in \mathbb{N} : \ (\mathcal{S} \models F \Leftrightarrow \forall \pi \in paths_{max}(\mathcal{S}) : \pi \models_b F)$ (cf. Lemma 5.9). Therefore, $\forall \mathcal{S} \in \mathbb{S}_{Kripke,1,finite} \forall F \in \mathcal{L} \exists b \in \mathbb{N}$: BMC for $\mathcal{S} \models F$ is sound and complete. Such a bound b is called a **completeness threshold** ($\mathcal{CT}$) for $\mathcal{S}$ and F. Finding the smallest completeness threshold is as hard as the model checking problem itself, therefore approximations are used.

Many BMC tools (e.g., CBMC, LLBMC, cf. Chapter 7) do not approximate $\mathcal{CT}$, but instead lazily do a **bound check**, i.e., check on-the-fly whether the user-supplied bound b is sufficiently large to cover all of $\mathcal{S}$ relevant to F. Thus a BMC tool with a bound check always reports one of three possible results:

1. a counterexample $\pi \in paths(\mathcal{S})$ to F when π is within b;
2. a **counterexample $\pi \in paths(\mathcal{S})$ to b being sufficiently large**: a path π with $|\pi| = b$ that can be extended in $\mathcal{S}$ to π' with $|\pi'| \gneq b$ and relevant to F;
3. it is detected that b is sufficiently large to cover all of $\mathcal{S}$, and that $\mathcal{S} \models F$.

Successively, b can be increased until a shortest counterexample is found, or b has risen to a completeness threshold, i.e., become sufficiently large for BMC with a bound check to determine that there exists no counterexample of arbitrary length.

If BMC with a bound check is used to verify software, it is called **software bounded model checking** (**SBMC**). Usually, it uses Kripke structures $\mathcal{S} \in \mathbb{S}_{Kripke,(labeled),1,finite,<\omega}$ that directly specify the source code. Thus its states correspond to program states of the source code, the transitions to updates of the states (similarly to PROMELA, cf. Subsec. 3.4.3). Therefore, a state of $\mathcal{S}$ contains the value of the heap, the call stack, all registers, and a program counter; so a counterexample has the form of a concrete program execution.

The major BMC tools encode the problem in SAT (as described in Subsec. 5.4.2) or SMT (cf. Subsec. 3.3.3), for instance the SBMC tools CBMC and LLBMC (cf. Chapter 7). But alternatives that do not transform the problem to SAT or SMT also exist ($\text{DFS}_{\text{incremental}}$, cf. Subsec. 6.4.1, can be considered as such).

5.2.4. Finite versus Infinite

There are extensions to MC that deal with infinite state space, i.e., $\mathcal{S} \in \mathbb{S}_{Kripke,infinite}$, which are called **infinite state model checking** (**infinite model checking** for short). To be able to process the system, its relevant properties must still be representable finitely. For example, if the state space contains an unbounded integer domain for variable x, exhaustive model checking of a given property description is only possible if it does not need to consider infinite many properties on x, e.g., not all $\{x == i | i \in \mathbb{N}\}$, but only $\{x < 0, x == 0, x > 0\}$. Then Σ can be reduced to a finite set, and $S/\approx_\Sigma$ becomes finite, too. Thus, these abstractions are lossy (cf. Sec. 3.7); they can cause MC to no longer be complete, sound, or computable (e.g., fix-points in symbolic computations might diverge, cf. [To, 2010]).

5.2.5. Practical Relationship between CTL and LTL MC

Choosing the right temporal logics is crucial: It should be sufficiently expressive for the properties to be investigated, but if it is more powerful, the required MC algorithms become unnecessarily complex or even unfeasible. Furthermore, reductions (cf. Sec. 5.4) become harder and weaker. This subsection explains why this thesis focuses on the temporal logics LTL and Büchi.

To be able to find the sweet spot between expressiveness and complexity, practical criteria need to be considered, too: Both LTL and CTL are easy to understand, more efficient to model check than more complex temporal logics, easier for reductions like partial order reduction and abstractions, and many properties in practice are covered by both CTL and LTL – many of the rest still by either CTL or LTL. Thus we restrict our investigation in this section to these logics, similarly to [Holzmann, 2004; Vardi, 2001]. Table 5.1 below summarizes the comparison of CTL and LTL.

Their expressibilities are depicted in Fig. 4.3 on page 85. Compared to LTL, CTL can additionally express branching time properties, for instance that from a state, at least one execution has some property (like returning to an initial state, called **reset property**). These kind of specifications are often used for hardware. LTL, in contrast, can additionally express certain important liveness properties like **recurrence** properties (i.e., $\Box\Diamond$), e.g., fairness properties as given in Def. 5.6, or livelocks, i.e., that specific types of states must be visited infinitely often (see Lemma 4.53 and Chapter 6). These kind of properties are often required for software, in particular algorithms and protocols, e.g., $\Box\Diamond taken(a)$ that a scheduler is not **starving** object a, i.e., continuously denying a some necessary resource. They will occur throughout this thesis since its focus is on software verification.

Definition 5.6. Let $\Sigma_a = \{enabled(a), taken(a)\}$.

The liveness property of **weak fairness** (aka justice) describes that an action a that is continuously enabled must recurrently be taken. Formally, $\mathbf{fairness_{weak}(a)} := \{\mathcal{S} \in \mathbb{S}_{Kripke,linear} | \mathcal{S} \models \Diamond\Box enabled(a) \rightarrow \Box\Diamond taken(a)\}/\approx_{\Sigma_a}$.

The liveness property of **strong fairness** (aka compassion) describes that an action a that is recurrently enabled must recurrently be taken. Formally, $\mathbf{fairness_{strong}(a)} := \{\mathcal{S} \in \mathbb{S}_{Kripke,linear} | \mathcal{S} \models \Box\Diamond enabled(a) \rightarrow \Box\Diamond taken(a)\}/\approx_{\Sigma_a}$.

The worst case time and space complexities of CTL are $O(|\mathcal{S}| \cdot |F|)$, of LTL roughly $O(|\mathcal{S}_{\rightarrow^*}| \cdot 2^{|F|})$ (cf. Sec. 5.3). So theoretically, CTL model checking is faster than LTL model checking. In practice, however, LTL model checkers are often faster than CTL model checkers for properties specifiable in CTL∩LTL since

- LTL formulas are often small and simple in practice, i.e., the factor in $2^{O(|subF_{lin}(F)|)}$ is negligible;
- good converters from LTL to Büchi automata are for most relevant cases sub-exponential in $|subF_{lin}(F)|$;
- the dominating factor for the runtime is the number of states that are really visited by the MC algorithm, which strongly varies and depends on the formula being checked.

Thus it is often hard in practice to tell in advance whether a given property is more efficiently checked by a CTL or an LTL model checker.

Besides the aforementioned advantages of LTL over CTL in software verification, this thesis uses linear time behavioral properties described via LTL or Büchi automata because:

- model checking them can easily combine explicit state and symbolic methods (cf. Subsec. 5.4.2 and [Vardi, 2001]) and be performed on-the-fly (cf. Subsec. 5.3.2). Though on-the-fly MC is possible even for CTL* and μ-calculus, they are much more complex algorithms (cf. Sec. 5.3 and Sec. 5.5);
- software verification analyzes execution traces, which have linear time behavior, where LTL and Büchi automata are more expressive than CTL (cf. Fig. 4.3). Furthermore, MC can be implemented by language containment of transition systems (cf. Subsec. 5.3.2); hence verification engineers need only employ one conceptual model for verification [Vardi, 2001]). In contrast, branching time behavioral properties can lead to an "impedance mismatch" for execution traces of software, i.e., the branching and linear structures are not compatible, cannot be fully matched and thus can lead to unnatural or impossible transformations, e.g., for branching time behavioral properties that have no linear counterexamples. So the above is even more severe for lightweight formal methods that search for counterexamples (cf. Chapter 7) or that really execute the SUT, as MBT (cf. Part III) or runtime verification (cf. Sec. 2.2);
- MC via language containment is also compatible with the following techniques that are relevant for software verification: Firstly, with abstractions, such as reducing the state space (cf. Sec. 5.4). Secondly, with MBT's ioco relations (cf. Chapter 8). Finally, with composition of system specifications (cf. Subsec. 3.4.3 or [Manna and Pnueli, 1988]) and of property descriptions in an assume-guarantee methodology [Pnueli, 1985; Flanagan and Qadeer, 2003], where the system is decomposed into the components $C_1||\ldots||C_n$, and then a property P_i for component C_i is verified modularly, similar to design-by-contract.

In summary, Table 5.1 compares these aspects for CTL and LTL; the remainder of this thesis focuses on linear time properties, especially the temporal logics LTL (and Büchi).

Table 5.1.: Comparison of CTL and LTL

CTL	LTL
can express branching time properties	can express more complex linear time properties
worst case time and space complexities $O(\|\mathcal{S}\| \cdot \|F\|)$	worst case time and space complexities roughly $O(\|\mathcal{S}_{\rightarrow^*}\| \cdot 2^{\|F\|}$
mainly for hardware verification	mainly for software verification

5.3. Model Checking Algorithms

5.3.1. CTL

As this thesis focuses on linear time properties, this subsection only briefly describes CTL model checking algorithms.

To model check whether $\mathcal{S} \models F$ for $F \in$ CTL and $\mathcal{S} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke,finite}$, the following labeling algorithm can be performed: iterate inside out over the sub-formulas $f \in subF(F)$, i.e., over the syntactic structure of F, and label each $s \in S$ with f iff $(\mathcal{S}, s) \models f$. This can be decided using the sub-formulas processed already and a traversal algorithm on T: a backward traversal and decomposition into **strongly connected components** (**SCCs**); an SCC of $\mathcal{S}$ is a maximal sub-graph $\mathcal{S}_c = (S_c, T|_{S_c^2}, \Sigma, I)$ such that $\forall s, s' \in S_c : s \in dest(s', \rightarrow^*)$ and $s' \in dest(s, \rightarrow^*)$; for this decomposition, Tarjan's DFS [Tarjan, 1972]) is often used, which requires $O(|\mathcal{S}|)$ time and space to decompose $\mathcal{S}$. Finally, $\mathcal{S} \models F$ iff $\forall s \in S^0 : s$ contains the label F [Clarke et al., 1999b].

With each reachable state and its outgoing transitions processed up to $|subF(F)| \in O(|F|)$ times, the **worst case time** and **space complexities** of CTL model checking are in $O(|\mathcal{S}| \cdot |F|)$ [Clarke et al., 1999b; Schnoebelen, 2002; Baier and Katoen, 2008]. Better worst case space complexities can be achieved, but on the cost of the worst case time complexity [Schnoebelen, 2002]. For practical considerations, see Subsec. 5.2.5. NuSMV [URL:NuSMV] is one of the most prominent CTL model checkers.

5.3.2. On-the-fly LTL

Two main approaches exist for LTL MC: Firstly, offline LTL MC, for instance tableau-based, similar to CTL MC [Clarke et al., 1997, 1999b]. It can be used for symbolic MC and if global MC is necessary (cf. Subsec. 5.3.3). Secondly, on-the-fly LTL MC, which is a more direct and simpler algorithm, more efficient (see Subsec. 5.2.2 and time complexity paragraph below) and more popular [Vardi and Wolper, 1986]. Thus this thesis focuses on on-the-fly LTL MC.

Fig. 5.1 describes the work-flow of on-the-fly LTL MC: When a desired property description $F \in$ LTL is given, we can use Def. 4.44 (or more efficient translations [Babiak et al., 2012]) to construct a Büchi automaton $\mathcal{A}_{never} \equiv \mathcal{A}_{\neg F} \equiv \neg F$. Alternatively, an undesired property description can be given as Büchi automaton $\mathcal{A}_{never}$ directly, which is more expressive (cf. Theorem 4.46).

For composition with $\mathcal{A}_{never}$, we transform the given system specification $\mathcal{S} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke,1,finite}$ into a Büchi automaton $\mathcal{A}_{\mathcal{S}} := K_{fin}2FSM(\mathcal{S})$ (cf. Def. 3.28). $\mathcal{A}_{\mathcal{S}}$ embodies the same linear Kripke structures as $\mathcal{S}$, as Lemma 5.7 shows. Alternatively to operating on Büchi automata, all operations could be performed on Kripke structures as well (cf. Def. 3.27), but using Büchi automata is the standard approach. Since $\mathcal{A}_{\mathcal{S}}$ only has accepting states, the conjunctive Büchi automaton $\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{never}$ can be constructed in the simplified way (cf. Note 4.34), which can be performed on-the-fly. This often leads to checks where only parts of the state space are constructed (cf. Subsec. 5.2.2). Furthermore, as Subsec. 4.3.2 has described for the conjunctive Büchi automaton, the property being checked can help prune the state space to its relevant part, such that late or no fault detection often still yields a smaller state space than offline MC (cf. Subsec. 5.5.1). Lemma 5.8 shows that on-the-fly LTL MC can now be implemented by

an emptiness check of $Models_{lin}(\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{never})$, for instance via the nested DFS algorithm described below. If an undesired property occurs in $\mathcal{S}$, a witness can be given (i.e., a counterexample to F).

Lemma 5.7. *Let $\mathcal{S} \in \mathbb{S}_{Kripke}$. Then*

$$Models_{lin}(K_{fin}2FSM(\mathcal{S})) = \{\mathcal{S}_\pi \in \mathbb{S}_{Kripke,linear} | \pi \in paths_{max}(\mathcal{S})\}.$$

Proof. For $\mathcal{S} \in \mathbb{S}_{Kripke}$, $Models_{lin}(K_{fin}2FSM(\mathcal{S})) = \{\mathcal{S}_{lin} \in \mathbb{S}_{Kripke,linear} | \exists \pi \in paths_{max}($ $K_{fin}2FSM(\mathcal{S})) : K_{fin}2FSM(\mathcal{S})$ accepts π and both $\mathcal{S}_{lin}$ and π describe the same linear behavior (cf. Def. 4.30) $\} = \{\mathcal{S}_\pi \in \mathbb{S}_{Kripke,linear} | \pi \in paths_{max}(\mathcal{S})\}$ since $K_{fin}2FSM(\mathcal{S})$ only has accepting states and exactly embeds $\mathcal{S}_{lin}$'s paths, encoding $I(\cdot, s_i)$ via labels. □

Lemma 5.8. *Let $\mathcal{S} \in \mathbb{S}_{Kripke,1,finite}$ and $\mathcal{A}_{never} \in$ Büchi (or $F \in$ LTL with $\mathcal{A}_{never} \equiv \mathcal{A}_{\neg F}$) be given. Then $\mathcal{S} \models \mathcal{A}^c_{never}$ (or $\mathcal{S} \models F$) $\Leftrightarrow$ $Models_{lin}(\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{never}) = \varnothing$.*

Proof. $\mathcal{S} \models \mathcal{A}^c_{never}$ (or $\mathcal{S} \models F$)

$\overset{\text{Def. 4.31}}{\Longleftrightarrow}$ $\forall \pi \in paths_{max}(\mathcal{S}) : (\mathcal{S}_\pi, init) \models \mathcal{A}^c_{never}$

$\overset{\text{Lemma 5.7}}{\Longleftrightarrow}$ $Models_{lin}(\mathcal{A}_{\mathcal{S}}) \subseteq Models_{lin}(\mathcal{A}^c_{never})$

(called **language containment check**)

$\Longleftrightarrow$ $Models_{lin}(\mathcal{A}_{\mathcal{S}}) \cap (\mathbb{S}_{Kripke,linear} \setminus Models_{lin}(\mathcal{A}^c_{never})) = \varnothing$

$\overset{Lemma\ 4.38}{\Longleftrightarrow}$ $Models_{lin}(\mathcal{A}_{\mathcal{S}}) \cap Models_{lin}(\mathcal{A}_{never}) = \varnothing$

$\overset{Lemma\ 4.35}{\Longleftrightarrow}$ $Models_{lin}(\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{never}) = \varnothing$ (called **emptiness check**)

□

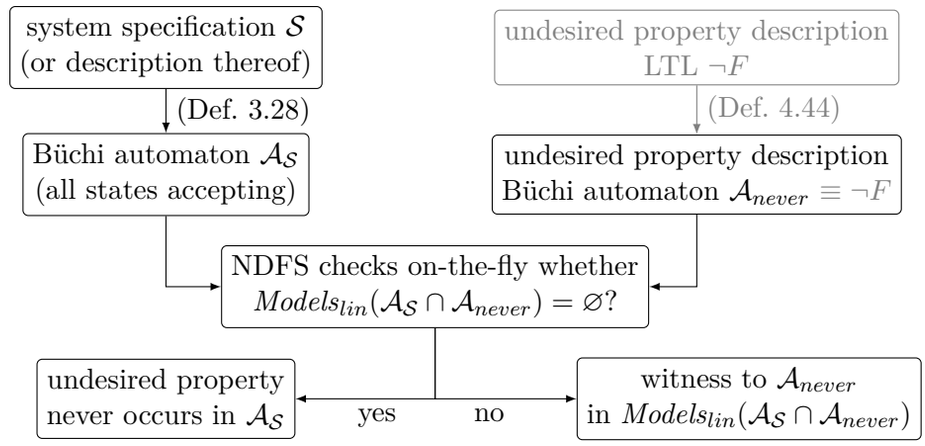


Figure 5.1.: Overview of on-the-fly LTL or Büchi MC

Nested DFS. Since $Models_{lin}(\mathcal{A}) \neq \varnothing$ iff $\exists \pi \in paths_{max}(\mathcal{A})$ that is accepting (using Def. 4.29 and $\mathcal{S}_\pi$), emptiness checking of $Models_{lin}(\mathcal{A})$ is equivalent to finding an accepting path $\pi \in paths_{max}(\mathcal{A})$. As $\mathcal{S} \in \mathbb{S}_{Kripke,1,finite}$, the conjunctive Büchi automaton $\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{never}$ is also finite. Because of Lemma 5.9, emptiness checking is reducible to the graph problem of **acceptance cycle** checks, i.e., to searching for a reachable cycle in $\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{never}$ that contains a state $s \in F$. This is sufficient if end states are forbidden. Otherwise, the case $\pi \in paths_{max}(\mathcal{A})$ with $|\pi| \in \mathbb{N}$ applies in Lemma 5.9, and we must additionally search for a reachable accepting end state.

Lemma 5.9. *Let $\mathcal{A} \in$ Büchi. Then $Models_{lin}(\mathcal{A}) \neq \varnothing \Leftrightarrow$*
$\exists \pi \in paths_{max}(\mathcal{A}) : \mathcal{S}_\pi \in Models_{lin}(\mathcal{A})$ *with π being a lasso or $|\pi| \in \mathbb{N}$.*

Proof. If $\mathcal{S} \in Models_{lin}(\mathcal{A})$, then $\pi_\mathcal{S}$ is accepted by $\mathcal{A}$, i.e., ends in an accepting end state or visits an acceptance state a infinitely often. If $|\pi| \in \mathbb{N}$, $\mathcal{S}_\pi \in Models_{lin}(\mathcal{A})$. Otherwise, let $\pi_\mathcal{S} = \pi_{prefix} \cdot \pi_{middle} \cdot \pi_{suffix}$ with $source(\pi_{middle}) = dest(\pi_{middle}) = a$. Then $\mathcal{A}$ accepts the lasso $\pi' := \pi_{prefix} \cdot (\pi_{middle})^\omega$, thus $\mathcal{S}_{\pi'} \in Models_{lin}(\mathcal{A})$. □

Note 5.10. The proof is valid for temporal logics that specify linear time behavioral properties that are ω-regular; then $\mathcal{A}$ really accepts the lasso. This is the case for all temporal logics we consider [Fisler et al., 2001; Ehlers, 2011]. Exemplary temporal logics that go beyond ω-regularity use non-regular automata as temporal operators (just as ECTL* used Büchi automata), for instance visibly push-down automata to enable counting [Demri and Gastin, 2012]. But this leads to undecidable model checking (and satisfiability checking), e.g., already for one operator based on the context-free language $\{a_1^k, a_2, a_3^k, a_4 | k \in \mathbb{N}\}$ [Demri and Gastin, 2012].

Acceptance cycle detection can be performed on-the-fly by the **nested depth-first search** (**NDFS**) algorithm [Courcoubetis et al., 1992], implemented amongst others in SPIN [Holzmann et al., 1996] and LTSMIN [Evangelista et al., 2012] (cf. Sec. 5.5). As depicted in Listing 5.3, NDFS is based on the **basic depth-first search** (**DFS**), whose worst case time complexity is in $O(\mathcal{S}_{\to^*})$ since it traverses $\mathcal{S}$, its worst case space complexity is in $O(|S_{\to^*}|)$ since transitions need not be stored. NDFS extends the DFS with the statement **if** accepting(s) **then** innerDFS(s) **fi**; in line 7: Before the basic DFS backtracks from an accepting state s and removes it from the stack, a second, inner DFS routine **innerDFS()** is started, to check whether s can reach itself by reaching the path from *init* to s (stored on the stack), thus resulting in an acceptance cycle.

The basic DFS models the trace from *init* to the currently visited state via a stack, used for backtracking. To avoid redundant work of re-exploring states already visited (those on the stack as well as those that have already been backtracked from), they need to be stored and looked up efficiently, which is usually done with a hash table. Avoiding revisits is essential, since the Kripke structures of typical specifications have a lot of states with many incoming transitions. Avoiding revisiting states can hence lead to an exponential speedup, for instance in leader election algorithms (cf. Subsec. 6.8 and [Faragó, 2007, Table 6.3], which shows a speedup from 176 hours to 70 seconds).

Also innerDFS() should avoid revisiting states, since a state can be reachable from many acceptance states (which would lead to worst case time complexity quadratic in the number of states). Therefore, innerDFS() uses a global hash table, too. The same hash table as for the outer DFS can be used by adding two bit to the state vector to indicate whether a state has been visited by the outer DFS and by the inner DFS. Immediately after DFS(s) has backtracked from all direct successors $dest(s, \to)$ of s, we have for all acceptance states $t \in dest(s, \to)$:

- either t is on the stack, in which case innerDFS(t) will be called later;
- or innerDFS(t) has already been called, terminated and thus inductively excluded that t is on an acceptance cycle.

So for all t with $\{t, 0\} \in$ hash table, either no acceptance cycle is reachable from t or for each path π from t to an acceptance state of a reachable acceptance cycle, a state from

```
proc DFS(S_→* s)                                                    1
  add {s,0} to hash table;                                          2
  push s onto stack;                                                3
  for each successor t of s do                                      4
    if {t,0} ∉ hash table then DFS(t) fi                            5
  od;                                                               6
  if accepting(s) then innerDFS(s) fi;                              7
  pop s from stack;                                                 8
end                                                                 9
                                                                    10
proc innerDFS(S_→* s) /* the nested search */                       11
  add {s,1} to hash table;                                          12
  for each successor t of s do                                      13
    if {t,1} ∉ hash table                                           14
    then innerDFS(t)                                                15
    else if t ∈ stack then report cycle fi                          16
    fi                                                              17
  od;                                                               18
end                                                                 19
```

Listing 5.3: Nested DFS

π is on the stack. Therefore NDFS eventually finds an acceptance cycle if there exists one. More detailed proofs can be found in [Holzmann, 2004; Baier and Katoen, 2008].

Notes 5.11. Depending on how $\mathcal{S}$ is represented (cf. Subsec. 3.4.3), a basic DFS might require additional space for bookkeeping of the outgoing transitions that are iterated over for each state on the stack. Exemplary worst case space complexities, depending on $\mathcal{S}$'s representation and the implementation of the DFS, are in $O(S_{\to^*} + depth_{\mathcal{S}} \cdot log(branch_{\mathcal{S}}))$ or in $O(S_{\to^*} \cdot log(S_{\to^*}))$, but of course always in $O(\mathcal{S}_{\to^*})$.

The original NDFS implementation in SPIN [Holzmann and Peled, 1994] performed a simpler check for the states visited in innerDFS(): Instead of checking **if** $t \in$ stack (cf. line 16 of Listing 5.3), it only checked whether t equals the accepting state that initiated innerDFS(). Holzmann et al. showed in [1996] that checking within the complete stack (i.e., **if** $t \in$ stack) instead of only with the seed is required for correctness of the NDFS algorithm in combination with partial order reduction (see Subsec. 5.4.1). Checking within the complete stack also has better overall performance since the NDFS algorithm aborts earlier and hence avoids traversing possibly further, large parts of the graph.

The NDFS is not on-the-fly in a strict sense: only after the outer $DFS(s)$ for an acceptance state s has explored all new reachable states in $dest(s, \to^*)$, innerDFS(s) searches for an acceptance cycle containing s. In the extreme case of $s = init$ being the only accepting state, the full state space is explored before acceptance cycles are searched for, so NDFS is fully offline. More generally, if there are multiple accepting states, but all at a shallow depth in the state space, still a large part of the state space is explored before acceptance cycles are searched for. For the LTL subclass of livelock detection, Subsec. 6.3.2 shows a concrete walk-through of the described work-flow and NDFS. Chapter 6 then covers a better algorithm that is on-the-fly in the strict sense.

For finite $\cup$ infinite trace semantics, l.7 of Listing 5.3 must additionally check if s is

an accepting end state, in which case the finite path described by the stack can simply be reported.

Time and Space Complexities of NDFS. Since a NDFS algorithm over $\mathcal{S} = (S, T, \Sigma, I)$ visits each reachable state at most twice, its worst case time complexity is $O(|\mathcal{S}_{\to^*}|)$. Since only two bit per state are required to indicate whether a state has yet been visited by the outer DFS and by the inner DFS, the worst case space complexity is the same as for the DFS: in $O(|S_{\to^*}|)$ (see also Note 5.11).

Notes. Tarjan's DFS [Tarjan, 1972] could be used instead of the NDFS to detect all SCCs in $\mathcal{S}_{\to^*}$ and then check whether they contain an accepting state. It has the same worst case time and space complexity as the NDFS, but the factors are higher, and some transitions between states need to be stored. It is able to produce all accepting paths, though, and to implement also strong fairness.

Time and Space Complexities of LTL MC. When $F \in \text{LTL}$ is given and translated to a Büchi automaton $\mathcal{A}_{\neg F}$ (cf. Def. 4.44), $\mathcal{A}_{\neg F}$'s states are subsets of $subF(F)$, so $\mathcal{A}_{\neg F}$ has $O(2^{|subF(F)|}) = O(2^{|F|})$ states in the worst case. Thus $\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{\neg F}$ has $O(|S| \cdot 2^{|F|})$ states and $O(|T| \cdot 2^{|F|})$ transitions in the worst case.

In total, on-the-fly LTL MC has a **worst case time complexity** in $O(|\mathcal{S}_{\to^*}| \cdot 2^{|F|})$ [Baier and Katoen, 2008], and a **worst case space complexity** in $O(|S_{\to^*}| \cdot 2^{|F|})$. Contrarily, global LTL MC has a **worst case time complexity** of $|\mathcal{S}| \cdot 2^{O(|F|)}$ [Clarke et al., 1999b; Schnoebelen, 2002]. For practical considerations, see Subsec. 5.2.5.

SPIN [URL:Spin], LTSmin [URL:LTSmin] and DiVinE [URL:DIVINE] are prominent on-the-fly LTL model checkers (cf. Sec. 5.5), SAL [URL:SAL] and SMV [URL:SMV] offline tableau-based global LTL model checkers [Clarke et al., 1997].

5.3.3. CTL*

As Corollary 4.50 shows, CTL* cannot express stronger linear time properties than LTL, but Büchi can. As this thesis focuses on linear time properties, this subsection only briefly describes CTL* model checking algorithms.

The labeling algorithm for CTL MC can be lifted to CTL* by treating sub-formulas f that are not in CTL as LTL formulas and using a global LTL MC (e.g., tableau based) to label all $s \in S$ with f iff $(\mathcal{S}, s) \models f$. The complexity of CTL* model checking is the maximum of the complexity of CTL MC and global LTL MC, $|\mathcal{S}| \cdot 2^{O(|L|)}$ [Clarke et al., 1999b]. ARC [URL:ARC] and LTSmin [URL:LTSmin] (via μ-calculus) are CTL* model checkers (cf. Sec. 5.5).

Note. Explicit state, on-the-fly MC, as described in the previous subsection, can also be lifted from LTL to CTL* model checking: [Visser and Barringer, 2000] shows how SPIN could use hesitant alternating automata for this instead of Büchi automata, and non-emptiness games instead of the NDFS. But this is much more complex than Büchi MC and has therefore not been implemented in SPIN.

5.4. Reductions

Because of state space explosion, MC quickly becomes infeasible. To increase the maximal problem size that can still be model checked, this section covers the possibilities of reducing space requirements of MC by abstraction: either lossy by reducing the state space, or lossless by compressing it. Since these reductions require computations, the total runtime can increase, for instance for SPIN's minimized automata (cf. Subsec. 5.5.1). But for strong lossy abstractions, considering fewer states can strongly outweigh the additional computations, resulting in drastically reduced total runtime. This can be the case even if the reduction sustains completeness, e.g., for partial order reduction (cf. Subsec. 5.4.1 and Table 6.8).

Although manual reductions, like problem specific abstractions, can yield large improvements, they are highly individual and therefore not covered in this thesis, which only considers fully automatic methods. Many reductions abstract from the original state space, which needs to preserve the relevant behaviors for the property under investigation, otherwise the reduction would not be sound and complete. For MC, this thesis focuses on sound and complete techniques. Examples of sound and complete abstractions are certain **abstract interpretations**, i.e., abstractions from the concrete system specification description by subsuming or eliminating values of parts of the state vector. Examples are selective data hiding via program slicing (cf. Subsec. 5.4.3), whereas general data type abstraction can become unsound or incomplete, e.g., CBMC's (cf. Chapter 7) **predicate abstraction** [Clarke et al., 1986], which maps larger data to a set of Boolean variables by only keeping track of certain relevant predicates over the original variables, which are discarded.

Roadmap. For lossy abstractions, one of the most effective and prominent **state space reduction techniques** is partial order reduction, introduced in Subsec. 5.4.1 (and also taken into account in Subsec. 6.6). Other techniques like symmetry reduction, program slicing and statement merging will be covered briefly by Subsec. 5.4.3.

For lossless abstractions, one of the most compact and prominent representations of the state space is by **symbolic techniques**, covered in Subsec. 5.4.2: Firstly SAT- and SMT-based, then BDD-based (cf. Chapter 3). Other symbolic techniques for compression, like minimized automata and ETF, and non-symbolic techniques like tree compression and the incomplete bitstate hashing and hash compaction, will be covered briefly in Subsec. 5.4.3.

Notes. For MC hardware, BDD-based techniques perform well (since bit-vectors and synchronizing clocks are heavily used). For MC software, partial order reduction often yields the strongest reductions, especially when complex data structures and many asynchronously executing processes are involved [Holzmann, 2004].

The strength of most of these reduction methods have been considered and compared roughly in [Faragó, 2007; Laarman, 2014; Barnat, 2015]. Most reduction methods can be combined (cf. Subsec. 5.5.2), but their factors of improvement often do not multiply. Thus, combinations are often researched, e.g., partial order reduction in combination with symmetry reduction in [Bosnacki et al., 2002] and partial order reduction in combination with parallelization, hash compaction and tree compression in Subsec. 6.8.

5.4.1. Partial Order Reduction

Covering all execution orders of the system's component automata is the cause for state space explosion (cf. Sec. 5.1). But often statements of concurrent processes are commutative, i.e., their execution orders lead to the same state (see Def. 5.13). **Partial order reduction** (**POR**) does not consider all possible interleavings of concurrent statements, only a subset relevant to the currently checked property F. This often reduces the size of the state space exponentially, and is hence one of the most powerful reduction methods in MC linear time properties.

So POR (more precisely, the dependency relation D below) requires information about which component automaton provides the update of a transition. This information can be deduced from transitions labeled with basic statements, which cause deterministic updates (cf. page 47). Hence POR operates on $\mathbb{S}_{\mathit{Kripke,labeled,finite,deterministic,1}}$.

All variants of partial order reduction choose a subset of $\mathit{enabled}(s)$ in each reachable state s to select only few of the interleavings that have the same effect with respect to the checked property F [Peled, 1993]. This thesis uses the technique of ample sets [Clarke et al., 1999a,b], which is implemented in the model checker SPIN [Holzmann et al., 1996]. Similar variants are: the stubborn set method [Valmari, 1989], implemented in the model checker LTSMIN [Pater, 2011]; the persistent set and sleep set methods [Godefroid, 1996], which all weaken the provisos C0 and C1 of Table 5.2; the cartesian POR [Gueta et al., 2007], which is fully dynamic, i.e., needs no static approximations; the probe set method [Kastenberg and Rensink, 2008], which is a more flexible dynamic POR; the guard-based method [Laarman et al., 2013a], which is a language-agnostic stubborn set method that uses guards as enabling as well as disabling conditions; the peephole method [Wang et al., 2008], which allows POR of implicit symbolic MC based on SAT- and SMT-solving; and monotonic POR [Kahlon et al., 2009], which is an optimal POR for implicit symbolic MC based on SAT- and SMT-solving and can also be used for explicit MC. There are also methods that stem from POR but differ more (in the method or field) and are no longer called POR. For instance confluence reduction [Timmer et al., 2011] for Markov decision processes (cf. Subsec. 5.5.4), and statement merging (cf. Subsec. 5.4.3).

The ample set technique chooses in each reachable state $s \in S_{\to^*}$ not the whole set $\mathit{enabled}(s)$, but the set $\mathit{ample}(s) \subseteq \mathit{enabled}(s)$, called the **ample set** of s. If $\mathit{ample}(s) = \mathit{enabled}(s)$, we say s is **fully expanded**. $\mathcal{S}_{\to^*}$ is reduced to $\mathcal{S}_{\mathrm{POR}}$, as defined in Def. 5.12.

Definition 5.12. Let $\mathcal{S} = (S, \mathfrak{T}, L, \Sigma, I) \in \mathbb{S}_{\mathit{Kripke,labeled,finite,deterministic,1}}$.
Then
- $\boldsymbol{\mathfrak{T}_{\mathbf{POR}}} := \{(s, \alpha, s') \in \mathfrak{T} \mid \alpha \in \mathit{ample}(s)\}$;
- $s \xrightarrow{\alpha}_{\mathbf{POR}} s' :\Leftrightarrow (s, \alpha, s') \in \mathfrak{T}_{\mathrm{POR}}$;
- $\boldsymbol{S_{\mathbf{POR}}} := \mathit{dest}(\mathit{init}, \longrightarrow^*_{\mathrm{POR}})$;
- $\boldsymbol{\mathcal{S}_{\mathbf{POR}}} := (S_{\mathrm{POR}}, \mathfrak{T}_{\mathrm{POR}}, L, \Sigma, I)$.

For model checking to retain completeness, Table 5.2 lists various provisos on $\mathit{ample}(\cdot)$. They do not fully determine $\mathit{ample}(\cdot)$, so any subset of $\mathfrak{T}$ satisfying the provisos retains completeness. Depending on what kind of property is being checked, not all provisos have to hold, though: For **deadlock detection**, $\mathit{ample}(\cdot)$ only needs to fulfill the emptiness proviso C0, which forbids invalid end states, and the dependency proviso C1 (also called **ample decomposition proviso**), which forbids ignoring depending transitions. To check C1, the ample set selection has to determine which statements are independent, as

defined in Def. 5.13. Model checkers like SPIN and LTSMIN (cf. Sec. 5.5) conservatively approximate this dependency by static analysis, resulting in the **dependency relation** $D \subseteq L \times L$ [Katz and Peled, 1988]. By using D, C1 can be checked locally.

Definition 5.13. Let $(S, \mathfrak{T}, L, \Sigma, I) \in \mathbb{S}_{Kripke,labeled,finite,deterministic,1}$ and statements $\alpha, \beta \in L$.

Then α, β are **independent** iff $\forall s \in S : \alpha, \beta \in enabled(s) \implies$
- $\alpha \in enabled(\beta(s))$ and $\beta \in enabled(\alpha(s))$ (**enabledness**)
- and $\alpha(\beta(s)) = \beta(\alpha(s))$ (**commutativity**)

α, β are **dependent** $:\Leftrightarrow \alpha, \beta$ are not independent, in which case $(\alpha, \beta) \in D$.

To check a formula $F \in \mathrm{LTL}_{-X}$, the remaining two provisos also need to hold: The visibility proviso C2 (also called **invisibility proviso**) guarantees that transition from not fully expanded states are invisible. So as long as transitions from not fully expanded states are taken, all propositions relevant to F stutter, i.e., $\mathcal{S}$ and $\mathcal{S}_{\mathrm{POR}}$ are stuttering equivalent related to F. So C2 guarantees that transitions from not fully expanded states do not influence whether F holds (cf. Lemma 4.27), and C1 guarantees that the visible transitions can still be taken afterwards. Consequently, POR is correct only for formulas in LTL_{-X}, but this is not a severe problem: as the LTS contains all possible interleavings, very little is known about the next state, so the next operator is rarely used. Finally, the cycleClosing proviso C3 (also called **cycle proviso**) prevents the so-called **ignoring problem**, i.e., that some transition is postponed indefinitely [Evangelista and Pajault, 2010]. C3 considers all cycles that the current state s is on, which is too costly to enforce. Thus the notInStack proviso C3' (also called **cycle implementation proviso**) is used: It implies C3, but can be checked without considering all those cycles. Since it enforces full expansion of a state on the cycle, it is stronger than C3. Since C3' checks the stack, i.e., it depends on the path leading to s, it is not a local proviso. Consequently, C3' complicates MC: Considering the full stack impedes parallelization (cf. Sec. 6.7), and additional constraints and bookkeeping is required to ensure identical $ample(s)$ for NDFS's (outer) DFS(s) and innerDFS(s). These additional constraints can strongly weaken POR [Holzmann et al., 1996].

Since all provisos C0, C1 via D, C2 and C3' can be checked on-the-fly, these provisos can be used for on-the-fly LTL_{-X} MC with POR [Peled, 1994].

Notes 5.14. Besides the different POR versions mentioned above, many minor variations exist, for instance **transparent POR** [Siegel, 2012], where C2 is relaxed to $\mathbf{C2}^{transparent}$: for a literal l that does not occur negated in F (e.g., for $l = \neg p$, F does not contain p without negation), α is visible iff α swaps l from **false** to **true**. Chapter 6 shows further variations.

POR can easily be applied for our finite $\cup$ infinite trace semantics (cf. page 71, just as for finite trace semantics [Baier and Katoen, 2008]): C0 guarantees that $\mathcal{S}$ and $\mathcal{S}_{\mathrm{POR}}$ have the same end states, and all other provisos do not conflict with end states.

5.4.2. Symbolic Techniques

Symbolic techniques [Clarke et al., 1999b, Chapter 6]) use formulas (or representations thereof) to encode states and transitions: all $s \in S$ can be encoded by Σ_{sv}, T by a

Table 5.2.: Provisos on $ample(s)$

emptiness (**C0**)	$\forall s \in S_{\text{POR}} : (ample(s) = \varnothing \Leftrightarrow enabled(s) = \varnothing)$.
dependency (**C1**)	$\forall s \in S_{\text{POR}}\ \forall \alpha \in L \setminus ample(s) : \ (\exists \beta \in ample(s) : (\alpha, \beta) \in D) \implies \forall \pi \in paths(\mathcal{S}, s)$: if α occurs on π, some statement in $ample(s)$ occurs earlier on π.
visibility (**C2**)	$\forall s \in S_{\text{POR}} : ample(s) \neq enabled(s) \implies \forall \alpha \in ample(s) : \alpha$ is **invisible** to F (α does not swap any $p \in \Sigma$ that occur in F).
cycleClosing (**C3**)	$\forall s \in S_{\text{POR}}\ \forall$ cycles $\pi \in paths(\mathcal{S}_{\text{POR}}, s)\ \forall \alpha \in enabled(s)\ \exists s'$ on $\pi : \alpha \in ample(s')$.
notInStack (**C3'**)	$\forall s \in S_{\text{POR}}\ (\exists \alpha \in ample(s) :$ $\alpha(s)$ is in the DFS stack while s is the last element on the stack$) \implies ample(s) = enabled(s)$.

propositional formula representing the function $f_T : \Sigma_{sv}^2 \to \mathbb{B}, (I_{sv}(s), I_{sv}(s')) \mapsto (s \to s')$, for $s, s' \in S$. Then MC does not operate on Kripke structures and sets of states, but on the formulas. Two main approaches are described below: the BDD-based approach, using BDD-like data structures as representations for propositional formulas, and the SAT- and SMT-based approach, using propositional formulas (plus optional theories) directly.

BDD-based

BDD-like data structures are mainly used for implicit state model checking of branching time logics and in hardware verification. Since this thesis focuses on linear time logics and software verification, BDD-based techniques are only described briefly.

Instead of explicitly traversing the state space, **symbolic implicit state model checking via BDDs** performs operations on BDDs, as described in Subsec. 3.2.4. For this, a Kripke structure is represented by a BDD encoding of f_T, called BDD_T. Furthermore, a temporal logic operator can be described as a **least** or **greatest fix-point** of a function f, i.e., a least or greatest set $S' \subseteq S$ such that $f(S') = S'$ [Emerson and Clarke, 1980]. The function f can be expressed using only the operators from propositional logic, AX and EX. For instance, similar to Def. 4.44, the CTL operator A (b_1 U b_2) (with branching time formulas b_i) can be described as $\mu Z.b_2 \vee (b_1 \wedge \text{AX } Z)$, i.e., as the least fix-point of Z for the function $f := b_2 \vee (b_1 \wedge \text{AX } Z)$. Thus a smallest set $Z \subseteq S$ such that $f(Z) = Z$ contains the states s with $(\mathcal{S}, s) \models$ A (b_1 U b_2).

Since all these functions f are monotonic and S is finite, their least fix-point is $\mu Z.f(Z) = \bigcup_{i \in \mathbb{N}} f^i(\varnothing)$, their greatest fix-point is $\nu Z.f(Z) = \bigcap_{i \in \mathbb{N}} f^i(S)$ [Clarke et al., 1999b]. Using BDD representations and operations, $f(Z)$ can be computed from Z and BDD_T using the operations described by f. So by performing at most $|S|$ iterations of those BDD-based operations, a fix-point BDD emerges that represents the states satisfying the property description. To construct counterexamples in implicit state model checking, extra work has to be performed [Clarke et al., 1999b].

BDD encodings are also applicable for implicit state LTL MC by using offline tableau-based LTL MC. Symbolic techniques can also be used for explicit state model checking by

replacing the hash table used for the set of visited states with a symbolic representation. Examples are LTSMIN's ETF and SPIN's minimized automata (cf. Sec. 5.5).

SAT- and SMT-based

Reducing MC to a SAT or SMT problem (cf. Chapter 3) is mainly used if a linear time property description L needs to be checked. The most prominent approaches use bounded model checking (cf. Subsec. 5.2.3). Many propositional encodings $[[\boldsymbol{\mathcal{S}, L}]]^{\boldsymbol{b}}$ for BMC of $\mathcal{S} \models L$ with a user-supplied bound b do not use fix-point operations on some function f (see BDD-based technique above), but only unwind f and the transition relation T for b times [Biere et al., 1999]. In more detail, the encodings can take on many forms [Kroening et al., 2011]:

All encodings need to reason about paths $\pi = (s_i)_{i \in [0,\ldots,b]} \in paths(\mathcal{S})$ of length b, for which versioned Σ_{sv} (cf. SSA in Sec. 7.1) encode the states on π (and thus also encode I_{sv}). Propositional formulas ensure:

- that π starts in S^0, via formula $\boldsymbol{F_{S^0}}$;
- that π is an unwinding of T of length b, via formula $\boldsymbol{F_T^b}$;
- that π is a lasso with a cycle of length $i \in \mathbb{N}_{\geq 0}$, via $\boldsymbol{F_i^b}$. Since $\mathcal{S}$ is finite, searching for a lasso is sufficient, analogously to Lemma 5.9.

For a property $L \in$ LTL, several encodings have been published [Clarke et al., 2005]: The cases where π of length b with a cycle of length i fulfill L can be encoded as propositional formula $\boldsymbol{F_{L,i}^b}$ directly, making use of the encodings $F_{L',i}^b$ for $L' \in subF(L)$ [Biere et al., 1999], similar to Subsec. 5.3.1. With this, $[[\mathcal{S}, L]]^b = F_{S^0} \wedge F_T^b \wedge \big(\vee_{i \in [0,\ldots,b]} (F_i^b \wedge F_{\neg L,i}^b)\big)$. More complex encodings distinguish finite and infinite witnesses to achieve lower completeness thresholds for finite witnesses; this was the original encoding in [Biere et al., 1999]. Alternatively, for a property $\mathcal{A}_{never} \in$ Büchi, the Büchi automaton $\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{never}$ (cf. Subsec. 5.3.2) can be propositionally encoded similarly to how $\mathcal{S}$ was encoded [Clarke et al., 2004b, 2005]). Then the formulas $F_{L,i}^b$ check (similar to $L = \square\lozenge$ accept) whether the cycles corresponding to F_i^b contain accepting states.

$[[\mathcal{S}, L]]^b$ is checked for satisfiability using a SAT-solver. If a propositional model exists, it is a counterexample to L (a witness for $\mathcal{A}_{never}$) that has length $\leq b$.

Let X be some structure, $C(X)$ the size of a circuit defining X, and $\boldsymbol{\Sigma_{sv}(X)}$ a propositional signature required for X (so usually $|\Sigma_{sv}(X)| \in O(log(|X|))$. Then the worst case size of the original encoding is $O(b \cdot C(\mathcal{S}) + b^3 \cdot |L|)$, with the worst case number of propositional variables in $O(b \cdot |\Sigma_{sv}(\mathcal{S})| + (b+1)^2 \cdot |L|)$ [Clarke et al., 2004b, 2005]. The worst case size of the simpler encoding for $(Q, \Delta, A, Q^0, F) = \mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{never}$ is in $O\big(b \cdot (C(\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{never}) + |\Sigma_{sv}(\mathcal{S})| + |\Sigma_{sv}(L)| + C(F))\big)$, with the worst case number of propositional variables in $O(b \cdot (|\Sigma_{sv}(\mathcal{S})| + |\Sigma_{sv}(L)|))$ [Clarke et al., 2004b, 2005]. Many other and more efficient encodings are possible, e.g., by not simply unwinding T but using fix point encodings [Latvala et al., 2004], or by encoding into SMT instead of SAT [Armando et al., 2009].

If no models exist for the Büchi automaton encoding up to the number b of states in $\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{never} + 1$, i.e., $b \in O(|Q|)$, there exists no counterexample of any length, i.e., L holds [Clarke et al., 2004b]. Thus a completeness threshold $\mathcal{CT}$ is in $O(|S| \cdot 2^{|L|})$, with S being the set of states of $\mathcal{S}$ and $\neg L$ the LTL formula transformed into $\mathcal{A}_{never}$. There are lower estimates for $\mathcal{CT}$, but finding the smallest completeness threshold is as hard

as the model checking problem itself [Kroening et al., 2011].

Since $\mathcal{CT}$ is in $O(2^{|\Sigma_{sv}(\mathcal{S})|})$, an encoding has a worst case number of propositional variables in $O(2^{|\Sigma_{sv}(\mathcal{S})|})$. In summary, SAT-based BMC has a worst case time complexity in $O(2^{(2^{|\Sigma_{sv}(\mathcal{S})|})})$. In practice, the runtime is often much better and can be lower than for other LTL MC approaches, and the encodings are often smaller than BDD encodings [Clarke et al., 2004b, 2005].

BMC can check ACTL* in the same way. Similarly, BMC of safety properties for system specification descriptions in the programming language C is possible (cf. Chapter 7).

5.4.3. Other Reductions

This subsection covers reductions that are usually less powerful and thus less prominent.

State space reductions besides partial order reduction can also be performed by symmetry reduction, program slicing and statement merging:

Symmetry reduction identifies global states that are equivalent for the property P being checked, due to some given symmetry of P [Baier and Katoen, 2008]. Two types of symmetry occur frequently: **full symmetry** between all n processes, i.e., arbitrary permutations, and **rotation** when the processes form an appropriate topology. For instance in leader election on a ring with n processes (cf. Sec. 6.1), $P = \Diamond(\text{nr\_Leaders} = 1 \wedge \text{nr\_Passive} = n - 1)$ is symmetric under rotation of the ring. Thus symmetry reduction partitions the state space into classes of states modulo rotation, and only considers the classes (or a representative each). The resulting Kripke structure is called **quotient structure**. For rotation, its state space is reduced by the factor n, for full symmetry by the factor $n!$.

Program slicing is an abstract interpretation by selective data hiding: data and operations irrelevant for the currently checked property are statically removed. So the abstract interpretation detects which statements from the system specification description can be ignored without changing the verification result [Millet, 1998; Holzmann, 2004]. For infinite trace semantics (cf. page 71), this only works for LTL properties and assertions, whereas other properties like deadlock detection are incompatible with program slicing.

Statement merging is a special case of POR computed statically [Holzmann, 2004]: A sequence of invisible, deterministic statements within a process is wrapped into a d_step call (cf. Subsec. 3.4.3). MC retains completeness when statement merging is used, since the sequence may be executed without interleavings, as POR shows. Since the statements are invisible, the sequence may be executed atomically, i.e., intermediate steps may be eliminated.

State space compressions besides the symbolic techniques described in Sec. 5.4.2 are for instance: ETF (cf. Subsec. 3.4.3), minimized automata, tree compression and collapse compression:

Minimized automata compression replaces the hash table with a symbolic representation, using **multi-valued decision diagrams** (**MDDs**), which use bytes, not bits like BDDs do.

Tree compression [Laarman et al., 2011b] reuses parts of a state vector for multiple states, enabling compact storage: State vectors are not stored individually in one hash

table, but split up and stored in a binary tree of hash tables, where subtrees are shared amongst state vectors.

Collapse compression is similar to tree compression, but reuse is limited to one level, i.e., not done recursively via trees.

Incomplete reductions via hashing methods (often called **lossy hashing**) are mainly bitstate hashing and hash compaction. Even though this thesis focuses on sound and complete MC techniques, bitstate hashing and hash compaction are also mentioned briefly since they are very efficient. For liveness properties, lossy hashing can also become unsound.

Bitstate hashing assumes that the number b of buckets in the hash table is large enough such that the hash function is injective on the state space. Thus not the full state vectors need to be stored in the hash table, a simple bit (representing that the corresponding state has been visited) is sufficient. Needing only one bit per state, b can become very large, so collisions becomes unlikely. If they do occur, i.e., the hash function is not injective on the state space, none but one of the states colliding in a bucket are explored. Thus verification becomes incomplete. Identifying colliding states as one can cause fake cycles, resulting in unsound MC of liveness properties.

Hash compaction lowers the possibility of collisions even stronger than bitstate hashing by increasing the number of buckets even further (e.g., currently to 2^{64}) – above the number of bits fitting into memory. Thus a level of indirection is used: the bucket number (i.e., the value of the hash function f) is stored in a regular hash table. Though collisions in the regular hash table are resolved, verification again becomes incomplete (and unsound for liveness properties) iff f is not injective on the state space.

5.5. Tools

Many tools for model checking exist [URL:listFMToolsHP; URL:listVerifToolsHP; Frappier et al., 2010; Bérard et al., 2013]. One of the most prominent, SPIN, is described in Subsec. 5.5.1. Subsec. 5.5.2 describes LTSmin, a language-independent parallel MC tool. Subsec. 5.5.3 briefly describes DiVinE, a language-independent explicit state parallel on-the-fly LTL model checker. Subsec. 5.5.4 briefly describes PRISM, an implicit state probabilistic model checker. SBMC tools will be covered in Chapter 7.

5.5.1. SPIN

SPIN [URL:Spin] stands for *s*imple *P*ROMELA *in*terpreter and is one of the most popular explicit state on-the-fly MC tools for LTL and Büchi. It received the Software System Award by the Association for Computing Machinery (ACM) in the year 2001 [URL:ACMAWARD]. Its specification language is PROMELA (cf. Subsec. 3.4.3).

SPIN's Property Checks

For SPIN to be able to check properties, Boolean expressions need to be defined by symbolic names, which then can be used as atomic propositions.

SPIN can check the following safety properties (also called **safety checks**):

- deadlock detection via **end state validity**: for this check, self-loops at end states are avoided and each end state must be marked by a PROMELA label starting with the string "**end**" to be a valid end state;
- **user-supplied assertions** via **assert()**, which check the expression supplied as parameter at the locations (determined by the corresponding program counter *pc_*, cf. page 44) the assertion is given (cf. Sec. 2.2). If the expression evaluates to **false**, an **assertion violation** is reported;
- **unreachable code** existence, i.e., some statement in a process declaration that is never reached;
- correctness of **xr** and **xs assertions**: xr channelName in a process p for a channel channelName with capacity larger 0 states that all other processes do not read from channelName, likewise xw channelName for writing. These assertions are used especially to improve POR.

SPIN can check the following liveness properties (also called **liveness checks**):

- acceptance cycle detection via NDFS (cf. Subsec. 5.3.2), where acceptance states are marked by PROMELA labels starting with the string "accept";
- **never claims** (cf. Subsec. 3.4.3), for checking property descriptions in Büchi via acceptance cycle detection and for pruning the state space (cf. Subsec. 3.4.3 on page 47 and Subsec. 5.3.2). Mainly never claims without side effect are used;
- property descriptions in LTL (cf. Subsec. 5.3.2, but using a faster LTL to Büchi translation [Gastin and Oddoux, 2001]);
- livelock detection via the specialized LTL property of non-progress cycles (covered thoroughly in Chapter 6).

For high performance verification, SPIN generates a C source file (**pan.c**) that is highly specialized for the property being checked. Then the file is compiled and executed.

Liveness checks can be restricted to weakly fair linear Kripke structures (cf. Def. 5.6), but costs a strong growth of the state space. In the context of SPIN's processes, fairness_{weak} means that a process that remains continuously enabled will eventually be schedules (i.e., is not **starving**). SPIN does not support fairness_{strong}, which would forbid any **process starvation**, i.e., guarantee that a process that is recurrently enabled will eventually be scheduled.

SPIN's reductions

SPIN's various reduction methods contribute strongly to its power and success, and can mostly be combined. SPIN can reduce the state space by POR, program slicing and statement merging. Symmetry reduction has only been implemented experimentally [Bosnacki et al., 2002]. SPIN can compress the state space using minimized automata and collapse compression. It also offers lossy hashing via bitstate hashing and hash compaction.

5.5.2. LTSmin

LTSmin [URL:LTSmin; Laarman, 2014] (short for "Minimization and Instantiation of Labelled Transition Systems") is an award-winning [Howar et al., 2012] tool-set for manipulation and MC of LTSs and (labeled) Kripke structures. Its strengths are language-

independents and high-performance parallel (multi-core and distributed) MC. Fig. 5.2 (from [Laarman et al., 2013a]) shows the overall structure of LTSmin.

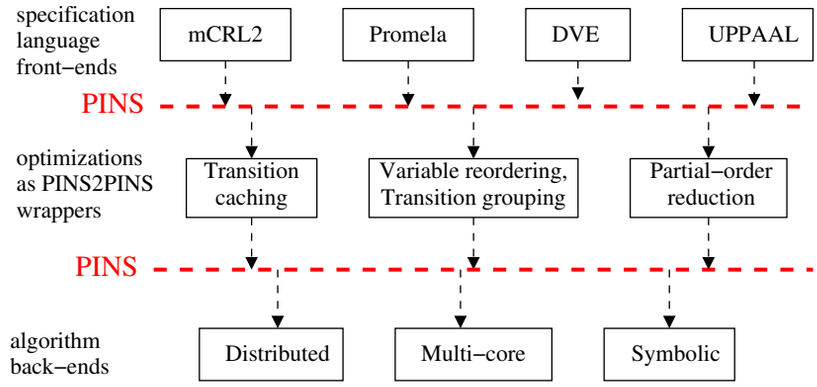


Figure 5.2.: Overview of LTSmin

Specification Language Front-Ends. LTSmin supports several system specification description languages (cf. Subsec. 3.4.3 and [Blom et al., 2010]): process algebras and infinite data structures in μ-CRL and mCRL2 [Kant and van de Pol, 2012; Cranen et al., 2013; Groote and Mousavi, 2014], state based languages (ETF (cf. Example 3.29), DVE (cf. Subsec. 5.5.3), PROMELA (cf. Subsec. 3.4.3) via **SPINS** [van der Berg and Laarman, 2012]), discrete abstractions of (biological) ODE models (MAPLE, GNA), and generalized timed automata [Dalsgaard et al., 2012; Laarman et al., 2013b]. Additionally, LTSmin can interface with other tools: with CADP (offering their formats BCG, DIR and OPEN/CAESAR interface [Garavel, 1998; Garavel et al., 2011], JTorX (cf. Subsec. 10.3.3), DiVinE (cf. Subsec. 5.5.3), the Genetic Network Analyzer (GNA), and Opaal (for timed automata, the UPPAAL input format and rapid prototyping MC algorithms).

PINS. The high number of front-ends, optimizations (on the state space or parallel algorithms) and back-ends, as well as their flexible combination, are achieved by LTSmin's design with separation of concerns [Blom et al., 2010, Fig. 1]: Its **Interface based on a Partitioned Next-State function** (**PINS**) offers the right level of abstraction for connecting specification language front-ends to algorithm back-ends. It defines not only a **next-state interface** for $\mathit{enabled}(\cdot)$ to reflect the operational semantics, but also a **static dependency matrix** for the dependency relation D (cf. Subsec. 5.4.1). Thus a concrete specification language is hidden, but not the parallel structure inherent within the specification. Therefore, the combinatorial structure of the specified system is preserved, enabling fast symbolic exploration and the optimizations described below. Integrating a new language, i.e., implementing the next-state interface and the dependency matrix, requires a few hundred LOC only. Similarly to SPIN and DiVinE, language front-ends can generate specialized C code, resulting in high-performance MC [van der Berg and Laarman, 2012].

In PINS, states are encoded as state vectors of $N \in \mathbb{N}$ **slots** (similar to Def. 5.4, but integer values instead of binary), and the transition relation T is partitioned into

K groups, describing **local transitions** between sub-vectors. This results in static dependency matrices of type $\mathbb{B}^{K \times N}$, denoting the slots that the transitions of a group might depend on (this part is called **read matrix**) and the slots those transitions might update (this part is called **write matrix**). For many specified systems, there are only few dependent slots for a group $i \in \{1, \ldots, K\}$, so the dependency matrix is sparse and the **next-state function** for i needs only operate on the **short (sub-)vector** comprising those dependent slots, not all N. Thus a single next-state function call covers the set of all transitions on full vectors where the slots independent of i have arbitrary fixed values [Blom et al., 2010].

Optimizations as PINS2PINS wrappers are reductions, such as Partial Order Reduction (cf. Subsec. 5.4.1 and [Pater, 2011; Laarman et al., 2013a]) or improvements for concurrency, such as transition grouping of the dependency matrix by merging groups, variable reordering for symbolic MC, and caching local transitions for slow language front-ends (like mCRL2 and UPPAAL). These and further optimizations are independent of front-ends and back-ends and can therefore be implemented as **PINS2PINS wrappers** to perform the optimization on-the-fly between the front-end and back-end. Therefore, they are composable and reusable for all language front-ends and most (but not all, see below) algorithm back-ends.

Algorithm Back-ends for Property Checks. LTSMIN has distributed algorithms [Lynch, 1996] to manipulate LTSs and (labeled) Kripke structures by τ-cycle elimination, strong and branching bisimilar minimization and comparison.

LTSMIN efficiently implements fully symbolic on-the-fly MC for CTL and μ-calculus properties with the help of PINS: the transition relation T is learned and stored as BDD successively.

LTSMIN uses high-performance multi-core variants of the following on-the-fly algorithms, which all yield counterexamples to violated properties:

- DFS for exploring LTSs and (labeled) Kripke structures, and checking invariants and freedom of deadlocks [Laarman, 2014];
- $\text{PDFS}_{\text{FIFO}}$, the parallelized variant of DFS_{FIFO} (cf. Chapter 6). It is currently the most efficient explicit state livelock verification;
- CNDFS for general Büchi automata LTL verification. It is currently the most efficient multi-core NDFS [Evangelista et al., 2012; van der Berg and Laarman, 2012; Laarman, 2014] and the integration of two former multi-core NDFS algorithms (LNDFS and ENDFS, both using swarm verification [Holzmann et al., 2011], but with some synchronization). Since CNDFS influences the exploration order, DFS order is not retained. Thus, the notInStack proviso C3' cannot be applied. Up to now, no solution has been found for combining CNDFS with POR.

Implementation of Parallelism. The implementations use shared memory via a lockless shared hash table (cf. Def. 6.32) that is specialized for MC and hence only offers one method [Laarman et al., 2010]: find_or_put. It is cache oriented, uses a table of fixed size 2^n, and resolves conflicts via open addressing, i.e., probing subsequent buckets. Hashing is performed incrementally (using Zobrist hashing [Laarman et al., 2011a]), which is much faster since short vectors are usually small. Freedom of locks is achieved by

compare-and-swap operations on buckets (cf. Subsec. 6.7.3). Dynamic load-balancing (cf. Subsec. 3.5.2, Subsec. 3.6.2, and Subsec. 6.7.3) for (N)DFS is implemented via synchronous random polling, for DFS_{FIFO} and symbolic MC via work stealing.

Reductions. As described above, LTSmin offers flexible reductions via PINS2PINS wrappers, e.g., for POR and symbolic techniques. Optionally, informed, incremental tree compression of states can be activated (cf. Subsec. 5.4.3). PINS is exploited to do incremental tree updates, so that the compression only requires negligible runtime.

With these implementations, LTSmin can explore more than 10 million states/sec ($\approx$ 1 GiB memory/sec) with 16 cores and achieves ideal linear scalability for DFS and $\text{PDFS}_{\text{FIFO}}$ and almost ideal for CNDFS.

5.5.3. DiVinE

DiVinE [URL:DIVINE; Barnat et al., 2013] is an explicit state parallel on-the-fly LTL model checker that offers multi-core and distributed MC. Parallelization is based on partitioning the state space into disjoint parts [Barnat et al., 2007].

Note. DiVinE is related work to this thesis since we compare it with $\text{PDFS}_{\text{FIFO}}$ in Chapter 6.

Similarly to LTSmin, DiVinE offers flexibility of system specification description languages: Next to its original, guarded command **DiVinE modeling language** (**DVE**), it also understands LLVM bitcode, UPPAAL timed automata, component interaction automata (CoIn) and MurPHI. Similarly to LTSmin, it offers an interface to decouple the language front-ends from the algorithm back-ends, called Common Explicit-State Model Interface (CESMI).

Property checks that DiVinE offers are: assertions, deadlock and livelock freedom, LTL and Büchi automata MC, time deadlocks for UPPAAL timed automata.

The current enhancements of the tool focus on LTL MC of C and C++ (C++11) via LLVM. For this, state space explosion is severe, e.g., for the peterson mutual exclusion model with two processes, over 16GB of memory are required [Rockai et al., 2013]. Thus strong **reductions** are required: besides tree compression and lossy hashing via hash compaction, DiVinE offers two reduction methods that are very strong for LTL MC of LLVM bitcode: heap symmetry reduction (also for MurPHI models) and POR (also for DVE and CoIn models). DiVinE uses $\tau+$ **reduction**, which is POR in combination with a path reduction that is similar to statement merging. To achieve these strong reductions on LLVM bitcode, it is instrumented and a shadow copy of the heap is created, for adding further information to the heap. This enables pointer analysis, for instance for POR's visibility proviso (similar to shape analysis), and effective $\tau+$ reduction. Furthermore, the heap can be transformed into a **canonical layout** via **topological sorting** [Barnat et al., 2010]. Topological sorting enables symmetry reduction for LLVM bitcode, and also a parallel cycle detection algorithm for LTL MC that is independent of the DFS order, called **One-Way-Catch-Them-Young** (**owcty**) [Fisler et al., 2001]. Therefore, owcty can combine parallelism with POR (cf. Subsec. 5.5.2 and 6.8). Furthermore, owcty is extended by a heuristic based on the MAP algorithm [Brim et al., 2004],

which optimizes OWCTY to a worst case time complexity in $O(|\mathcal{S}_{\rightarrow^*}|)$ for checking **weak LTL** properties, which are those expressible by **weak Büchi automata**, i.e., Büchi automata that contain no cycles with both accepting and non-accepting states [Barnat et al., 2009]. OWCTY is a level 0 OTF algorithm, OWCTY with heuristics a level 1 OTF algorithm (cf. Subsec. 3.6.2, [Barnat et al., 2009]).

DiVinE's current memory consistency model is very simple [Berg, 2014], but plans for DiVinE version 3.2 include LLVM's sequential consistency (seq_cst [Lattner and Adve, 2011]) and nondeterministically deferring stores in LLVM, which simulates lazy store operations, an out-of-order execution done by many modern CPUs (cf. Subsec. 3.5.2).

5.5.4. PRISM

PRISM [URL:PRISM] stands for probabilistic symbolic model checker. It is an implicit state probabilistic model checker. Since floating point values are needed to represent probability values, PRISM uses **multi-terminal binary decision diagrams** (**MTBDDs**), which extend BDDs to be able to represent arbitrary function ranges D, not just $\mathbb{B}$. Thus terminal nodes are labeled with D instead of $\mathbb{B}$ [Rutten et al., 2004; Fujita et al., 1997]; PRISM uses floating points for D.

Probabilistic Structures. Some applications exhibit probabilistic behavior that needs to be reasoned about. For instance in Subsec. 14.3.10, the probability of reaching a specific state during randomized traversal needs to be computed.

To reason about probabilities, they need to be captured: **Discrete-time Markov chains** (**DTMCs**) are structures in $\mathbb{S}_{Kripke,labeled,finite,\geq\omega}$, with transitions labeled with probability values in $[0,1]$, such that $\forall s \in S : \sum_{l \in enabled(s)} l = 1$ (i.e., $enabled(s)$ is a probability distribution). Transitions can additionally be labeled with floating points representing costs or awards, to be able to compute expected costs over a DTMC, e.g., for non-functional verification [Werner and Schmitt, 2008].

If nondeterminism should also be present, **Markov decision processes** (**MDPs**) can be used, where transitions are not only labeled with a probability l_1, but additionally with a nondeterministic choice $l_2 \in N$, with N being the set of all nondeterministic choices. In this case, $\forall s \in S\ \forall c \in N$: either $\sum_{(l_1,c) \in enabled(s)} l_1 = 1$ or $\forall (l_1, l_2) \in enabled(s) : l_2 \neq c$ must hold.

Thus for $\mathcal{S} \in$ DTMC and a path $\pi = (s_{i-1} \xrightarrow{l_i} s_i)_{i \in [1,\ldots,1+|\pi|)} \in paths(\mathcal{S})$, π represents a **random walk** on $\mathcal{S}$, and the **probability** of π is $\mathbf{P}_{paths(\mathcal{S},source(\pi))}[\pi] = \prod_{i \in [1,\ldots,1+|\pi|)} l_i$. For $\mathcal{S} \in$ MDP, the same applies, using the probabilities $(l_i)_1$ and the nondeterministic choices $(l_i)_2$.

To specify DTMCs or MDPs, PRISM offers its own system specification description language based on implicit enumeration and guarded commands.

PRISM can check **probabilistic properties** for MDPs and DTMCs using **probabilistic CTL** (**PCTL**) [Rutten et al., 2004].

Being a probabilistic model checker, PRISM does not give a counterexample if a property does not hold.

Note. PRISM also handles **continuous time Markov chains** (**CTMCs**), in which

transitions can occur in real-time, in contrast to the discrete time-steps we cover in this thesis.

5.6. Conclusion

5.6.1. Summary

This chapter gave various definitions and aspects related to MC (see also positioning in Fig. 15.1 on page 377). Several MC algorithms were introduced, with a focus on on-the-fly LTL MC. Furthermore, reductions to optimize MC were given, with a focus on partial order reduction. Finally, various MC tools were introduced, with a focus on LTSmin.

5.6.2. Contributions

Although MC is an established field with several extensive textbooks [Holzmann, 2004; Clarke et al., 1999b; Baier and Katoen, 2008], this chapter made some minor contributions:

- taxonomies of MC and of the reductions they use (similarly to [Holzmann, 2004], but not tailored towards one MC tool) were defined;
- MC was presented in a generalized way, based on Chapter 3, so that many parts can be applied in the remainder of this thesis, including MBT (cf. Part III);
- the implementation of finite ∪ infinite trace semantics was depicted for on-the-fly LTL MC (cf. Note 5.11) and POR (cf. Note 5.14).

5.6.3. Future

Possible future work includes the integration of finite ∪ infinite trace semantics into model checkers: This is simple for on-the-fly LTL MC, since the work-flow (cf. Fig. 5.1) stays identical; solely a check for acceptance in NDFS's outer DFS at end states is necessary (cf. Note 5.11). POR can simply be implemented on top since the provisos need not be adapted and the emptiness proviso guarantees that $\mathcal{S}$ and $\mathcal{S}_{\text{POR}}$ have the same end states (cf. Note 5.14).

Hence the main future work for finite ∪ infinite trace semantics is:

- implementing a more efficient translation from LTL to Büchi (cf. Subsec. 4.5.3) with finite ∪ infinite trace semantics;
- implementing finite ∪ infinite trace semantics for other temporal logics;
- investigating the results with a suitable case study, e.g., one from Subsec. 4.5.3. Measurements should also show how efficient these implementations are for property checks like LTL and deadlock detection in combination with optimizations and reductions, especially for those checks that were incompatible with some reductions for infinite trace semantics (like deadlock detection and program slicing, cf. Subsec. 5.4.3).

6. Explicit State Livelock Detection

6.1. Introduction to Livelocks

This chapter introduces new techniques for explicit state model checking of livelock properties, defined in Def. 6.1. These techniques are particularly useful for on-the-fly MC. The roots of this chapter, i.e., the motivation for improving non-progress cycle (NPC) checks, and the core idea of the algorithm, have been published in [Faragó, 2007]. The integration with POR, improvements to the algorithm and correctness proofs have been published in [Faragó and Schmitt, 2009], parallelization and experiments in [Laarman and Faragó, 2013].

Definition 6.1. A **livelock** is the liveness property of continuously making no progress.

So a livelock only occurs for infinite execution paths, and is equivalent to making progress only a finite number of times. When no more progress is made, the processes are starving (more colloquial: they hang). Like termination is one of the most prominent liveness properties for transformational systems [Manna and Pnueli, 1992], so is livelock for reactive systems.

The verification engineer determines where progress is made – which can be, for instance, the increase of a counter or the access to a shared resource. Progress is usually modeled compositionally: A **local progress state** is a local state that is marked as making progress. This can either be done in the system specification description by corresponding labels (e.g., in PROMELA all labels starting with "progress", cf. Subsec. 3.4.3), or specified separately by enumerating the local progress states for each process (e.g., in DiVinE, cf. Subsec. 5.5.3). A **global progress state** (**progress state** for short) is a global state in which at least one of the processes is in a local progress state. Global progress states are represented in Kripke structures by a corresponding propositional variable. This thesis uses the same variable as SPIN, np_ (cf. Subsec. 3.4.3), leading to Def. 6.2.

Definition 6.2. Let $\mathcal{S} = (S, T, \Sigma, I) \in \mathbb{S}_{Kripke,1,finite}$ (or $\mathcal{S} = (S, \mathfrak{T}, \Sigma, I) \in \mathbb{S}_{Kripke,1,finite,labelled}$) with $\text{np\_} \in \Sigma$. Then $\boldsymbol{S^{\mathcal{P}}} := supp(I(\text{np\_}, \cdot))$ is the set of **progress states**.

Sec. 6.5 presents progress transitions as alternative for modeling progress, especially for labeled transition systems. Therefore, Def. 6.3 and Def. 6.4 define progress more generally.

Definition 6.3. Let $\mathcal{S} = (S, \mathfrak{T}, L, \Sigma, I) \in \mathbb{S}_{Kripke,labeled,1,finite}$ or $\mathcal{S} = (S, \mathfrak{T}, \Sigma, I) \in \mathbb{S}_{Kripke,1,finite}$. Then $\boldsymbol{\mathcal{P}} \subseteq S \dot{\cup} \mathfrak{T}$ is the set of **progress**.

Definition 6.4. Let $\mathcal{S} \in \mathbb{S}_{Kripke,(labeled),1,finite}$, state $s \in S$ and $\pi \in paths(\mathcal{S}, s)$.

Then π **makes progress** (written $\boldsymbol{\pi \cap \mathcal{P} \neq \varnothing}$ for short) iff π contains a state or transition in $\mathcal{P}$: for $\pi = (s_{i-1} \xrightarrow{l_i} s_i)_{i \in [1,\ldots,1+n)}$ with $\pi \cap \mathcal{P} \neq \varnothing$, there is $j \in [1, \ldots, 1+n)$:

$s_{j-1} \xrightarrow{l_j} s_j$ makes progress, i.e.,$s_{j-1} \in \mathcal{P}$ or $s_j \in \mathcal{P}$ or $l_j \in \mathcal{P}$; for $\pi = (s_i)_{i\in[0,\dots,1+n)}$ with $\pi \cap \mathcal{P} \neq \varnothing$, there is $j \in [1,\dots,1+n) : \ s_{j-1} \to s_j$ makes progress, i.e.,$s_{j-1} \in \mathcal{P}$ or $s_j \in \mathcal{P}$ or $s_{j-1} \to s_j \in \mathcal{P}$.

We say an algorithm **traverses progress** iff it traverses a path that makes progress.

Livelocks can potentially occur in almost all protocols and parallel algorithms; Example 6.5 enumerates some. Therefore, livelock detection is one of the most important properties being model checked, and still gains importance with the rise of parallel programming (cf. Sec. 3.5 and Sec. 6.7). Livelock detection is used in about half of the case studies of [URL:PromelaDatabase] and a third of [Pelánek, 2007].

Example 6.5. Some relevant, established protocols that were all checked for livelocks (mostly with SPIN, cf. Subsec. 5.5.1) are:

- the **Group Address Registration Protocol** (**garp**), a datalink-level multicast protocol (cf. Sec. 3.5) for dynamically joining and leaving multicast groups on a bridged LAN: The garp case study [Nakatani, 1997] proved garp to be free of livelocks;
- the **General Inter-Orb Protocol** (**giop**), a key component of OMG's Common Object Request Broker Architecture (CORBA) specification for service oriented architectures: It specifies a standard protocol that enables interoperability between services (ORBs). The case studies [Kamel and Leue, 1998, 2000] found livelocks in system specifications for giop's remote object invocation and migration, which are no longer present in the current version;
- the **i-Protocol** (**i-prot** for short), an asynchronous sliding-window protocol for GNU Unix to Unix Copy (UUCP), with several optimizations to minimize traffic. The case studies [Dong et al., 1999; Holzmann, 1999] found livelocks, which are no longer present in the current version;
- the **Dynamic Host Configuration Protocol** (**DHCP**) is a client-server protocol enabling the server to dynamically configure clients for communication via the Internet Protocol. The case study Islam et al. [2006] proved DHCP to be free of livelocks;
- a **leader election protocol** determines a unique node, the **leader**, amongst all nodes. Most protocols are based on rounds, in which nodes turn passive; after finitely many rounds, exactly one node remains active and becomes the leader. The leader election protocol $\mathbf{leader}_{DKR}$ [Dolev et al., 1982; Pelánek, 2007] requires each node on a ring to have a unique ID. For anonymous rings (i.e., rings where all nodes are identical), $\mathbf{leader}_{\mathbf{Itai}}$ communicates random values and counts and limits the number of rounds [Itai and Rodeh, 1981; Faragó, 2007]. $\mathbf{leader}_t$ (baptized $\mathcal{A}_{timing}$ in [Faragó, 2007]) does not use rounds and transmissions of random values, but random waiting for higher efficiency. All these protocols were verified to be free of livelocks.

Many of these protocols are summarized in [Atiya et al., 2005].

Roadmap. Sec. 6.2 introduces NPC checks, used for livelock detection. Sec. 6.3 describes NPC checks via LTL MC, which is the currently established approach, and its deficits. Sec. 6.4 introduces possibilities for better NPC checks, leading to the algorithm

DFS_{FIFO}. It is then compared to NPC checks via LTL MC. Sec. 6.5 shows that progress transitions model progress more accurately than progress states, and how DFS_{FIFO} can process progress transitions. Sec. 6.6 motivates POR for NPC checks, modifies POR for DFS_{FIFO} and shows its correctness. It is then compared to NPC checks via LTL MC with POR. Sec. 6.7 parallelizes DFS_{FIFO}, resulting in $\text{PDFS}_{\text{FIFO}}$, shows its correctness and implementation details, and concludes with a comparison to NPC checks via parallel LTL MC. Sec. 6.8 shows experiments with four established protocols (cf. Example 6.5) to measure $\text{PDFS}_{\text{FIFO}}$'s sequential and parallel runtime and memory use, without and with POR, and its on-the-fly performance. The section compares the results with alternative algorithms and tools. Sec. 6.9 concludes the chapter with a summary of $\text{PDFS}_{\text{FIFO}}$, contributions and future work.

6.2. Introduction to Non-progress Cycle Checks

This chapter makes statements on system specifications $\mathcal{S} \in \mathbb{S}_{Kripke,(labeled),1,finite}$. Since their Büchi representations $\mathcal{A}_{\mathcal{S}}$ exhibit the same behavior and can be operated on similarly, all statements and algorithms apply for $\mathcal{A}_{\mathcal{S}}$ analogously.

If progress states are used, a livelock can be described by the LTL property $\boldsymbol{L_{livelock}} := \Diamond\Box\text{np\_}$. Since $\mathcal{S}$ is finite, Lemma 5.9 shows that $\mathcal{S}$ has a livelock iff there is a lasso $\pi \in paths^{\omega}(\mathcal{S})$ whose cycle makes no progress, called non-progress cycle in Def. 6.6. Therefore, if $\mathcal{S}$ has no deadlock (cf. Subsec. 5.5.1) and no non-progress cycle, $\mathcal{S}$ definitely makes progress eventually.

Definition 6.6. Let $\mathcal{S} \in \mathbb{S}_{Kripke,(labeled),1,finite}$. A **non-progress cycle** (**NPC**) in $\mathcal{S}$ is a reachable cycle π that makes no progress.

A **non-progress cycle check** determines whether $\mathcal{S}$ contains an NPC. Listing 6.1 refines the general contract for model checking (cf. Listing 5.1) by specifying the input, output, and contract for NPC checks. If NPC_check returns **true**, it may optionally output an error path, which is a witness for NPCs. Depending on the NPC_check, the error path can either be degenerated to a single state that is on an NPC, or be the complete NPC, or a path π from *init* to an NPC, or π concatenated with the NPC.

```
// PRE: S is a well-formed (labeled) Kripke structure                    1
//      with markings for progress.                                      2
// POST: NPC_check always terminates                                     3
//      and gives as result a Boolean output                             4
//      that is true iff S contains an NPC.                              5
//      Optionally, an error path can be output if the result is true.   6
𝔹 NPC_check(S)                                                          7
```

Listing 6.1: Contract for non-progress cycle checks

6.3. Non-progress Cycle Checks via LTL

6.3.1. Introduction

In explicit state on-the-fly MC (cf. Subsec. 5.3.2), a livelock is a counterexample to the property $\neg L_{livelock}$. $L_{livelock}$ can be translated into the Büchi automaton $\boldsymbol{\mathcal{A}_{\Diamond\Box\mathbf{np\_}}}$ given in Fig. 6.1 (cf. Subsec. 5.5.1).

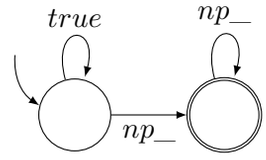


Figure 6.1.: Büchi automaton for NPC checks via $\Diamond\,\Box\,\mathrm{np\_}$

In SPIN (and LTSMIN's former implementation), $\mathcal{A}_{\Diamond\Box\mathrm{np\_}}$ is represented by the never claim given in Listing 6.2 (cf. [Holzmann, 2004]). The comments are introduced to ease understanding the walk-through of NPC checks via the NDFS for LTL MC in the next subsection. As described in Subsec. 5.3.2, $\mathcal{S} \models \neg L_{livelock}$ can be checked by the NDFS on $\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{\Diamond\Box\mathrm{np\_}}$. To avoid detecting NPCs in original end states, self-loops are either not added or they must also make progress.

Note. Using finite ∪ infinite trace semantics (cf. page 71), non-progress is also detected in finite paths with end states: a path π with an end state should terminate directly when making progress; otherwise π performs unnecessary actions between its last progress and its termination and is thus a counterexample to the property $\neg L_{livelock}$. For a check via Büchi automata, an accepting end state s is added to $\mathcal{A}_{\Diamond\Box\mathrm{np\_}}$ with transitions np_ from both states of Fig. 6.1 to s. If livelocks are defined to infinitely perform actions without making progress, we can use $L_{livelock} := \Diamond\Box(\mathrm{np\_} \wedge \mathrm{X}\ \mathbf{true})$ instead, and $\mathcal{A}_{\Diamond\Box\mathrm{np\_}}$.

```
never { /* for LTL formula <>[] np_ */                    1
  do /*nondet. delay or swap to NPC search mode*/          2
  :: np_ -> break                                          3
  :: true /*nondeterministic delay mode*/                  4
  od;                                                      5
accept:                                                    6
  do                                                       7
  :: np_ /*NPC search mode*/                               8
  od                                                       9
}                                                          10
```

Listing 6.2: Never claim for NPC checks via $\Diamond\,\Box\,\mathrm{np\_}$

6.3.2. Deficits

For the fixed formula $L_{livelock}$, LTL MC has the same worst case time and space complexities as a simple reachability (e.g., via a DFS), which is in $O(\mathcal{S}_{\rightarrow^*})$. But complexities in

Landau notation [Cormen et al., 2001] are very rough, with too pessimistic worst cases in practice, and hence only moderately meaningful (cf. Subsec. 10.2.5 and [Schnoebelen, 2002]); often the strength of optimizations like partial order reduction (cf. Subsec. 6.6.1) and parallelization (cf. Subsec. 6.8.5) are more relevant.

In detail, NPC checks via LTL perform elaborate and redundant traversals and checks, which require more resources and aggravates optimizations. The following algorithmic walk-through depicts the steps of an NPC check via the NDFS for LTL MC (cf. Listing 5.3):

1. When the traversal starts at *init* (which is usually a non-progress state), the never claim immediately swaps to its *NPC search mode* (cf. comment in Listing 6.2) because the never claim process firstly chooses np_ −> break in the first **do**-loop. Hence the outer DFS of the NDFS is performed, in which all (global) states are marked as acceptance states by the never claim and progress states are omitted, i.e., truncated (see Listing 6.2).
2. Just before the outer DFS backtracks from a state s that is traversed in the *NPC search mode*, innerDFS(s) (i.e., the nested search) starts an acceptance cycle search (since all traversed states were marked as acceptance states). For these acceptance cycle searches, the reachable non-progress states are traversed **again** (but the order in which innerDFS() is called on these states may differ from the outer DFS order in step 1).
3. If an acceptance cycle is found during innerDFS(), it is also an NPC since only non-progress states are traversed. If no acceptance cycle is found, the original call of innerDFS(s) terminates and the outer DFS in the *NPC search mode* backtracks from s. Before it backtracks from the predecessor s' of s, innerDFS(s') is called. Fortunately, states that have already been visited by some innerDFS() are not traversed again. But innerDFS() is repeatedly started many times and at least one transition has to be considered each time. Eventually, when innerDFS() has been performed for all states that the outer DFS in the *NPC search mode* has visited, the outer DFS finally backtracks to *init*.
4. Now the outer DFS in the *nondeterministic delay mode* (cf. comment in Listing 6.2) explores $\mathcal{A}_{\mathcal{S}}$ **once more**. During this, after each forward step of a non-progress state, **all previous steps are repeated**. Since most of the time the (global) states have already been visited, those procedures are immediately aborted. For a state s in the *nondeterministic delay mode*, innerDFS(s) is not called, but a progress state is also traversed.

This walk-through demonstrated that:

- the state space exploration for reaching an NPC and the NPC search are performed in separate steps. Since NPC checks are performed in innerDFS(), which the outer DFS in *NPC search mode* only calls when backtracking from a state, on-the-flyness is weak;
- the size of the state space $\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{\Diamond\Box \text{np\_}}$ is up to twice the size of $\mathcal{A}_{\mathcal{S}}$, and often close to the maximum due to few progress states. Hence the worst case space requirement of NPC checks via LTL is about $2 \cdot |\mathcal{S}_{\rightarrow^*}|$, twice as high as for a basic DFS. Furthermore, at least one additional bit per state is required to differentiate visits of the outer DFS and innerDFS();
- the original state space (i.e., $\mathcal{A}_{\mathcal{S}}$ without composition of $\mathcal{A}_{\Diamond\Box \text{np\_}}$) is traversed

three times: in the outer DFS in the *NPC search mode*, in the outer DFS in the *nondeterministic delay mode*, and in innerDFS(). Hence the worst case runtime requirement of NPC checks via LTL is about $3 \cdot |\mathcal{S}_{\rightarrow^*}|$, triple as high as for a basic DFS.

Notes. If the order within the never claim's **do**-loop were swapped, the *nondeterministic delay mode* DFS in step 4 would precede the *NPC search mode* in step 1.

In step 1, use the first np_ state instead of *init* in the rare case that *init* is a progress state.

6.4. A Better Non-progress Cycle Check

The detailed walk-through in the previous section has shown that NPC checks via LTL MC unnecessarily often traverse states and transitions (inner and outer DFS, *nondeterministic delay mode* and *NPC search mode*). The cause for this inefficiency is the general approach: LTL and Büchi MC are very powerful and cover more eventualities and options than necessary for NPC checks. So we are looking for a more specific algorithm for NPC checks that performs less redundantly. We will achieve this by combining the exploration and the NPC search phase, so that traversing the state space only once is sufficient. But with only a single traversal, we have to cope with the following problem: Simply checking for each cycle found in a basic DFS (cf. page 99) whether it makes progress is an incomplete NPC search since the DFS aborts traversal in states which have already been visited (cf. Example 6.7). Hence not all cycles are traversed.

Example 6.7. Fig. 6.2 shows an NPC that is not traversed and therefore not found by a basic DFS: From s_1, the DFS first traverses *path 1* over state s_2 that makes progress (marked with $+$) and s_3 back to s_1. After backtracking from *path 1* to s_1, the DFS traverses *path 2*, but aborts it at s_3 before closing the (red, thick) NPC. Hence if an NPC has states that have already been visited, the cycle will not be found by a basic DFS.

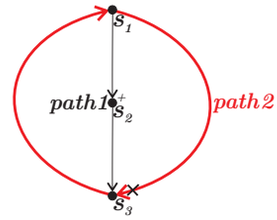


Figure 6.2.: NPC not traversed by the basic DFS

The idea for our alternative NPC checks is to guarantee that the DFS has lazy progress traversal, as defined in Def. 6.8.

Definition 6.8. Let A be a DFS algorithm that traverses $\mathcal{S}$.

Then A has **lazy progress traversal** iff after reaching a non-progress state s_0 for the first time, A traverses progress only after backtracking from s_0.

Lemma 6.9. *Let A be a DFS algorithm that has lazy progress traversal and s_0 a non-progress state.*

After A reaches s_0 for the first time, A performs a DFS over all states reachable from s_0 without traversing progress or states already visited before s_0; only thereafter A traverses progress.

Proof. The proof uses induction over the length n of the shortest path $\pi = (s_i)_{i\in[0,\dots,n]}$ from s_0 to a non-progress state s_n with $\forall i \in [0,\dots,n] : s_i$ is a non-progress state and has not yet been visited by A at the time t_0 when s_0 is visited for the first time (i.e., $s_i \notin$ hash_table).

The base case for $n = 0$ is trivial, since s_0 has just been reached at t_0.

For the induction step from n to $n+1$, let $\pi = (s_i)_{i\in[0,\dots,n+1]} \in paths^{fin}(\mathcal{S}, s_0)$ with $\forall i \in [0,\dots,n+1] : s_i$ is a non-progress state and has not yet been visited by A at t_0. If A traverses π, the induction step is proved. Otherwise, A aborts traversal at $s_i, i \in [1,\dots,n]$ because s_i has already been visited via other paths from s_0. Since these paths have been traversed after t_0, but before the path $(s_i)_{i\in[0,\dots,i]}$ has been traversed, they all contain only non-progress states that have not been visited before t_0. Let $h \in \{1,\dots,n\}$ be maximal such that s_h has already been visited via some other path from s_0, and π' the path from s_0 to s_h that A traversed to reach s_h for the first time. Then we can apply the induction hypothesis on the path $(s_i)_{i\in[h,\dots,n+1]}$ of length $\leq n$, so A reaches s_n after visiting s_h for the first time via some path π'' without traversing progress or states already visited before s_h. So A reaches s_{n+1} after visiting s_0 for the first time via the path $\pi' \cdot \pi''$ without traversing progress or states already visited before s_0. □

Lemma 6.10. *Let A be a DFS algorithm that checks NPCs with lazy progress traversal. Then A is complete.*

Proof. Let $s \in S$ be the first state on an NPC π visited by A, and $s' \in S$ the predecessor of s on π (i.e., $s' \to s$ is part of π). After A visits s for the first time, A either finds another NPC or traverses a path π' from s to s' without any progress state, as Lemma 6.9 shows. But then A detects the NPC $\pi' \cdot (s', s)$, so A is complete. □

Notes. The proof of Lemma 6.10 shows that A finds an NPC before backtracking from s, but the NPC does not have to contain s: Fig. 6.3 depicts such an example, in case that t_1 is traversed ahead of t_2.

Lemma 6.10 is shown with a pure existence proof. Constructively proving that A finds a certain NPC would be much more difficult, since that would require to consider various complex situations and the technical details of A, e.g., the order in which the transitions are traversed (cf. Fig. 6.3).

Roadmap. The following two subsections introduce two new algorithms that perform a kind of DFS with iterative deepening, i.e., they search through $\mathcal{S}$ incrementally deeper.

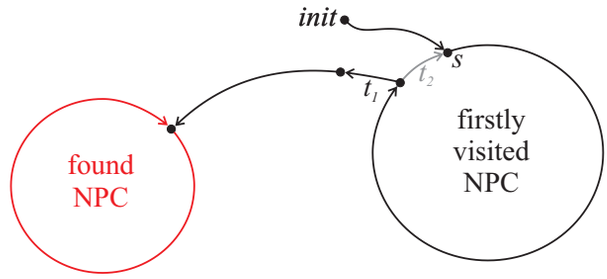


Figure 6.3.: The found NPC does not contain s

They simultaneously explore $\mathcal{S}$ from the system specification description and search for NPCs.

Firstly, Subsec. 6.4.1 considers the incremental DFS algorithm ($\text{DFS}_{\text{incremental}}$), which has similarities to BMC and was the first attempt in [Faragó and Schmitt, 2009] to improve NPC checks. Some of its techniques are reused later on in this chapter. The argumentation via sub-graphs (G_L) and the abstract method $\text{DFS}_{\text{prune,NPC}}()$ are also introduced. We show that the incremental DFS does not traverse progress lazily in all situations, and is indeed incomplete. Therefore, we improve it in Subsec. 6.4.2 by using a FIFO instead of advancing with iterative deepening, resulting in the second algorithm, DFS_{FIFO}. Finally, Subsec. 6.4.3 makes a theoretical comparison of DFS_{FIFO} with NPC checks via LTL MC, considering both expressiveness and complexity.

6.4.1. $\text{DFS}_{\text{incremental}}$

The idea is to search for NPCs using a bounded depth-first iterative deepening search with stepwise increasing **bounds** $b \in \mathbb{N}_{\geq 0}$, similar to BMC (cf. Subsec. 5.4.2). The bound is not on the length of the paths, but on the number of progress states that may be traversed on each path. So the DFS is supposed to repeatedly explore the sub-graphs $\boldsymbol{G_b}$, which are the maximal parts of the state space where all $paths_{max}(G_b, init)$ make progress maximally b times. The **incremental depth-first search** ($\textbf{DFS}_{\textbf{incremental}}$) tries to achieve this with basic DFSs (cf. page 99) by successively increasing b by one, starting from 0. $\text{DFS}_{\text{incremental}}$ terminates either with an error path from *init* to an NPC, inclusively, when one is found, or with "structure does not contain NPCs" when b becomes big enough for $\text{DFS}_{\text{incremental}}$ to explore the complete state space.

So in each state s we might prune some of the outgoing transitions by omitting those which exceed the current progress limit b, and only consider the remaining transitions.

The algorithm for $\text{DFS}_{\text{incremental}}$ is given as pseudocode in Listing 6.3 and Listing 6.4: It uses a **progress_counter** for the number of progress states on the current path from *init* to the current state. The underlined methods of $\text{DFS}_{\text{prune,NPC}}$ in Listing 6.4 are polymorphic, with the following implementations for $\text{DFS}_{\text{incremental}}$:

- pruned(s,t): **if** (t is a progress state)

```
then if (progress_counter == b)
     then return true;
     fi;
     progress_counter++;
```

```
int b:=0;                                                          1
int progress_counter:=0;                                           2
𝔹 DFS_pruned;                                                      3
Stack stack := new Stack();                                        4
HashTable hash_table;                                              5
                                                                   6
proc DFS_starting_over(S→* s)                                      7
  repeat                                                           8
    DFS_pruned:=false;                                             9
    hash_table := new HashTable();                                 10
    DFS_prune,NPC(s); //cf. Listing 6.4                            11
    b++;                                                           12
  until (!DFS_pruned);                                             13
end;                                                               14
                                                                   15
proc main()                                                        16
  DFS_starting_over(init);                                         17
  printf("structure does not contain NPCs");                       18
end;                                                               19
```

Listing 6.3: Typed $\text{DFS}_{\text{incremental}}$

```
        fi;
        return false;
```

- pruning_action(t): DFS_pruned := **true**;
- np_cycle(t): **return** (cycle on stack contains no progress state);. Instead, this check can be accelerated with an insignificant increase in memory (maximally $\log(b_{max}) \cdot$ depth(state space) bits): l.3 of Listing 6.4 pushes the current progress_counter together with each state onto stack, but the progress_counter is ignored in l.10 when states are being compared. Then np_cycle(t): **return** (progress_counter == counter on stack **for** t); (i.e.,**true** iff the current progress_counter == progress_counter when t was visited last time on the current path);
- error_message(): print(stack);, which corresponds to the path from *init* to the NPC (inclusively);
- backtrack(): pop s from stack;

```
        if (s is a progress state)
        then progress_counter--;
        fi;
```

For progress transitions, the only modification is checking for progress transitions instead of progress states in pruned(s,t) and np_cycle(t).

Unfortunately, $\text{DFS}_{\text{incremental}}$ has several deficiencies:

- due to iterative deepening, the beginning part of the state space is traversed repeatedly. But since usually the state space grows exponentially in depth, $\text{DFS}_{\text{incremental}}$ is only a constant factor slower than a basic DFS;
- $\text{DFS}_{\text{incremental}}$'s main disadvantage is that it does not always traverse progress lazily: It can happen that NPCs are reachable in the state space via $p \in \mathbb{N}$ progress states (i.e., are within G_p), but $\text{DFS}_{\text{incremental}}$ reaches them only for bounds larger

```
proc DFS_prune,NPC(S_→* s)                                          1
  add s to hash_table;                                              2
  push s onto stack;                                                3
  for each S_→* t with s → t do                                     4
    if (t ∉ hash_table)                                             5
    then if (!pruned(s,t))                                          6
         then DFS_prune,NPC(t)                                      7
         else pruning_action(t)                                     8
         fi                                                         9
    else if (t ∈ stack && np_cycle(t))                              10
         then halt with error_message()                             11
         fi                                                         12
    fi                                                              13
  od;                                                               14
  backtrack();                                                      15
end;                                                                16
```

Listing 6.4: Typed generic $\text{DFS}_{\text{prune,NPC}}$: DFS with pruning and NPC check

than p. Such a situation is depicted in Fig. 6.4, with the traversal order equal to the path number, s_0 a state on an NPC and $+$ marking progress: for $b = 2$, *path 1* is pruned before s_2 is reached. Then *path 2* is traversed via s_1 until the progress between s_1 and s_0. Then *path 3* is traversed but aborted at s_1, which has already been visited. *path 3*$\cdot$NPC at s_0 is, however, within G_2. For $b = 3$, *path 1* visits s_2, but traversal does not continue to s_1. Then *path 2* is pruned before s_1 is reached. Thus *path 3* reaches s_0 and hence the NPC for the first time. Since *path 3* makes progress twice but $b = 3$, the NPC might not be detected, as described for Fig. 6.2. Thus $\text{DFS}_{\text{incremental}}$ is incomplete;

- hence, depending on the traversal order of $\text{DFS}_{\text{prune,NPC}}$, the progress_counter limit might have to become unnecessarily large until an NPC is found.

$\text{DFS}_{\text{incremental}}$ could be modified to become complete by storing the current progress progress_counter for each state on the hash_table and allowing retraversals for a smaller progress_counter. But the following subsection shows a more efficient modification to $\text{DFS}_{\text{incremental}}$.

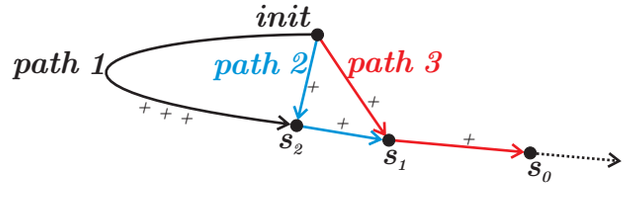


Figure 6.4.: Example for $\text{DFS}_{\text{incremental}}$ not traversing progress lazily

6.4.2. $\mathrm{DFS}_{\mathrm{FIFO}}$

The **incremental depth-first search with FIFO** buffering of progress does not repeatedly increase the bound b and retraverse the beginning part of the state space. Instead, it buffers the pruned progress states to jump back to them later on to continue traversal. Roughly speaking, we perform a breadth-first search (BFS) with respect to the progress states, and in-between progress states we perform DFSs: To reuse sub-graphs already explored instead of repeatedly retraversing sub-graphs like $\mathrm{DFS}_{\mathrm{incremental}}$, we have to buffer some extra information to know which transitions have been pruned. One way to track these pruned transitions is by using a **FIFO queue** (**FIFO** for short), causing a BFS over progress states and hence shortest counterexamples (but other orders than FIFO also yield correct NPC checks, cf. Note 6.33). This results in the $\mathrm{DFS}_{\mathrm{FIFO}}$ algorithm as defined in Def. 6.11.

Definition 6.11. $\mathrm{DFS}_{\mathrm{FIFO}}$ is the algorithm given in Listing 6.5. It uses the abstract $\mathrm{DFS}_{\mathrm{prune,NPC}}$ from Listing 6.4, with the polymorphic methods implemented as follows (for progress modeled by states):

- pruned(s,t): **return** (t is a progress state);
- pruning_action(t): **if** ($t \notin$ fifo) **then** put t into fifo; **fi**;
- np_cycle(t): **return** (t is not a progress state);
- error_message(): print(stack);, which corresponds to the lasso from the last progress to the NPC (inclusively);
- backtrack(): pop state from stack;

```
Stack stack := new Stack();                                          1
HashTable hash_table := new HashTable();                             2
FIFO fifo := new FIFO();                                             3
                                                                     4
proc DFS_FIFO(S_→* s)                                                5
  put s in fifo;                                                     6
  repeat                                                             7
    pick first s out of fifo;                                        8
    DFS_prune,NPC(s)                                                 9
  until (fifo is empty);                                             10
end;                                                                 11
                                                                     12
proc main()                                                          13
  DFS_FIFO(init);                                                    14
  printf("structure does not contain NPCs");                         15
end;                                                                 16
```

Listing 6.5: Typed $\mathrm{DFS}_{\mathrm{FIFO}}$

Notes 6.12. For progress transitions, the only modification is again to check for progress transitions instead of progress states in pruned(s,t) and np_cycle(t) (cf. Sec. 6.5, Sec. 6.6 and Sec. 6.7 for details).

The check of np_cycle(t) is sufficient since only the first element on the stack (except maybe for *init*) is a progress state. This and further properties also become apparent when considering the contract for $\text{DFS}_{\text{prune,NPC}}$ in Listing 6.6.

Instead of introducing a FIFO, a simpler version of DFS_{FIFO} puts pruned states underneath the stack instead of in fifo. When it is time to consider the state, the original stack becomes empty and $\text{DFS}_{\text{prune,NPC}}$ will pop them from the stack and consider them. Compared to DFS_{FIFO}, this results in a simpler version:

- pruning_action(t): put t underneath the stack;
- np_cycle(t): needs not change since the stack now has only progress states at its bottom (representing the former fifo) and thereafter only non-progress states (representing the original stack);
- error_message(): print(non−progress states of stack);
- backtrack(): pop s from stack;
  ```
  if (stack.peek() ∉ hash_table)
  then DFS_prune,NPC(stack.pop());
  fi;
  ```
- $\text{DFS}_{\text{prune,NPC}}(\textit{init})$ can be called directly to start an NPC check.

In summary, DFS_{FIFO} does not repeatedly start exploring the state space from scratch, but rather uses the sub-graphs already explored, lazily picks the progress states out of fifo, and expands the state space further by continuing the basic DFS. When a new progress state is reached, traversal is postponed by putting the state into fifo. When eventually $\text{DFS}_{\text{prune,NPC}}$ is finished and fifo is empty, the complete state space has been explored.

Implementation pruning_action(t): put t into fifo; (i.e., without first checking whether $t \in$ fifo) is also correct: if t is present multiple times in fifo, all but the first call of $\text{DFS}_{\text{prune,NPC}}(t)$ will not cause recursive calls. But then performance can be improved by $\text{DFS}_{\text{FIFO}}()$ only calling $\text{DFS}_{\text{prune,NPC}}(t)$ if $t \notin$ hash_table (cf. Sec. 6.7).

DFS_{FIFO} erases a large part of the stack: everything behind progress states, i.e., all of the stack between *init* and the last progress state, is lost. But for detecting NPCs via the stack, having only non-progress states on the stack (besides the first element) is a feature and not a bug: Exactly the NPCs are detected by considering the stack; the cycles that go back to states from previous runs are progress cycles (i.e., cycles with some progress state) and stay undetected. A further benefit will arise in combination with partial order reduction (see Sec. 5.4.1 and 6.6).

Consequently, if an NPC is detected, the stack from the current run supplies the NPC, but additional work is required for error_message() to report more information like the complete error path from *init*: The shortest (w.r.t. progress) counterexample π can be found quickly, for instance with a basic BFS up to the NPC, or with one of the following methods, which usually require only little additional memory: Instead of storing only the last progress state of each pruned path in fifo, all progress states are saved forever, e.g., in a tree of progress states. Then error_message() can simply print all progress states on π, or reconstruct the full path π using a fast DFS that is guided by the progress states on the error path. Alternatively, each state can store a back-link to the predecessor it was reached from. The path from *init* to the NPC can then be constructed by backward traversal in O(|counterexample|) steps. This approach will be used in Sec. 6.8, since LTSMIN already uses back-links. For easy combination with lossy hashing techniques,

the other approaches can be chosen.

DFS_{FIFO} avoids the deficiencies of $\text{DFS}_{\text{incremental}}$:

- the traversed sub-graphs do not become unnecessarily large, which saves resources and produces shortest counterexamples with respect to progress;
- we avoid the redundancy of $\text{DFS}_{\text{incremental}}$ by reusing sub-graphs, i.e., the part of the state space already explored previously, which saves resources;
- lazy progress traversal is fulfilled, as the proof of Theorem 6.13 shows in detail.

Theorem 6.13. *DFS_{FIFO} finds an NPC if one exists and otherwise outputs that no NPC exists. An NPC is found at the smallest depth w.r.t. progress, i.e., after the smallest number (b_0) of progress states that have to be traversed from init to an NPC.*

Proof. DFS_{FIFO} is **sound**, i.e., it does not output false NPCs: DFS_{FIFO} only postpones transitions, but does not generate new ones. It checks for NPCs by searching through the stack (except the first state if it is a progress state), so it only considers non-progress states.

DFS_{FIFO} is **complete**, i.e., it finds an NPC if one exists: This is shown via Lemma 6.10 since $\text{DFS}_{\text{prune,NPC}}$ has lazy progress traversal: during the call $\text{DFS}_{\text{prune,NPC}}(s)$ on a non-progress state s being visited the first time, all encountered progress states (or progress transitions) are pruned due to pruned() and pruning_action(). They are traversed only by retrieving them from fifo in $\text{DFS}_{\text{FIFO}}()$, which is done only after $\text{DFS}_{\text{prune,NPC}}(s)$ has finished.

DFS_{FIFO} finds an **earliest NPC w.r.t. progress**: As long as DFS_{FIFO} explores G_b for $b < b_0$, all paths leading to an NPC are pruned. Let $b = b_0$. Since $\text{DFS}_{\text{prune,NPC}}$ has lazy progress traversal and DFS_{FIFO} considers progress states in a first come first serve approach (with the help of fifo), DFS_{FIFO} explores the full sub-graph G_{b_0}. During this exploration, an NPC is found since DFS_{FIFO} is complete. □

Notes 6.14. DFS_{FIFO} does not know which G_b is currently explored. If we want to keep track on the NPC's depth w.r.t. progress, we can swap between two FIFOs: one for reading and one for writing; when the FIFO that is read from is empty, the current G_b is finished and we swap it with the FIFO for $G_{b+1} \setminus G_b$. But even the original DFS_{FIFO} successively generates $G_{b+1} \setminus G_b$ for increasing b, although the phases are not explicitly separated. Hence we may still talk about the sub-graphs G_b.

Theorem 6.13 is proved graph-theoretically via Lemma 6.9, which is shorter than the proofs via contracts that we found. As the contract can help understand the algorithm, it is given in Listing 6.6. The precondition can be weakened by not requiring $s \notin$hash_table (cf. Note 6.12 and Note 6.23). Sec. 6.7 will prove the parallel DFS_{FIFO} correct using similar contracts. The parallel DFS_{FIFO} subsumes DFS_{FIFO} – algorithmically as well as in the implementation. This subsection presented the sequential algorithm and its proof since parallelization adds further complexity to the DFS_{FIFO} algorithm and its proof of correctness.

In short: DFS_{FIFO} traverses each state through a path with the fewest possible progress states and successively explores G_b for increasing b until an earliest NPC w.r.t. progress is reported or the whole state space has been explored and "structure does not contain NPCs" is reported.

```
// INV: no element ever gets removed from the hash_table;                                    1
// INV: ∀s′ ∈ stack: s′ is oldest element on stack <==>                                      2
//       s′ is a progress state or init;                                                     3
// INV: ∀s, s′ ∈ stack: s′ is the successor of s on the stack ==> s → s′;                    4
// INV: ∀s ∈ hash_table \ (stack ∪ fifo): no NPC is reachable from s                         5
//       without traversing progress and ∀s′ ∈ dest(s, →) : s′ ∈ hash_table ∪ fifo;          6
//                                                                                           7
// PRE: s ∉ hash_table and (s == init or ∃s′ ∈ hash_table: s′ → s);                          8
//                                                                                           9
// POST: NPCs are reachable from s without traversing progress ==>                           10
//       an NPC is reported;                                                                 11
// POST: ∀ states s′ ∉ |old(hash_table): s′ is reachable from s only by                      12
//       traversing progress ==> s′ ∉ hash_table;                                            13
// POST: NPC has been reported ==>                                                           14
//       a real suffix in the stack forms an NPC;                                            15
// POST: no NPC has been reported ==>                                                        16
//       fifo consists of |old(fifo) appended by all states                                  17
//       s′ ∉ |old(fifo)∪|old(hash_table) that are reachable via some                        18
//       path π · s″ → s′ and π makes no progress, but s″ → s′ does;                         19
// POST: no NPC has been reported ==>                                                        20
//       s ∈ hash_table and s ∉ stack and s ∉ fifo;                                          21
// POST (redundantly): no NPC has been reported ==>                                          22
//       ∀ states s′ reachable from s without traversing progress:                           23
//       s′ ∈ hash_table and s′ ∉ stack and s′ ∉ fifo;                                       24
proc DFS_prune,NPC(S_→* s)                                                                   25
```

Listing 6.6: Contract for DFS_{FIFO}'s $\text{DFS}_{\text{prune,NPC}}$

6.4.3. Comparing LTL NPC Checks to DFS$_{\text{FIFO}}$

This subsection theoretically compares DFS$_{\text{FIFO}}$ with the basic DFS and with NPC checks via LTL MC. Practical experiments will be presented in Sec. 6.8.

Before we compare performance, we consider expressiveness. For this Def. 6.15 restricts the conjunctive Büchi automaton of Büchi and LTL MC (cf. Subsec. 5.3.2): instead of using an arbitrary Büchi automaton (e.g., generated from an LTL formula) $\mathcal{A}_{never}$ as property description, $\mathcal{A}_{\Diamond\Box\phi}$ is used.

Definition 6.15. Let $\phi \in \text{PROP}_\Sigma$. Then $\boldsymbol{NPC_\phi}$: Büchi $\to$ Büchi, $\mathcal{S} \mapsto \mathcal{S} \cap \mathcal{A}_{\Diamond\Box\phi}$ maps the specification Büchi automaton $\mathcal{S}$ to the **generalized NPC conjunctive Büchi automaton** $\mathcal{S} \cap \mathcal{A}_{\Diamond\Box\phi}$.

Since $Models_{lin}(NPC_{\text{np\_}}(\mathcal{S})) = \{\mathcal{S}_\pi | \pi \in paths^\omega(\mathcal{S}) \text{ with } \mathcal{S}_\pi \models \Diamond\Box\text{np\_}\}$, $NPC_{\text{np\_}}(\cdot)$ corresponds to the temporal logic formalism that represents our NPC checks. We can easily implement the generalized NPC checks $NPC_{\text{np\_}}(\cdot)$ to not check for np_, but rather for an arbitrary $\phi \in \text{PROP}_\Sigma$. This results in checking the **persistence property** [Baier and Katoen, 2008] $\Diamond\Box\phi$. Still LTL $\gtrsim NPC_{\text{PROP}_\Sigma}(\cdot)$; $\Box\phi$ is an LTL example not expressible as persistence property, which can only enforce that ϕ eventually becomes an invariant. So $NPC_{\text{PROP}_\Sigma}(\cdot)$ is a specialization of LTL, i.e., a real sub-class of LTL. This explains why DFS$_{\text{FIFO}}$ can perform better for livelock detection, but unfortunately cannot be employed for full LTL MC.

Extending DFS$_{\text{FIFO}}$ by allowing checks, e.g., assertions, in the specification Büchi automaton $\mathcal{S}$ (cf. Subsec. 3.4.3) is not a strong improvement: Lemma 6.16 shows that expressivity does not reach LTL.

Lemma 6.16. *Büchi* $\gtrsim NPC_{PROP_\Sigma}$*(Büchi)* $\not\geq$ *LTL.*

Proof. Let $q \in \Sigma, \phi \in \text{PROP}_\Sigma, Q := \{R \subseteq \Sigma | q \in R\}, \overline{Q} := 2^\Sigma \setminus Q, \Phi := \{R \subseteq \Sigma | \exists I : val_I(\phi) = \textbf{true} \text{ and } R = supp(I(\cdot))\}$, and $L := \Box\Diamond q$. So $Prop(L) \leftrightharpoons (\overline{Q}^* \cdot Q)^\omega$ (the ω-regular language of L, with $A = 2^\Sigma$). Assume $\exists \mathcal{A} \in$ Büchi: $NPC_\phi(\mathcal{A}) \equiv L$. Let n be the number of states of $NPC_\phi(\mathcal{A})$. Since $(\overline{Q}^{n+1} \cdot Q)^\omega \subseteq Prop(L)$, there must be a path $\pi \in paths^\omega(NPC_\phi(\mathcal{A}))$ for which eventually invariantly ϕ and infinitely many prefixes have traces that end with $\overline{Q}^{n+1}$. Thus $\pi = \pi_{prefix} \cdot \pi_{middle} \cdot \pi_{suffix}$ with π_{middle} being a cycle whose trace is in $(\overline{Q} \cap \Phi)^+$, i.e., for $\pi_{prefix} \cdot \pi_{middle}^\omega \in paths^\omega(NPC_\phi(\mathcal{A}))$ eventually invariantly ϕ. Thus $NPC_\phi(\mathcal{A})$ accepts a property $P \in A^{|\pi_{prefix}|} \cdot (\overline{Q} \cap \Phi)^\omega$, contradicting $P \notin Prop(L)$.

Since Büchi $\gtrsim$ LTL and NPC_{PROP_Σ} constructs a conjunctive Büchi automaton, we also have Büchi $\gtrsim NPC_{\text{PROP}_\Sigma}$(Büchi). □

Note 6.17. More advanced modifications to DFS$_{\text{FIFO}}$ are possible, e.g., checks that are not decided in a local state. But the modifications for these checks can become difficult. For instance, efficiently lifting $NPC_{\text{PROP}_\Sigma}(\cdot)$ to $NPC_{NPC}(\cdot)$ by nesting formalisms and DFS$_{\text{FIFO}}$ (similarly to ECTL*), is not possible since pruning (as for progress in the original DFS$_{\text{FIFO}}$) must be at the same states for each nesting and decided on-the-fly for DFS$_{\text{FIFO}}$ to work in one pass over the state space. One possible extension, suggested by Henri Hansen in a conversation, is to verify the important class of **response properties**, which have the form $L = \Box(A \to \Diamond B)$ with $A, B \in \text{PROP}_\Sigma$ [Baier and Katoen, 2008].

Since $\neg L = \Diamond(A \wedge \Box\neg B)$, this is a generalization of persistence properties. DFS_{FIFO} has to be modified to check for NPCs only for certain states, which adapts aspects from testing automata (see end of this subsection and [Hansen et al., 2002]). However, implementing it and investigating correctness and compatibility with optimizations is future work.

Comparing the expressiveness of DFS_{FIFO} and NDFS also shows that NPC checks can be performed easier than acceptance cycle checks: NPC checks make use of the fact that the moment progress is found at state s, the current sub-path up to s is definitely not part of an NPC. Thus the current sub-path (i.e., the current stack) can be discarded, and the NPC search can be resumed in s later on. Contrarily, when searching for an acceptance cycle, there is no situation where the current sub-path can be discarded: There always remains the possibility that an acceptance state still occurs and the path forms a loop to an element earlier in the current sub-path. Therefore, only when all reachable acceptance states do not contribute to an acceptance cycle, the path to s is no longer required. This is exactly the check that NDFS performs.

As for NPC checks via LTL (cf. Subsec. 6.3.2), we compare DFS_{FIFO}'s complexities also to the basic DFS: For DFS_{FIFO}, both time and space requirements are about the same as for the basic DFS because DFS_{FIFO} only requires **one pass**: all states are traversed only once by a simultaneous exploration of the state space and NPC search. DFS_{FIFO} only needs additional operations for checking whether $s \to t$ makes progress, and if so pushing and popping t once to and from fifo. To construct a full counterexample, maximally the resources of one more basic DFS are required, but usually far less (cf. Subsec. 6.4.2 on page 126). DFS_{FIFO} requires no additional space compared to a basic DFS because progress states are stored only temporarily in fifo until they are stored in hash_table (cf. Listing 6.4 and Listing 6.5). Since DFS_{FIFO}'s stack stores only part of the full path, it needs less space than DFS's stack.

Besides the time and space requirements, other aspects of MC algorithms are also important: Summarizing what work has already been performed, usability, on-the-flyness, heuristics, and effectiveness of reductions. These will be considered in the rest of this subsection.

DFS_{FIFO} can output more information upon termination or when it runs out of time or space: The bound b (or the sub-graph G_b) for which DFS_{FIFO} has verified G_b to be free of NPCs (which is as much information as BMC gives for too small bounds, cf. Subsec. 5.4.2). Furthermore, fifo can be output as helpful information for the test engineer, or to turn DFS_{FIFO} into a conditional model checking algorithm (cf. Subsec. 5.1): By adding the contents of fifo as input C_{input} and as output C_{output} to DFS_{FIFO}, the NPC check can continue where a former check left off, i.e., at the border of G_b instead of *init*.

Sec. 6.5 shows that DFS_{FIFO} can specify progress in a better way using progress transitions instead of progress states.

Since DFS_{FIFO} performs NPC checks in one pass, it has strong on-the-flyness (cf. Subsec. 6.8.6).

DFS_{FIFO} comprises an efficient search heuristic since paths making less progress are preferred: Since the small scope hypothesis often holds (cf. Subsec. 5.2.3), many livelocks in practice already occur after very little progress (e.g., for the i-Protocol after 2 sends and 1 acknowledge, cf. [Dong et al., 1999]). Since the state space usually grows exponen-

tially with its depth, this heuristic also improves on-the-flyness (cf. *shallow* in Table 6.9). Additionally, **shortest (w.r.t. progress) counterexamples** show faults that occur more often in practice and are easier to understand. Since progress is the most important aspect for NPC checks, minimizing the number of progress states (or progress transitions, cf. Sec. 6.5) on the counterexamples is more important than minimizing other aspects, such as the overall length of the path. Alternative algorithms to find better counterexamples minimize the length of the counterexample, require additional processing and are hard to combine with POR; examples are [Gastin and Moro, 2007], which uses a nested BFS guided by priority queues, [Hansen and Kervinen, 2006], which uses an interleaved BFS that explores some transitions backwards, and [Edelkamp et al., 2004], which uses heuristics that operate on the counterexample found by SPIN's NDFS to reduces its length afterwards, and does not guarantee to find a shortest counterexample.

Finally, the effectiveness of a verification largely depends on the strength of the additional optimization techniques involved, especially partial order reduction (cf. Subsec. 5.4.1 and Sec. 6.6) and parallelization (cf. Sec. 6.7).

We compare DFS_{FIFO} to the following algorithms, besides NDFS, that we found to also cover on-the-fly livelock detection:

Before performing NPC checks via LTL MC, SPIN used a variant of the NDFS algorithm (cf. Listing 5.3) specialized to detect NPCs [Holzmann, 1992]: $\text{innerDFS}(s)$ is called each time the outer DFS backtracks from a non-progress state s. $\text{innerDFS}(s)$ only traverses non-progress states and checks with an own stack whether a cycle is closed, which is an NPC. Similar to NPC checks via LTL MC, this algorithm does not have lazy progress traversal, performs redundant work, and does not have strong on-the-flyness. Furthermore, it is not compatible with partial order reduction (see Note 6.22).

Henri Hansen has pointed out that [Valmari, 1993] already contains an algorithm that is similar to DFS_{FIFO}: It performs a check for **divergent traces**, which are reachable livelocks from special states, but does not use Büchi automata but **tester processes**, which comprise multiple property detections like deadlocks, divergent traces, and acceptance cycles. Furthermore, the algorithm additionally marks transitions and has only weak optimizations: weak partial order reduction (cf. Subsec. 6.6.3) and no parallelization. The algorithm is not further investigated; [Valmari, 1993] "only makes informal arguments" and "does not have any analytical or experimental results" that compares the algorithm to alternatives. But the algorithm already uses the idea of performing lazy progress traversal by postponing progress. Valmari's later work on detecting divergent traces via testing automata (similar to tester processes) used a different algorithm that is closer to the NDFS [Hansen et al., 2002]: the so-called one pass algorithm does not traverse progress lazily but traverses the state space in one pass, e.g., in a DFS or BFS, but additionally calls an inner traversal function (called LLDET) for non-progress states. LLDET marks states with 3 colors during its traversal to detect NPCs. The one pass algorithm is more general but also more complex than DFS_{FIFO}: its set of states to still work on contains both non-progress and progress states and an inner traversal and coloring is performed. Consequently, correctness is more difficult to prove, the algorithm has weaker optimizations (like partial order reduction and parallelization), it has less on-the-flyness, and does not find shortest counterexamples even if the set of states to still work on is a FIFO.

6.5. Progress Transitions

NPC checks via LTL MC need to mark states as having progress because the Büchi automata can only consider I, not $\mathfrak{T}$ of the labeled Kripke structure $\mathcal{S} = (S, \mathfrak{T}, L, \Sigma, I)$. Since our DFS_{FIFO} can operate directly on $\mathcal{S}$ without needing Büchi automata, it can use progress transitions instead, i.e., labeled transitions in $\mathfrak{T}$ that are marked as making progress.

6.5.1. Semantics

We define progress transitions and one way of marking them in Def. 6.18. Using a labeled Kripke structure instead of an unlabeled one is no restriction for MC of system specification descriptions where basic commands are used to describe how transitions update the state vector, since these basic commands can be used as statements L (cf. Subsec. 3.4.3).

Definition 6.18. Let $\mathcal{S} = (S, \mathfrak{T}, L, \Sigma, I) \in \mathbb{S}_{Kripke,labeled,1,finite}$. The set of **progress transitions** is $\mathfrak{T}^{\mathcal{P}} := \{(s, \alpha, s') \in \mathfrak{T} \mid \alpha \text{ starts with "progress"}\}$.

Progress transitions model progress more naturally than progress states, since an action of the system constitutes the actual progress, e.g., the increase of a counter or acquisition of a resource. Similarly, tester processes and testing automata (see end of Subsec. 6.4.3) observe changes in state propositions, not states and their propositions.

So **progress transition semantics**, i.e., $\mathcal{P} = \mathfrak{T}^{\mathcal{P}}$, is more direct than **progress state semantics**, i.e., $\mathcal{P} = S^{\mathcal{P}}$, which also shows when comparing them for NPC checks: Fig. 6.5 shows an automaton with $\mathcal{S}^{\mathcal{P}} = \{s_2\}$ and $\mathfrak{T}^{\mathcal{P}} = \{(s_2, \alpha, s_1)\}$. With progress state semantics, the cycle $\pi = s_2 \to s_3 \to s_2$ exhibits only **fake progress**: the system's action performing progress, resulting in the labeled transition α, is never taken. With progress transition semantics, π can be detected as NPC.

Figure 6.5.: Fake progress cycle for $\mathcal{P} = S^{\mathcal{P}}$, but not for $\mathcal{P} = \mathfrak{T}^{\mathcal{P}}$

Example 6.19. Fake progress cycles occur often in parallel systems: If process1 in Fig. 6.6 has the progress transition $s_2 \to s_1$ and the progress state s_2, and process2 has no progress, then the parallel composition contains the fake progress cycle $(s_2, t_1) \to (s_2, t_2) \to (s_2, t_1)$. In this simple case, the alternative NPC $(s_1, t_1) \to (s_1, t_2) \to (s_1, t_1)$ can be detected. But if a transition between t_1 and t_2 is only enabled if process1 is in s_2, no alternative NPC exists. An implementation for this is given in Appendix A.1. SPIN's LTL NPC check via progress states detects no NPC for the implementation, i.e., is not complete for livelock detection when actions and not states constitute progress. Our fake progress cycle in Fig. 6.6, which corresponds to the NPC with output $(\text{process2 restarted without progress})^{\omega}$, is constructed by SPIN as a counterexample for the LTL property $\neg\Diamond\Box(\text{process1globalPC==1})$ when weak fairness is disabled.

Fake progress cycles can often be hidden by enforcing strong (A-)fairness, but (A-)fairness, like SPIN's weak fairness, is usually insufficient (cf. Def. 5.6, Subsec. 5.5.1 or Baier and Katoen [2008]; Holzmann [2004]). If the action making progress is only enabled outside the fake progress cycle (e.g., implemented analogously to Listing A.1), no fairness can hide the fake progress cycle. Only if all fake progress cycles in $\mathcal{S}$ are hidden by fairness does the livelock detection become complete again with fairness semantics. Enforcing any kind of fairness is costly.

Note. To directly compare progress states with progress transitions, let $(s, \alpha, t) \in \mathfrak{T}$. If $(s, \alpha, t) \in \mathfrak{T}^{\mathcal{P}}$, then $s \in S^{\mathcal{P}}$. But if $s \in S^{\mathcal{P}}$, then not necessarily $(s, \alpha, t) \in \mathfrak{T}^{\mathcal{P}}$ because s is a progress state if s contains a local state s_e (cf. Subsec. 3.4.3) and $\exists s' \in S$ such that s' contains s_e and $\mathit{enabled}(()s') \cup \mathfrak{T}^{\mathcal{P}} \neq \varnothing$.

If it only depends on α whether $(s, \alpha, s') \in \mathfrak{T}$ (see next subsection), then a state s is a progress state iff $\mathit{enabled}(s) \cap \mathfrak{T}^{\mathcal{P}} \neq \varnothing$. So a progress state just means that a progress transition is enabled. With this, if none of the enabled progress transitions on a cycle are actually chosen, it is a fake progress cycle in the progress state semantics.

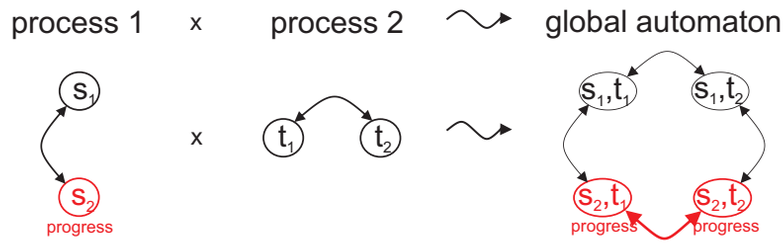


Figure 6.6.: Fake progress cycle by parallel composition

6.5.2. System Specification Descriptions

Similar to progress states, progress transitions can be specified in system specification descriptions by annotating the appropriate basic command c with some specification language label, e.g., "progress:c" (cf. Subsec. 3.4.3). Then each update $(s, \alpha, s') \in \mathfrak{T}$ caused by progress:c is marked as making progress, i.e., $\forall (s, \alpha, s') \in \mathfrak{T}$ with $\alpha =$progress:c: $(s, \alpha, s') \in \mathfrak{T}^{\mathcal{P}}$. Hence we also write $\alpha \in \mathfrak{T}^{\mathcal{P}}$ for labeled Kripke structures. Alternatively, all progress transitions can be enumerated separately. Both approaches will be used in Sec. 6.8: labels for PROMELA specifications, enumeration for DVE specifications.

Note. Using progress labels as just described changes PROMELA's semantics. To avoid this change, one could try staying in PROMELA's progress state semantics: statement$_i$; progress: statement$_j$; sets the local state s between statement$_i$ and statement$_j$ to a local progress state . Transforming the code to statement$_i$; atomic {skip;progress: statement$_j$} moves progress from s to the following composite transition. Fig. 6.7 shows the difference in the labeled Kripke structures. atomic{...} guarantees that the progress state is left immediately after it was entered (cf. Subsec. 3.4.3). Unfortunately, SPIN does not interleave atomic sequences with the never claim process, so this technique cannot be used for SPIN's NPC checks. Trying nevertheless, SPIN sometimes claims to

find an NPC, but returns a trace which has a progress cycle; at other times, SPIN gives the warning that a "`progress label inside atomic - is invisible`". SPIN's inconsequent warning suggests that it does not always detect progress labels inside atomic.

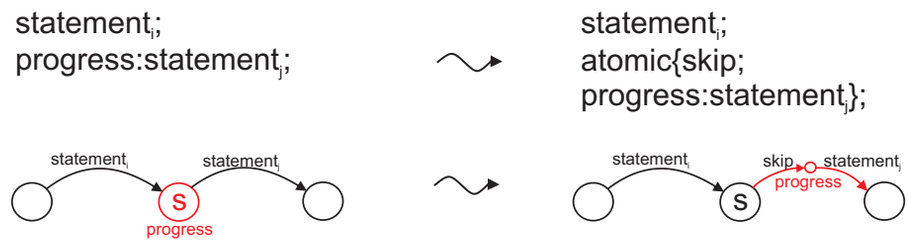


Figure 6.7.: Progress state (left) and progress transition without changing PROMELA's semantics (right)

6.5.3. DFS$_{\text{FIFO}}$

Adapting DFS$_{\text{FIFO}}$ to progress transitions only requires to change two polymorphic methods of DFS$_{\text{prune,NPC}}$ within the loop for each $s \xrightarrow{\alpha} t$ (cf. Def. 6.11):

- pruned(s, α, t): **return** $((s, \alpha, t) \in \mathfrak{T}^{\mathcal{P}})$;. So for this check, statement $\alpha \in L$ was added as parameter;
- np_cycle(t): **return true**;

The different versions of these polymorphic methods can be combined to handle both progress states and progress transitions simultaneously, which will be demonstrated in Sec. 6.7.

The biggest advantages of using progress transitions emerge in combination with POR, described in the next section.

6.6. Compatibility of DFS$_{\text{FIFO}}$ with POR

Most reduction methods for MC (cf. Sec. 5.4) are combinable with DFS$_{\text{FIFO}}$ without difficulty, for instance program slicing, statement merging, compressions, and symmetry reduction [Bosnacki et al., 2002; Faragó, 2007]. POR is one of the most powerful reduction methods, but its combination with DFS$_{\text{FIFO}}$ not trivial.

Roadmap. Subsec. 6.6.1 shows how relevant POR is for NPC checks. Subsec. 6.6.2 shows how POR has to be modified for DFS$_{\text{FIFO}}$ and its correctness. Finally, Subsec. 6.6.3 compares DFS$_{\text{FIFO}}$ with POR activated and NPC checks via LTL MC with POR activated and considers variations of POR provisos.

6.6.1. Motivation for Strong POR

POR's strength can decrease exponentially when changing from safety to liveness checks via LTL MC, caused by:

- the visibility proviso C2 (cf. Subsec. 5.4.1) becoming stricter;

- changing the exploration algorithm from a basic DFS to the NDFS (cf. Subsec. 5.3.2;
- with NDFS, the cycle proviso C3' is intricate: Since C3' depends on the stack, the set of states that innerDFS() with POR reaches also depends on the stack. So for correct POR, the NDFS must preserve information about the ample set selection between the NDFS's two phases, DFS() and innerDFS() (cf. [Holzmann et al., 1996, page 6]). This weakens POR – in many states up to full deactivation, and can cause several attempts to select a correct ample set for a state.

Example. Table 6.1 shows the runtime (in seconds), depth and number of states for a case study that verified a version of $leader_t$ that is highly optimized for MC (cf. [Faragó, 2007]): the DFS for safety checks with partial order reduction were performed in $O(|processes|^2)$ time and memory (i.e., easily up to SPIN's default limit of 255 processes), whereas the NPC checks via LTL MC could only be performed up to 6 processes. So a very critical aspect of NPC checks is how strong POR can perform.

Table 6.1.: Difference of DFS with POR and NPC checks via LTL MC with POR

|processes|	DFS			NPC checks via LTL MC		
	time	depth	states	time	depth	states
3	5"	33	66	5"	387	1400
4	5"	40	103	5"	2185	18716
5	5"	47	148	6"	30615	276779
6	5"	54	201	70"	335635	4.3e+06
7	5"	61	262	memory overflow (> 1GB)		
254	100"	1790	260353	memory overflow (> 1GB)		

6.6.2. Correctness of DFS_{FIFO} with POR

For DFS_{FIFO} with POR via ample sets (see Subsec. 5.4.1), the provisos C0, C1 and C2 can remain, since they can be checked by DFS_{FIFO} in each state s, independently of the path that the state space exploration took to reach s. So the different order in which states are traversed by DFS_{FIFO} compared to the basic DFS is irrelevant.

We investigate C2 closer in relation to Sec. 6.5; although C2 is general enough to cover both progress state semantics and progress transition semantics, we name and investigate the proviso separately for both semantics: C2 for progress transition semantics can directly be translated to $C2^{\mathfrak{T}}$ (cf. Table 6.2). Progress transition semantics and $C2^{\mathfrak{T}}$ correctly differentiates fake progress cycles from real progress cycles. C2 for progress state semantics considers alternations in np_ and hence makes no constraints for all-progress cycles, i.e., cycles that only contains progress states: all states on an all-progress cycle are invisible since np_ does not swap its value between successive progress states. For our adaption of POR (cf. Lemma 6.20), this will be too weak to guarantee completeness. Thus we will not use C2 for progress state semantics, but $C2^S$ (cf. Table 6.2). $C2^S$ is stricter than $C2^{transparent}$ (cf. Note 5.14) since it additionally constraints all-progress cycles visible. As trade-off for completeness, POR becomes weaker with $C2^S$.

Table 6.2.: Visibility provisos for DFS_{FIFO}

$\mathbf{C2^{\mathfrak{T}}}$	$\forall s \in S_{\text{POR}} : \mathit{ample}(s) \neq \mathit{enabled}(s) \Rightarrow \forall \alpha \in \mathit{ample}(s) : \alpha \notin \mathfrak{T}^{\mathcal{P}}$
$\mathbf{C2^{S}}$	$\forall s \in S_{\text{POR}} : \mathit{ample}(s) \neq \mathit{enabled}(s) \Rightarrow s \notin S^{\mathcal{P}}$

C3' in isolation no longer implies C3: Since a large part of the stack gets lost by DFS_{FIFO} postponing the traversal at progress, the premise of C3' (that $\alpha(s)$ is in the DFS stack) no longer holds for progress cycles, so the conclusion of C3' needs to be guaranteed otherwise. Fortunately, our visibility provisos already enforce that all pending transitions are enabled when we are about to destroy the stack by making progress, as Lemma 6.20 and its proof show.

Lemma 6.20. *For DFS_{FIFO}, (C1 and $C2^{S}$) $\Longrightarrow$ C3, as well as (C1 and $C2^{\mathfrak{T}}$) $\Longrightarrow$ C3.*

Proof. Let $s_0 \in S_{\text{POR}}$, cycle $\pi = (s_i)_{i\in[0,\ldots,n]} \in \mathit{paths}(\mathcal{S}_{\text{POR}}, s_0)$ and $\alpha \in \mathit{enabled}(s_0)$. If π is an NPC, the proof of Lemma 6.10 shows that DFS_{FIFO} finds some NPC in $\mathcal{S}_{\text{POR}}$ and terminates. If π contains some progress $s_j \xrightarrow{\beta} s_{j'}$ with $j \in [0,\ldots,n]$ and $j' := (j+1) \mathit{mod} (n+1)$, then either $\beta \in \mathfrak{T}^{\mathcal{P}}$ or $s_j \in S^{\mathcal{P}}$, so $\text{C2}^{\mathfrak{T}}$ or C2^{S} guarantee that $\mathit{enabled}(s_j) = \mathit{ample}(s_j)$. Thus C1 for $i \in [0,\ldots,j]$ guarantees that for some $h \in [0,\ldots,j] : \alpha \in \mathit{ample}(s_h)$. □

Theorem 6.21. *Theorem 6.13 also holds for DFS_{FIFO} with POR (i.e., via C0, C1, ($C2^{\mathfrak{T}}$ or $C2^{S}$)).*

Proof. DFS_{FIFO} with POR is **sound**: POR does not create new NPCs. Thus DFS_{FIFO} still does not output false negatives.

DFS_{FIFO} with POR is **complete**: Because of Lemma 6.20, POR via C0, C1, ($\text{C2}^{\mathfrak{T}}$ or C2^{S}) for NPC checks implies (C2 or $\text{C2}^{transparent}$) and C3, so all provisos of POR for NPC checks via LTL MC are met. Furthermore, all C0, C1, $\text{C2}^{\mathfrak{T}}$ and C2^{S} are independent of the path π used to reach s, so the order in which states are traversed is irrelevant. Therefore, the lemma of [Holzmann and Peled, 1994, page 6,7] can be applied for DFS_{FIFO}, too, showing that stutter equivalence related to progress is retained. Therefore, $\mathcal{S}_{\text{POR}}$ has an NPC iff the original $\mathcal{S}$ has one. Theorem 6.13 can hence be applied to $\mathcal{S}_{\text{POR}}$.

DFS_{FIFO} finds an **earliest NPC w.r.t. progress**: Let there be an NPC in the state space, found at the smallest depth w.r.t. progress, b_0. If $b < b_0$, all paths leading to an NPC are pruned before the NPC is reached. The completeness proof above can be applied to G_{b_0} instead of the complete $\mathcal{S}$. Thus $(G_{b_o})_{\text{POR}}$ also has an NPC, to which Theorem 6.13 can be applied. □

6.6.3. Comparing LTL NPC checks with POR to DFS_{FIFO} with POR

Adapting POR for DFS_{FIFO}, the constraint C3' becomes unnecessary. Subsec. 6.6.1 has shown that POR with C3' is very intricate. Therefore, eliminating C3' makes POR easier, stronger, and faster, which is investigated here theoretically and measured empirically in Sec. 6.8.

DFS_{FIFO} and its POR show that there are subclasses of LTL liveness properties, i.e., livelocks (more generally: persistence properties) for which no cycle proviso (such as C3 or C3') is necessary. Thus the simpler, locally checkable provisos C0, C1 and ($C2^{\mathfrak{T}}$ or $C2^{S}$) are sufficient to avoid the ignoring problem and render POR correct.

Furthermore, using progress transition semantics not only simplifies modeling and DFS_{FIFO}, but also POR: The visible transitions are exactly the progress transitions, facilitating the visibility proviso. Furthermore, only one of the originally two transitions (swapping np_ back and forth) is now visible, hence fewer full expansions are necessary compared to the original C2. This is a special case of transparent POR and of guard-based POR (cf. Subsec. 5.4.1). It can dramatically improve POR [Faragó and Schmitt, 2009; Siegel, 2012].

The restrictions on POR by the visibility proviso are also reduced by lazy progress traversal, which postpones progress traversal as long as possible, i.e., the proviso does not force fully expand states as often.

Notes 6.22. We can also compare our DFS_{FIFO} with SPIN's former NPC check, which used a specialized NDFS algorithm (cf. Subsec. 6.4.3 or [Holzmann, 1992]). [Holzmann et al., 1996] explains that this algorithm is not compatible with POR because of condition C3. The authors of the paper "do not know how to modify the algorithm for compatibility with POR" and NPC checks via LTL MC as alternative. But DFS_{FIFO} can be regarded as such modification of SPIN's old NPC check: the state space creation and search for an NPC are performed simultaneously, whereby C3 is reduced to C2.

The lazy progress traversal algorithm from [Valmari, 1993] (see end of Subsec. 6.4.3) has weak partial order reduction: It uses a stubborn set variant of POR that is not adapted to the special case of livelock detection. It combines livelock detection with other property detections, like deadlocks, such that at least one property detection is preserved, but it does not guarantee that every kind of property detection is preserved [Valmari, 1993]. Furthermore, the reduction requires (a DFS) to find strongly connected components or a reachability closure, which can be time consuming.

$C2^{S}$ weakens POR much more than $C2^{\mathfrak{T}}$ since a path may contain many progress states but rarely perform actual progress, i.e., not have many progress transitions. Thus $C2^{S}$ causes many unnecessary full expansions. Table 6.5 on page 147 shows that this can have a large effect. In (the rare) case a path has many successive progress states due to actual progress, i.e., different local progress states, the original POR with C2 and progress state semantics might outperform $C2^{\mathfrak{T}}$ with progress transition semantics since np_ does not swap its value between successive progress states, thus avoiding full expansions. This inspires **lazy full expansion due to progress**, with the corresponding provisos given in Table 6.3. The check with hash_table avoids invisible all-progress cycles, but complicates POR. $C2^{\mathfrak{T}}_{\text{lazy}}$ and $C2^{S}_{\text{lazy}}$ can probably be implemented and optimized by coupling the provisos to the exploration algorithm [Evangelista and Pajault, 2010], but this is future work.

6.7. Parallel DFS_{FIFO}

As described in Sec. 3.5, programs must be parallelized to make use of current advances of CPUs, which is also the trend in formal methods. This section focuses on the algorithmic

Table 6.3.: Visibility provisos for lazy full expansion

$\mathbf{C2}^{\mathfrak{T}}_{\mathbf{lazy}}$	$\forall s \in S_{\text{POR}} : \mathit{ample}(s) \neq \mathit{enabled}(s) \Rightarrow \forall \alpha \in \mathit{ample}(s) : (\alpha \notin \mathfrak{T}^{\mathcal{P}}$ or $\mathit{enabled}(\alpha(s)) \subseteq \mathfrak{T}^{\mathcal{P}}$ and $\alpha(s) \notin$ hash_table$)$
$\mathbf{C2}^{\boldsymbol{S}}_{\mathbf{lazy}}$	$\forall s \in S_{\text{POR}} : \mathit{ample}(s) \neq \mathit{enabled}(s) \Rightarrow s \notin S^{\mathcal{P}}$ or $(\forall \alpha \in \mathit{ample}(s) :$ $\alpha(s) \in S^{\mathcal{P}}$ and $\alpha(s) \notin$ hash_table$)$

aspects to parallelize DFS_{FIFO}, for which multi-threading was chosen since DFS_{FIFO} has to handle a large amount of global communication with higher performance, and we can reuse data structures and concepts of LTSMIN (cf. Subsec. 5.5.2). Thus it covers general concurrency aspects and data structures only as much as necessary, but gives sufficient references for further reading. The main reference on multiprocessor and multi-threaded programming is [Herlihy and Shavit, 2008], on concurrent implementations for LTL MC and their data structures is [Laarman, 2014].

Roadmap. Subsec. 6.7.1 introduces $\text{PDFS}_{\text{FIFO}}$, a parallel version of DFS_{FIFO}, and enumerates the differences to DFS_{FIFO}. Subsec. 6.7.2 proves $\text{PDFS}_{\text{FIFO}}$ correct. Subsec. 6.7.3 gives some implementation details of $\text{PDFS}_{\text{FIFO}}$. Finally, Subsec. 6.7.4 compares $\text{PDFS}_{\text{FIFO}}$ with parallel LTL MC, considering mainly scalability aspects.

6.7.1. The Algorithm $\text{PDFS}_{\text{FIFO}}$

Listing 6.7 presents a parallel version of DFS_{FIFO}, $\mathbf{PDFS_{FIFO}}$. The algorithm does not differ much from Listing 6.5:

$\text{DFS}_{\text{prune,NPC}}(S_{\to^*}\ s)$ is split into parallel $\text{DFS}_{\text{prune,NPC}}(S_{\to^*}\ s$, int workerNumber), handling states from fifo concurrently. The technique to parallelize the $\text{DFS}_{\text{prune,NPC}}(s, w)$ calls is based on shared state storage as in the successful multi-core NDFS algorithms [Laarman et al., 2011c; Laarman and van de Pol, 2011; Evangelista et al., 2012]. Each worker thread $w \in [1, \ldots, P]$ uses a local stack$[w]$, while hash_table is shared. Here fifo is shared, but later it is partially localized for lower contention (cf. Subsec. 6.7.3). The local stacks may overlap (see l.36 and l.27), but eventually diverge because we use a randomly ordered next-state function, $\mathit{enabled}_w(\cdot)$ (see l.7).

Instead of adding a state s to hash_table at the beginning of $\text{DFS}_{\text{prune,NPC}}(s, w)$, it is added just before backtracking (see l.21). Therefore, only when $\text{DFS}_{\text{prune,NPC}}(s, w)$ has finished exploration, l.12 forbids worker threads to visit s. Beforehand, $\text{DFS}_{\text{prune,NPC}}(s, w')$ for $w' \neq w$ may be called, such that worker thread w' can help by undertaking part of the exploration of $\mathit{dest}(s, \to^*)$, resulting in better parallel scalability.

$\text{DFS}_{\text{FIFO}}(w)$ calls $\text{DFS}_{\text{prune,NPC}}(s, w)$ for a state s from fifo iff $s \notin$ hash_table (cf. Note 6.12).

As announced in Sec. 6.5, $\text{PDFS}_{\text{FIFO}}$ handles both progress states and progress transitions simultaneously, so $\mathcal{P} = S^{\mathcal{P}} \dot{\cup} \mathfrak{T}^{\mathcal{P}}$. Consequently, $\forall s \in \text{fifo} \backslash \{\mathit{init}\} : s \in S^{\mathcal{P}}$ or s was reached via $s' \xrightarrow{\alpha} s$ with $\alpha \in \mathfrak{T}^{\mathcal{P}}$. We call these states collectively **after-progress states**.

```
HashTable hash_table := new HashTable();                                    1
FIFO fifo := new FIFO();                                                    2
Stack[] stack;                                                              3
                                                                            4
proc DFS_prune,NPC(S_→* s, int w)                                           5
  push s onto stack[w];                                                     6
  for each α ∈ enabled_w(s) do                                              7
    t := α(s);                                                              8
    if (t ∈ stack[w] && α ∉ P && t ∉ P)                                     9
    then halt with error_message()                                          10
    fi;                                                                     11
    if (t ∉ hash_table)                                                     12
    then if (α ∉ P && t ∉ P)                                                13
         then DFS_prune,NPC(t,w)                                            14
         else if (t ∉ fifo)                                                 15
              then put t in fifo                                            16
              fi                                                            17
         fi                                                                 18
    fi                                                                      19
  od;                                                                       20
  add s to hash_table;                                                      21
  pop s from stack[w];                                                      22
end;                                                                        23
                                                                            24
proc DFS_FIFO(int workerNumber)                                             25
  while (fifo not empty) do                                                 26
    peek some s in fifo;                                                    27
    if (s ∉ hash_table)                                                     28
    then DFS_prune,NPC(s,workerNumber);                                     29
    fi;                                                                     30
    remove s from fifo;                                                     31
  od;                                                                       32
end;                                                                        33
                                                                            34
proc main(int P)                                                            35
  put init in fifo;                                                         36
  stack = new Stack[P];                                                     37
  stack[i] = new Stack() for all i ∈ [1,...,P];                             38
  DFS_FIFO(1) || ... || DFS_FIFO(P);                                        39
  printf("structure does not contain NPCs");                                40
end;                                                                        41
```

Listing 6.7: Typed parallel $\mathrm{DFS}_{\mathrm{FIFO}}$ ($\mathrm{PDFS}_{\mathrm{FIFO}}$)

If an NPC is found, l.10 calls error_message() and halts the algorithm, but the callee does not return.

Note 6.23. A state $s \in$ fifo might at any time be also added to hash_table (cf. Note 6.14) by some worker thread in two cases:

- some worker thread w takes s from fifo at l.27, calls $\mathrm{DFS}_{\mathrm{prune,NPC}}(s, w)$ and completes the call;
- the after progress state s is not in $\mathcal{S}^{\mathcal{P}}$ and some worker thread w reaches s also via some non-progress transition (see l.13), calls $\mathrm{DFS}_{\mathrm{prune,NPC}}(s, w)$ and completes the call (e.g., for $s \xrightarrow{\alpha} s$ with $\alpha \in \mathfrak{T}^{\mathcal{P}}$).

6.7.2. Correctness of $\mathrm{PDFS}_{\mathrm{FIFO}}$

Theorem 6.31 proves correctness of $\mathrm{PDFS}_{\mathrm{FIFO}}$, using the following lemmas: Lemmas 6.25, 6.27 and 6.29 use induction on $\mathrm{PDFS}_{\mathrm{FIFO}}$'s execution steps: They show that the respective induction hypothesis holds after initialization, and that it is maintained by execution of each statement of Listing 6.7. But only the statements that influence the induction hypothesis are considered. Rather than restricting progress to either transitions or states, we prove $\mathrm{PDFS}_{\mathrm{FIFO}}$ correct under $\mathcal{P} = \mathcal{S}^{\mathcal{P}} \cup \mathfrak{T}^{\mathcal{P}}$.

We define *NPC* as the set of states on NPCs: $NPC := \{s \in S \mid \exists \pi \in paths(\mathcal{S}, s) : \pi \cap \mathcal{P} = \varnothing \text{ and } \pi \text{ is a cycle}\}$.

Lemma 6.24. *Upon return of* $DFS_{prune,NPC}(s, i)$*, s is explored: $s \in$ hash_table.*

Proof. l.21 of $\mathrm{DFS}_{\mathrm{prune,NPC}}(s, i)$ adds s to hash_table. □

Lemma 6.25. *Invariantly, all direct successors of an explored state e are explored or in fifo: $\forall e \in$ hash_table $\forall f \in dest(e, \rightarrow) : f \in$ hash_table $\cup$ fifo.*

Proof. The proof uses induction on $\mathrm{PDFS}_{\mathrm{FIFO}}$'s execution steps. After initialization, the invariant holds trivially, as hash_table is empty. hash_table is only modified at l.21, where e is added by a worker thread w after all of e's immediate successors f are considered at l.9–l.16: If already $f \in$ hash_table $\cup$ fifo, we are done. Otherwise, $\mathrm{DFS}_{\mathrm{prune,NPC}}(e, w)$ terminates at l.10 or f was added to hash_table at l.14 (cf. Lemma 6.24) or to fifo at l.16. States are removed from fifo at l.31, but only after being added to hash_table at l.29 (cf. Lemma 6.24). □

Lemma 6.26. *Invariantly, all paths from an explored state e to a state f that is in fifo, but not yet explored, contain progress: $\forall f \in$ fifo$\setminus$ hash_table $\forall e \in$ hash_table $\forall \pi \in paths(\mathcal{S}, e)$ with $dest(\pi) = f : \pi \cap \mathcal{P} \neq \varnothing$.*

Proof. Let $\pi = (s_{i-1} \xrightarrow{l_i} s_i)_{i \in [1,\ldots,n]}$. By induction over π and Lemma 6.25, we obtain either the contradiction $s_n \in$ hash_table, or some $i \in [0, \ldots, n-1]$ with $\forall j \in [0, \ldots, i] : s_j \in$ hash_table and $s_{i+1} \in$ fifo$\setminus$ hash_table. By Lemma 6.24 and l.12–l.14 of $\mathrm{DFS}_{\mathrm{prune,NPC}}(s_i, w)$ for the worker thread w who has added s_i to hash_table, $s_i \xrightarrow{l_{i+1}} s_{i+1}$ makes progress. □

Lemma 6.27. *Invariantly, explored states are not on NPCs: hash_table$\cap NPC = \varnothing$.*

Proof. Initially, hash_table$= \varnothing$ and the lemma holds trivially. Let $e \in$hash_table and i be the first worker thread that added e to hash_table in $\mathrm{DFS}_{\mathrm{prune,NPC}}(e, i)$ at l.21. Assume that $e \in NPC$ after l.21 of $\mathrm{DFS}_{\mathrm{prune,NPC}}(e, i)$. Then there is an NPC in $e \to f \to^+ e$ with $e \neq f$, since otherwise l.10 would have reported an NPC. Now by Lemma 6.25, $f \in$hash_table$\cup$ fifo. By the induction hypothesis, $f \notin$hash_table, so $f \in$ fifo$\setminus$hash_table. Lemma 6.26 contradicts $e \to f$ making no progress. □

Lemma 6.28. *Upon normal termination of $PDFS_{FIFO}$, all reachable states have been explored: $S_{\to^*} =$ hash_table.*

Proof. After some worker thread w normally terminates from $\mathrm{DFS}_{\mathrm{FIFO}}(w)$, fifo$= \varnothing$ by l.26. By l.36, l.29 and Lemma 6.24, *init* $\in$hash_table. So by Lemma 6.25, $S_{\to^*} =$hash_table. □

Lemma 6.29. *$PDFS_{FIFO}$ terminates and either reports an NPC or "structure does not contain NPCs".*

Proof. Upon normal return of a call $\mathrm{DFS}_{\mathrm{prune,NPC}}(s, w)$ for some worker thread w and some $s \in$ fifo at l.29, w has put s in the hash_table (cf. Lemma 6.24), removed s from fifo at l.31, and will never add s to fifo again (by l.12). Since elements are never removed from hash_table, it grows monotonically, but is bounded by $|S_{\to^*}|$, and eventually fifo $= \varnothing$. Thus eventually $\mathrm{PDFS}_{\mathrm{FIFO}}$ either terminates abruptly after reporting an NPC, or all $\mathrm{DFS}_{\mathrm{FIFO}}(1), \ldots,$ $\mathrm{DFS}_{\mathrm{FIFO}}(P)$ terminate, and $\mathrm{PDFS}_{\mathrm{FIFO}}$ terminates normally after reporting "structure does not contain NPCs" at l.40. □

Lemma 6.30. *Invariantly, for some worker thread w, the states in stack[w] form a path that makes no progress except for possibly the first state: stack[w]$= \varnothing$ or stack[w]$= (s_i)_{i \in [0,\ldots,n]} \in paths(\mathcal{S}, s_0)$ and $(s_i)_i \cap \mathcal{P} \subseteq \{s_0\}$.*

Proof. By induction over the recursive $\mathrm{DFS}_{\mathrm{prune,NPC}}(s, w)$ calls, stack$[w] \in paths(\mathcal{S}, s_0)$. At l.14, we have $t \notin \mathcal{P}$ and $\alpha \notin \mathcal{P}$, but at l.29 with stack$[w]= \varnothing$ we may have $s \in \mathcal{S}^{\mathcal{P}}$ (by l.16). □

Theorem 6.31. *$S_{\to^*} \cap NPC \neq \varnothing \Leftrightarrow PDFS_{FIFO}$ reports an NPC.*

Proof. If $\mathrm{DFS}_{\mathrm{prune,NPC}}(s, w)$ reports an NPC, then $s \xrightarrow{\alpha} t \cdot$stack$[w]$ is an NPC by l.9 and Lemma 6.30.

If no $\mathrm{DFS}_{\mathrm{prune,NPC}}(s, w)$ reports an NPC, then $\mathrm{PDFS}_{\mathrm{FIFO}}$ reports "structure does not contain NPCs" by Lemma 6.29. Therefore, at l.40, $S_{\to^*} =$ hash_table by Lemma 6.28. Thus, $S_{\to^*} \cap NPC = \varnothing$ by Lemma 6.27. □

6.7.3. Implementation of $\mathrm{PDFS}_{\mathrm{FIFO}}$

$\mathrm{PDFS}_{\mathrm{FIFO}}$ has been implemented by Alfons Laarman in C as algorithm back-end in LTSMIN Version 2.0.

Lock-free Hash Table. To lower contention and thus improve parallel scalability, threads communicate via hash_table that is maintained in shared memory, using a lock-free (also called lockless) design via CAS operations, see Def. 6.32, Sec. 3.5 and Subsec. 5.5.2.

Definition 6.32. Let $\mathcal{P}$ be a multi-threaded program using shared memory, and progress be defined according to the contracts of the methods $\mathcal{P}$ calls. Then:

- $\mathcal{P}$ is **lock-free** $:\Leftrightarrow$ recurrently some thread makes progress;
- $\mathcal{P}$ is **wait-free** $:\Leftrightarrow$ all threads recurrently make progress.

Notes. Thus, $\mathcal{P}$ is lock-free iff the whole system $\mathcal{P}$ is free of deadlocks and livelocks, and $\mathcal{P}$ is wait-free iff each thread is free of deadlocks and livelocks.

Def. 6.32 shows that wait-freedom of $\mathcal{P}$ implies lock-freedom of $\mathcal{P}$.

Usually, progress corresponds to the termination of method calls [Herlihy and Shavit, 2008]. But relating progress to the methods' contracts is more general, e.g., for data structures that have operations whose contracts allow blocking, for instance pop on an empty stack.

A data structure D is called lock-free (respectively wait-free) if the multi-threaded program that reads from or writes to D is lock-free (respectively wait-free).

Already [Herlihy, 1988] has shown that for all algorithms, there exist wait-free implementations, using universal constructors, which transform sequential code into wait-free concurrent implementations. Latency and throughput of such wait-free implementations have been much worse than of lock-free or even blocking implementations [Fich et al., 2005; Herlihy and Shavit, 2008]: the employed helping pattern [Herlihy et al., 2003], which makes threads help each other to progress, costs many expensive synchronization primitives (e.g., CASs). Fortunately, [Kogan and Petrank, 2012] introduced a new methodology, called fast-path-slow-path, which achieves much better latency by using the expensive concurrent operations only in rare cases, as fall-back to achieve wait-freedom.

Differentiating performance aspects shows that the lock-free hash table is nevertheless more suitable for DFS_{FIFO} than a wait-free hash table: Both lock-freedom and wait-freedom guarantee system-wide throughput and stability from crash failures (i.e., continuously failing to return output [Cristian, 1993]). Wait-freedom additionally guarantees starvation-freedom and **real-time behavior**, i.e., it has a lower worst case latency. With wait-freedom and lower worst case latency, a thread in DFS_{FIFO} cannot fall far behind, reducing

- how much work of the slow thread are redone by other threads, hence reducing work duplication;
- the amount of the state space being explored late, as well as the delay, hence reducing the search depth and the length of counterexamples (cf. Note 6.33 below).

But for model checking algorithms, throughput is more important than latency because of the huge amount of data caused by state space explosion (cf. Table 6.1). Since communication is the most constraining factor for throughput [Lewis and Berg, 2000], the focus is on low communication cost. This is best achieved by the lock-free hash table design from LTSMIN, since it requires few synchronization primitives, is cache oriented, and can be implemented pointerlessly (cf. Subsec. 5.5.2 and [Laarman et al., 2010, 2011a; Laarman, 2014]). As result, the lock-free hash table has

- similar behavior and some similar concepts as the well-known wait-free hash table of [Click, 2007];
- very little false sharing;
- little difference in throughput and latency compared to the wait-free hash table of [Click, 2007]. For model checking, the wait-free hash table is slightly slower [Laarman, 2014].

Load Balancing. For fast concurrent inclusion checks and enumeration, fifo is maintained as both queue and in hash_table (using one additional bit per state to distinguish fifo from original entries). To also reduce contention for the queue [Sutter, 2009; Laarman et al., 2010; Scogland and chun Feng, 2015], it is split into P local queues fifo$[w]$ for each worker thread w, calling for the load balancing described below. Since each stack$[w]$ remains much smaller than the other data structures, we maintain it as local hash table, and construct counterexamples using back-links (cf. Subsec. 6.4.2), which are maintained by LTSmin anyways when needed.

Our first implementation of load balancing (cf. Subsec. 5.5.2) simply relaxed the constraint of $\text{DFS}_{\text{prune,NPC}}(s, w)$ at l.15 to $t \notin$ fifo$[w]$, so that the after-progress state t may end up in multiple local queues (cf. Note 6.12). This already provided good work distribution, since $\mathcal{A}_{\mathcal{S}}$ is usually sufficiently connected. But the total size of all local queues grew proportional to P, wasting a lot of memory on many cores. Therefore, we prefer explicit load balancing via **work stealing** instead, as depicted in Listing 6.8: steal(fifo$[w]$) returns **true** if fifo$[w]$ is not empty; otherwise, it tries to steal states randomly from another local queue, returning **true** iff successful. So if steal(fifo$[w]$) returns **false**, $\text{DFS}_{\text{FIFO}}(w)$ **detects termination**: it tried to steal but did not succeed for any local queue, so exploration of $\mathcal{S}$ has finished, and $\text{DFS}_{\text{FIFO}}(w)$ also terminates.

```
proc DFS_FIFO(int w)                                    1
  put init in fifo[w];                                  2
  while (steal(fifo[w])) do                             3
    get and remove s from fifo[w];                      4
    if (s ∉ hash_table)                                 5
    then DFS_prune,NPC(s,w);                            6
    fi;                                                 7
  od;                                                   8
end;                                                    9
```

Listing 6.8: Work stealing for $\text{PDFS}_{\text{FIFO}}$

Note 6.33. The proofs show that correctness of $\text{PDFS}_{\text{FIFO}}$ does not require a strict FIFO order on after-progress states in the queues, i.e., a strict BFS over the after-progress states is not required. Thus l.27 of Listing 6.7 does not say whether the order of the FIFO is preserved. Instead, LTSmin's command line option --strict turns on strict FIFO order in $\text{PDFS}_{\text{FIFO}}$ (**strict $\text{PDFS}_{\text{FIFO}}$** for short), using synchronization between the BFS levels (cf. Note 6.14 and [Dalsgaard et al., 2012]). This guarantees shortest counterexamples w.r.t. progress and low memory use of the FIFO. Without --strict, synchronization is dropped to optimize parallel scalability.

6.7.4. Comparing Parallel LTL NPC Checks to $\text{PDFS}_{\text{FIFO}}$

As for the sequential case (cf. Subsec. 6.4.3), we again compare the two approaches of livelock detection theoretically: parallel NPC checks via LTL MC and $\text{PDFS}_{\text{FIFO}}$. Practical experiments will be presented in Sec. 6.8.

LTL MC is probably not efficiently parallelizable: parallelization raises the worst case time complexity of cycle detection from $O(|\mathcal{S}_{\rightarrow^*}|)$ (cf. Subsec. 5.3.2) to either $O(|\mathcal{S}_{\rightarrow^*}| \cdot P)$ using CNDFS (cf. Subsec. 5.5.2 or [Evangelista et al., 2012]) or to $O(|\mathcal{S}_{\rightarrow^*}|^2)$ using OWCTY (cf. Subsec. 5.5.3 or [Barnat et al., 2009]). Since livelocks are in the class of weak LTL properties, OWCTY with heuristics can solve them in linear worst case runtime (cf. Subsec. 5.5.3): the algorithm explores each state at most twice, in two searches over the state space without any work duplication. Since the algorithm is based on state partitioning [Cerná and Pelánek, 2003], it has high communication and contention, which restricts parallel speedup [Sutter, 2009], resulting in logarithmic speedup for DiVinE [Barnat et al., 2009]. Experiments with CNDFS [Evangelista et al., 2012] demonstrated that its parallelization techniques make the state of the art for LTL MC; thus we used similar techniques for $\text{PDFS}_{\text{FIFO}}$ (cf. Subsec. 6.7.3). Since $\text{PDFS}_{\text{FIFO}}$ has BFS behavior on after-progress states, two different worker threads are likely to start $\text{DFS}_{\text{prune,NPC}}$ from different after-progress states. So we can expect more work pruning between different worker threads and less work duplication, resulting in better parallel scalability than CNDFS.

Furthermore, CNDFS needs additional synchronization to prevent workers from **early backtracking** [Laarman et al., 2011c]: a situation in which two workers exclude a third from part of the state space. Figure 6.8 illustrates this: Worker 1 can visit s, v, t and u, and then halt. Worker 2 can visit s, u, t and v and backtrack over v. If now Worker 1 resumes and backtracks over u, both v and u are in hash_table. A third worker will be excluded from exploring t, which might lead to a large part of the state space. Lemma 6.25 shows that this is impossible for $\text{PDFS}_{\text{FIFO}}$ because the successors of visited states are in hash_table or in fifo (but never solely in some stack$[w]$, as in CNDFS).

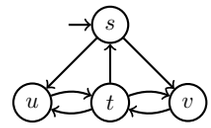


Figure 6.8.: Early backtracking

Finally, POR and parallelization of NDFS are not efficiently combinable due to the DFS order required by C3 and C3' (cf. Sec. 5.5 and [Barnat et al., 2010]).

6.8. Experiments

We benchmark the performance of our DFS_{FIFO} for the first time, using LTSMIN Version 2.0 (cf. Subsec. 5.5.2). This is currently the only implementation of DFS_{FIFO} and $\text{PDFS}_{\text{FIFO}}$ we know of. It uses work stealing and offers both strict and non-strict BFS order (cf. Note 6.33). We also use the $\text{PDFS}_{\text{FIFO}}$ algorithm for the sequential case. Since the next-state function is randomly ordered, $\text{PDFS}_{\text{FIFO}}$ is a randomized algo-

rithm. Since the results usually vary little, we took the average over 5 runs for each relevant experiment.

We benchmarked on a 48-core machine (a four-way AMD Opteron 6168) with 128GB of main memory, and considered four established, publicly available [URL:PromelaDatabase] PROMELA models with progress labels: leader$_t$, garp, giop and i-prot (see Subsec. 6.1). We adapted SPINS (LTSMIN's language front-ends for PROMELA, cf. Subsec. 5.5.2) to interpret the labels as either progress states, as in SPIN, or progress transitions. For all these models, the livelock property holds under $\mathcal{P} = \mathcal{S}^{\mathcal{P}}$ and $\mathcal{P} = \mathfrak{T}^{\mathcal{P}}$. Models that we modified are available at [URL:leader4DFSFIFOHP]. For fair comparison, we use SPINS and DVE, and ensure similar state counts by turning off control-flow optimizations in SPIN and SPINS. This is necessary because SPIN has a more powerful optimizer, which can be, but is not yet implemented in SPINS. Only one of our models, giop, still yields a larger state count in SPINS than in SPIN. Consequently, all measurements for giop that depend on $|S_{\rightarrow^*}|$ cannot be compared sensibly between LTSMIN and SPIN. We still include giop in our tables as it nicely features the benefits of DFS$_{\text{FIFO}}$ compared to LTSMIN's NDFS. All measurements in the tables and figures of this section have already been published in [Laarman and Faragó, 2013].

Roadmap. Our benchmarks cover both the sequential and parallel case, progress states and progress transitions, with strict and non-strict BFS order, without and with POR. Subsec. 6.8.1 compares the time and space requirements of DFS$_{\text{FIFO}}$ and NPC checks via LTL MC using LTSMIN and SPIN. Subsec. 6.8.2 shows the strength of POR for NPC checks via LTL MC using LTSMIN and SPIN, in relation to the space requirements from the previous subsection. Subsec. 6.8.3 investigates the parallel time requirements and scalability of PDFS$_{\text{FIFO}}$, and compares the results against the multi-core NDFS algorithm CNDFS, the state of the art for parallel LTL MC (cf. Subsec. 5.5.2 or [Evangelista et al., 2012]), and the piggyback algorithm in SPIN (**PB**) [Holzmann, 2012]. Subsec. 6.8.4 investigates the parallel space requirements for DFS$_{\text{FIFO}}$ and compares it to CNDFS's and shortly to PB's. Subsec.6.8.5 considers the combination of parallelism and POR, and compares PDFS$_{\text{FIFO}}$'s space and time requirements and speedups with those available for CNDFS and those of DiVinE's **owcty** [Barnat et al., 2009], which is also able to use POR for parallel MC. Finally, Subsec. 6.8.6 benchmarks on-the-flyness for PDFS$_{\text{FIFO}}$ and CNDFS and compares the results.

6.8.1. DFS$_{\text{FIFO}}$'s Performance

In theory, NPC checks via LTL MC require up to triple the time and double the space compared to DFS$_{\text{FIFO}}$ (cf. Subsec. 6.4.3). To verify this, we compare DFS$_{\text{FIFO}}$ to NDFS in LTSMIN and SPIN. In LTSMIN, we used the command line: `prom2lts-mc --state=tree -s28 --strategy=[dfsfifo/ndfs] [model]`, which replaces the shared table (for fifo and hash_table) by tree compression, to be able to cover more states in our experiments. Likewise, we used compression in SPIN as well (collapse compression): `cc -O2 -DNP -DNOFAIR -DNOREDUCE -DNOBOUNDCHECK -DCOLLAPSE -o pan pan.c`, and `pan -m100000 -l -w28`, avoiding table resizes and overhead. We write oom for runs that overflow the main memory. Similar to the theoretical comparisons in Subsec. 6.4.3, we also compare the performances to that of a basic DFS, using similar commands as above for both tools.

Table 6.4 shows the results: For both SPIN and LTSMIN, $|S^{\text{LTL}}_{\rightarrow *}|$ (the state space for NDFS) is 1.5 to 2 times as large as $|S_{\rightarrow *}|$ (the state space for DFS and DFS_{FIFO}). The factor can become smaller than 2 since $\mathcal{A}_{\Diamond\Box\text{np\_}}$ can prune $\mathcal{A}_{\mathcal{S}}$ (cf. Subsec. 5.3.2). giop fits in memory for DFS_{FIFO}, but $\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{\Diamond\Box\text{np\_}}$ overflows for NDFS with LTSMIN. T_{NDFS} is about 1.5 to 4.5 times larger than T_{DFS} for SPIN, 2 to 5 times larger for LTSMIN. The upward deviations from the theoretical factor 3 are probably caused by the increased number of execution steps per state (switching between inner and outer DFS, lockstepping between $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\Diamond\Box\text{np\_}}$, checking for progress and acceptance states), the downward deviations by $|S^{\text{LTL}}_{\rightarrow *}|$ being just 1.5 times as large as $|S_{\rightarrow *}|$. T_{NDFS} is 1.6 to 3.2 times as large as $T_{DFS_{\text{FIFO}}}$. The downward deviations from the theoretical factor 3 are probably caused by the previous deviations and by $T_{DFS_{\text{FIFO}}}$ being 1.5 to 2 times larger than T_{DFS}, likely caused by its set inclusion tests on stack and fifo.

Table 6.4.: Number of states and runtimes (sec) of (sequential) DFS, DFS_{FIFO}, NDFS in SPIN and LTSMIN

	LTSMIN					SPIN			
	$\lvert S_{\rightarrow *}\rvert$	$\lvert S^{\text{LTL}}_{\rightarrow *}\rvert$	T_{DFS}	$T_{DFS_{\text{FIFO}}}$	T_{NDFS}	$\lvert S_{\rightarrow *}\rvert$	$\lvert S^{\text{LTL}}_{\rightarrow *}\rvert$	T_{DFS}	T_{NDFS}
leader_t	4.5E7	198%	153.7	233.2	753.6	4.5E7	198%	304.0	1,390.0
garp	1.2E8	150%	377.1	591.2	969.2	1.2E8	146%	1,370.0	2,050.0
giop	2.7E9	oom	2.1E4	4.3E4	oom	8.4E7	181%	1,780.0	4,830.0
i-prot	1.4E7	140%	28.4	41.4	70.6	1.4E7	145%	63.3	103.0

6.8.2. Strength of POR

We extended LTSMIN's POR with the alternative provisos for DFS_{FIFO}, $\text{C2}^{\mathcal{S}}$ and $\text{C2}^{\mathfrak{T}}$ (cf. Table 6.2) and without C3'. Table 6.5 shows the reduction rate, i.e., the number of states relative to the corresponding $|S_{\rightarrow *}|$ or $|S^{\text{LTL}}_{\rightarrow *}|$ in Table 6.4, using the different algorithms in both tools: For all models, both LTSMIN and SPIN are able to obtain reductions of multiple orders of magnitude using their DFS algorithms. We also observe that much of this benefit disappears when using the NDFS algorithm, which is due to the cycle proviso: the reduction rate is over 6 to 300 times worse for LTSMIN, 1.2 to 38 times worse for SPIN, so often SPIN can retain a better reduction rate than LTSMIN. DFS_{FIFO} with progress states (column $DFS^{\mathcal{S}}_{\text{FIFO}}$) performs almost as poorly as NDFS; apparently, the $\text{C2}^{\mathcal{S}}$ proviso is so restrictive that many states are fully expanded (cf. Subsec. 6.6.3). But DFS_{FIFO} with progress transitions (column $DFS^{\mathfrak{T}}_{\text{FIFO}}$) retains DFS's impressive reduction rate with only a factor 1 to 2 larger state spaces, so the reduction rates are over 3 to over 200 times stronger than NDFS's reduction rates.

6.8.3. Parallel Runtime

To compare the parallel algorithms in LTSMIN, we use the options --threads=P --strategy=[dfsfifo/cndfs], where P is the number of worker threads, up to 48, the number of cores on our machine. In SPIN, we use -DBFS_PAR, which activates SPIN's multi-core BFS but also turns on lossy hashing [Holzmann, 2012], and run the pan binary with an

Table 6.5.: POR (%) for $DFS^{\mathfrak{T}}_{\text{FIFO}}$, $DFS^{\mathcal{S}}_{\text{FIFO}}$, DFS and NDFS in LTSMIN and SPIN

	LTSMIN				SPIN	
	DFS	$DFS^{\mathfrak{T}}_{\text{FIFO}}$	$DFS^{\mathcal{S}}_{\text{FIFO}}$	NDFS	DFS	NDFS$^{\text{SPIN}}$
leader$_t$	0.32%	0.49%	99.99%	99.99%	0.03%	1.15%
garp	1.90%	2.18%	4.29%	16.92%	10.56%	12.73%
giop	1.86%	1.86%	3.77%	oom	1.60%	2.42%
i-prot	16.14%	31.83%	100.00%	100.00%	24.01%	41.37%

option -uP. This turns on a parallel, linear-time, but incomplete, cycle detection algorithm called piggyback (PB) [Holzmann, 2012]. It might also be unsound and incomplete due its combination with lossy hashing [Barnat et al., 2012]. We chose SPIN's multi-core BFS since SPIN's multi-core DFS can "only take advantage of parallel execution on no more than two cpu-cores for liveness properties" [Holzmann, 2012].

Table 6.6 compares the runtimes of PDFS$_{\text{FIFO}}$, CNDFS, PB and again also DFS for comparison. giop causes oom errors for CNDFS. For the sequential case, DFS and DFS$_{\text{FIFO}}$ use the parallel implementations with $P = 1$. Thus their runtimes are the same as in Table 6.4. CNDFS, on the other hand, adds parallel overhead to NDFS, so the sequential runtime increases. Since PB is unsound and incomplete, it can be faster than SPIN's NDFS: PDFS$_{\text{FIFO}}$ is slightly slower than PB for leader$_t$, but about a factor 2 faster for garp and i-prot (and giop cannot be compared due to different $|S_{\rightarrow^*}|$). Thus, PDFS$_{\text{FIFO}}$ is the fastest algorithm for NPC checks for the sequential case, giving the following scalability analysis even stronger meaning (being the absolute speedup, not only the relative). Table 6.6 also includes runtimes for $P = 48$; since PB slows down after a certain amount of worker threads, we took the optimal number of worker threads instead of 48. PDFS$_{\text{FIFO}}$ is 3.4 to 9 times faster than CNDFS, 4.5 to 16 times faster than PB.

Table 6.6.: Runtimes (sec) for the parallel algorithms: DFS, PDFS$_{\text{FIFO}}$ and CNDFS in LTSMIN, and PB in SPIN

	DFS		PDFS$_{\text{FIFO}}$		CNDFS		PB	
	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{min}
leader$_t$	153.7	3.8	233.2	5.7	925.7	51.4	228.0	25.9
garp	377.1	8.8	591.2	13.1	1061.0	58.6	1180.0	70.9
giop	2.1E4	463.3	4.3E4	970	oom	oom	1200.0	57.8
i-prot	28.4	0.7	41.4	1.1	75.9	3.7	86.2	17.7

Figure 6.9 plots the obtained speedups: As expected, DFS's [Laarman et al., 2010] and PDFS$_{\text{FIFO}}$'s runtime (strongly) scale almost linearly, while CNDFS exhibits significant sub-linear scalability, even though it is the fastest parallel LTL solution [Evangelista et al., 2012]. PB's runtime also scales sub-linearly significantly.

6.8.4. Parallel Memory Use

As expected (similarly to Note 6.23), few duplications of locally stored states occur: Analyzing revisits of states by adding counters in the source code showed that they

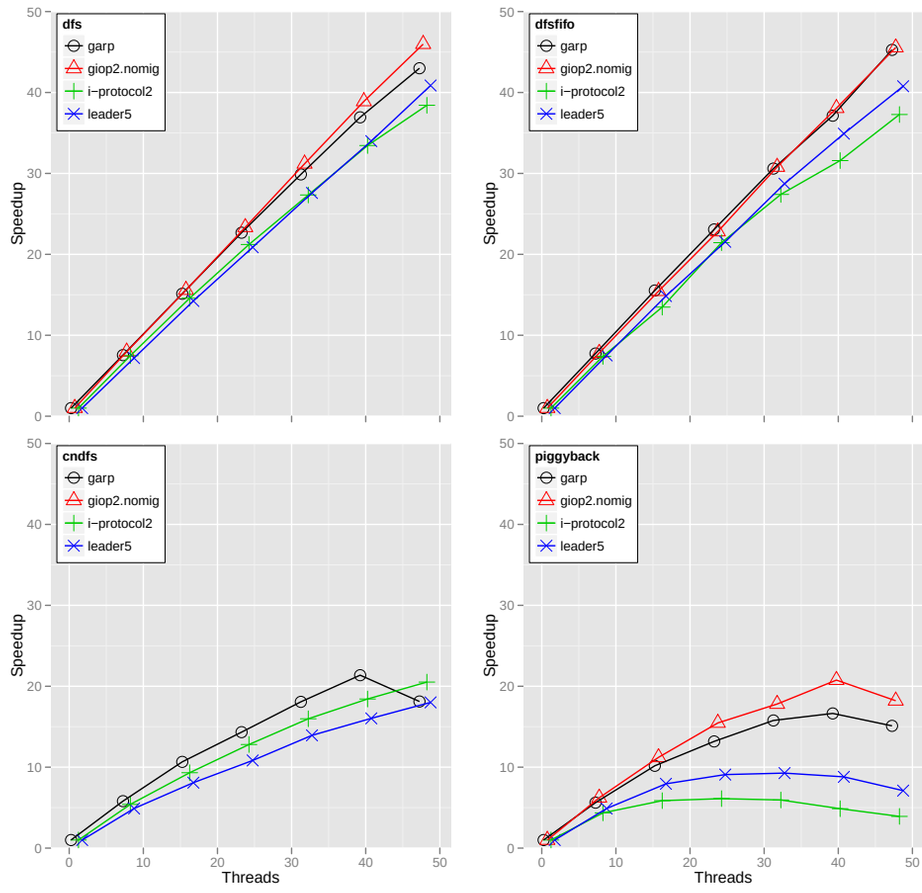


Figure 6.9.: Speedups of DFS, $\text{PDFS}_{\text{FIFO}}$ and CNDFS in LTSMIN(without giop due to oom), and piggyback in SPIN

occur at most 2.6% when using 48 cores. For CNDFS, strict and non-strict $\text{PDFS}_{\text{FIFO}}$, Table Table 6.7 shows the maximum number of states in all local queues and stacks: $Q_P := \sum_{w \in 1..P} (|\text{fifo}[w]| + |\text{stack}[w]|)$. Half the time, strict and non-strict $\text{PDFS}_{\text{FIFO}}$ have the same Q_i; in the other cases, non-strict is up to 1.2 times larger, except for garp with $P = 48$, where non-strict is almost 4 times smaller. Furthermore, $Q_{48} = 0.25 \cdot Q_1$ for non-strict $\text{PDFS}_{\text{FIFO}}$; this decrease is due to the randomness of the parallel runs and occurs in weaker form also for strict and non-strict giop. In all other cases, Q_{48} is at most 1.3 times Q_1. Q_1 of strict and non-strict $\text{PDFS}_{\text{FIFO}}$ is between 0.37 and 3.45 of CNDFS's Q_1. But CNDFS always has much larger Q_{48}: Due to the long paths from *init* to the current state for each worker thread in CNDFS, Q_{48} is at least 10 times as large as Q_1, resulting in Q_{48} of strict and non-strict $\text{PDFS}_{\text{FIFO}}$ being between 3 and 30 times smaller than Q_{48} of CNDFS where CNDFS does not run out of memory. For strict and non-strict $\text{PDFS}_{\text{FIFO}}$, $Q_i/S_{\to^*}$ is between 0.31 and 0.41 for giop, otherwise at most 0.17. This high factor does not cause memory problems since all fifo $[w]$ only store pointers to the real states in hash_table and revisits occur at most 2.6%. Accordingly, memory measurements showed that $\text{PDFS}_{\text{FIFO}}$'s total memory use with 48 cores was between 0.87 and 1.25 times the memory use of the sequential DFS. In the worst case, $\text{PDFS}_{\text{FIFO}}$ (with tree compression) required 0.52 times the memory use of PB (with collapse compression and lossy hashing), again giop excluded as its state counts differ.

Table 6.7.: Number of locally stored states for $\text{PDFS}_{\text{FIFO}}$ and CNDFS

	$\text{PDFS}_{\text{FIFO}}$ *strict*		$\text{PDFS}_{\text{FIFO}}$ *non-strict*		CNDFS	
	Q_1	Q_{48}	Q_1	Q_{48}	Q_1	Q_{48}
leader_t	1.0E6	1.2E6	1.2E6	1.4E6	2.7E6	3.6E7
garp	1.9E7	2.0E7	1.9E7	5.3E6	5.5E6	6.5E7
giop	1.1E9	8.4E8	1.1E9	8.4E8	oom	oom
i-prot	1.0E6	1.1E6	1.0E6	1.3E6	8.3E5	1.0E7

6.8.5. Scalability of Parallelism Combined with POR

We checked for NPCs on leader_{DKR} models, successively increasing the number of nodes N on the ring, i.e., the problem size. We compared $\text{PDFS}_{\text{FIFO}}$ with DiVinE's parallel LTL-POR algorithm, OWCTY, which we started with divine owcty [model] -wP -i30 -p. We also compared to CNDFS with parallelism and with POR, but not both (indicated by n/a), since CNDFS cannot handle the combination (cf. Subsec. 5.5.2). For $\text{PDFS}_{\text{FIFO}}$ and CNDFS, we turned on POR in LTSMIN with the options described above. We limited each run to half an hour (~~30~~' indicates a timeout). Table 6.8 shows that $\text{PDFS}_{\text{FIFO}}$ and POR complement each other rather well: Without POR (left half of the table), the almost linear speedup ($S_{48} = \frac{T_1}{T_{48}} = 40.8$) allows to explore one model more: $N \leq 10$ instead of only $N \leq 9$, just like CNDFS and OWCTY. But when POR is enabled (right half of the table), we see again multiple orders of magnitude reductions. Parallel scalability reduces to $S_{48} = 3.5$ for $N = 9$, though. This sub-linear scalability is caused by the small size of the reduced state space $|S^{\text{POR}}_{\to^*}|$, which is not sufficiently large to outweigh the parallel overhead (cf. Subsec. 3.5.2). When increasing N, the speedups grow again

to almost linear scalability: $S_{48} = 43$ for $N = 13$. With POR enabled, parallelism with $P = 48$ allows us to explore two more models compared to $P = 1$ within half an hour, up to $N = 15$. So $\text{PDFS}_{\text{FIFO}}$ handles 4 more orders of magnitude (cf. $|S^{\text{POR}}_{\rightarrow *}|$ and $|\mathfrak{T}^{\text{POR}}|$) compared to OWCTY ($N \leq 11$), 5 more orders of magnitude compared to CNDFS ($N \leq 10$).

Notes. As $\text{PDFS}_{\text{FIFO}}$ revisits states (cf. paragraph **Parallel Memory Use**), the randomly ordered next-state function could theoretically weaken POR (as for NDFS). But for all our experiments, this did not occur.

Since $\text{PDFS}_{\text{FIFO}}$ has low parallel memory use and never oom errors, it is not memory bound. Table 6.8 shows that eventually ~~30~~' occurs, so $\text{PDFS}_{\text{FIFO}}$ is CPU bound. Hence we focused on strong scalability in this section (cf. Subsec. 3.5.2), especially in Fig. 6.9. Table 6.8 does increase the problem size N, but not proportionally to p as for weak scaling, but always considers $p = 1$ and $p = 48$ to investigate the speedups and feasibility of $\text{PDFS}_{\text{FIFO}}$ (in comparison to OWCTY and CNDFS).

Piggyback reported contradictory memory use and far fewer states (e.g. <1%) compared to DFS with POR, although it must meet more provisos. Thus we did not compare against piggyback (and suspect a bug).

Table 6.8.: POR and speedups for leader_{DKR} using $\text{PDFS}_{\text{FIFO}}$, OWCTY and CNDFS

N	Alg.	$\lvert S_{\rightarrow *}\rvert$	$\lvert\mathfrak{T}\rvert$	T_1	T_{48}	S_{48}	$\lvert S^{\text{POR}}_{\rightarrow *}\rvert$	$\lvert\mathfrak{T}^{\text{POR}}\rvert$	T^{POR}_1	T^{POR}_{48}	S^{POR}_{48}
9	CNDFS	3.6E7	2.3E8	502.6	12.0	41.8	27.9%	0.1%	211.8	n/a	n/a
9	$\text{PDFS}_{\text{FIFO}}$	3.6E7	2.3E8	583.6	14.3	40.8	1.5%	0.0%	12.9	3.6	3.5
9	OWCTY	3.6E7	2.3E8	498.7	51.9	9.6	12.6%	0.0%	578.4	35.7	16.2
10	CNDFS	2.4E8	1.7E9	~~30~~'	90.7	~~30~~'	19.3%	5.4%	1102.7	n/a	n/a
10	$\text{PDFS}_{\text{FIFO}}$	2.4E8	1.7E9	~~30~~'	109.3	~~30~~'	0.7%	0.1%	35.0	2.5	14.0
10	OWCTY	2.4E8	1.7E9	~~30~~'	663.1	~~30~~'	8.7%	2.2%	~~30~~'	156.3	~~30~~'
11	$\text{PDFS}_{\text{FIFO}}$	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	5.1E6	7.1E6	109.8	5.3	20.7
11	OWCTY	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	9.3E7	1.7E8	~~30~~'	1036.5	~~30~~'
12	$\text{PDFS}_{\text{FIFO}}$	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	1.6E7	2.2E7	369.1	11.2	33.0
13	$\text{PDFS}_{\text{FIFO}}$	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	6.6E7	9.2E7	1640.5	38.1	43.0
14	$\text{PDFS}_{\text{FIFO}}$	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	2.0E8	2.9E8	~~30~~'	120.3	~~30~~'
15	$\text{PDFS}_{\text{FIFO}}$	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	8.4E8	1.2E9	~~30~~'	527.5	~~30~~'

6.8.6. On-the-flyness

We also investigate on-the-flyness, i.e., on-the-fly performance, which is quite relevant in practice (cf. Subsec. 3.6.2). Since $\text{leader}_{\text{Itai}}$ counts rounds (cf. Subsec. 6.1), we were able to modify the model to analyze on-the-fly performance: we injected early NPCs (model *shallow*), i.e., close to *init*, and late NPCs (model *deep*), i.e., far away from *init*. Both models are available at [URL:leader4DFSFIFOHP]. Table 6.9 shows the average runtime in seconds (T) and counterexample length (number of states C) over five runs; for CNDFS's T_1 we took the average over 4 runs since one run had a timeout (including the timeout results in an average of $T_1 \geq 373$ seconds). Since $\text{PDFS}_{\text{FIFO}}$ finds shortest counterexamples, it outperforms CNDFS for the *shallow* model (more relevant model in practice because of the small scope hypothesis, cf. Subsec. 5.2.3) by a factor larger 150 for $P = 1$ and 1.75 for $P = 48$ and pays a penalty for the *deep* model by a factor larger 4.8 for $P = 1$ (the timeout for CNDFS included) and 225.5 for $P = 48$. Both algorithms benefit greatly from massive parallelism (cf. [Laarman and van de Pol, 2011]). For the

model *deep*, CNDFS's counterexamples are about 10 times as long as for $\text{PDFS}_{\text{FIFO}}$; the model *shallow* only contains shortest counterexamples.

Table 6.9.: On-the-fly runtimes (sec) and counterexample lengths (states) for CNDFS and $\text{PDFS}_{\text{FIFO}}$

	CNDFS		$\text{PDFS}_{\text{FIFO}}^{\mathfrak{T}}$		CNDFS		$\text{PDFS}_{\text{FIFO}}^{\mathfrak{T}}$	
	T_1	T_{48}	T_1	T_{48}	C_1	C_{48}	C_1	C_{48}
shallow	~~30'~~	7	12	4	~~30'~~	16	16	16
deep	16($^{\text{once}}_{\sim 30'}$)	2	~~30'~~	451	577	499	~~30'~~	51

Notes 6.34. Table 6.9 focuses on level 2 OTF, which is why we do not consider DiVinE, which only has level 1 OTF.

CNDFS exhibits super-linear speedup of $S_{48} > 257 > 48$ on average for *shallow* (and $S_{48} \geq 187$ for *deep* if the timeout is included). If the NPCs were evenly distributed over the state space, "only" linear speedup would be possible on average. But since the NPCs are all in a relatively small region of the state space, even super-linear speedup on average is possible [Rao and Kumar, 1988].

Often, it is stated that super-linear speedup is impossible because the parallel algorithm can be simulated by a sequential algorithm with the same amount of overall work [Amdahl, 1967; Faber et al., 1986]. This, however, is only the case if the sequential and parallel algorithms offer the same operations, with the same costs. But it need not be the case [Janßen, 1987; Bader et al., 2000; Sutter, 2008], e.g., due to simulation costs, communication costs, or different hardware costs like caching.

Our experiments use standard parameters and measures, and established protocols from benchmarks, which lead to low threats to internal, construct, and external validity. Threats to statistical conclusion validity are also low since we usually have little variance in spite of $\text{PDFS}_{\text{FIFO}}$ being a randomized algorithm. Furthermore, the differences for our comparisons are usually orders of magnitude. Hence we took the average only over 5 runs for each relevant experiment. For *deep* with the sequential CNDFS, however, the variance was large, so the measurements are not statistically accurate. Since CNDFS is not our focus, these rough measurements are only a small threat to statistical conclusion validity for our experiments of $\text{PDFS}_{\text{FIFO}}$.

6.9. Conclusion

6.9.1. Summary

This chapter showed theoretically and by experiments that the important property of livelocks can be performed more efficiently than via LTL MC: By specializing on LTL's real subset of NPC checks, DFS_{FIFO} can simultaneously explore the state space and search for NPCs in one pass, instead of separating these steps, as NDFS for LTL MC does.

DFS_{FIFO} achieves this by lazy progress traversal: DFS_{FIFO} postpones traversing progress until all states that are reachable from the current state without progress have

been traversed. Then DFS_{FIFO} retrieves a postponed state and continues with this approach.

To express this algorithm and its laziness as a scheduling of tasks (cf. Def. 3.33), we partition DFS_{FIFO}'s transition tasks into tasks that traverse a progress transition and tasks that traverse a non-progress transition. Check tasks perform an NPC check, which only needs to check whether the current transition performs progress and whether its destination state is on the stack. These simple checks are sufficient if non-progress transitions are scheduled in DFS order. DFS_{FIFO} performs livelock detection by lazily scheduling progress transition tasks only after non-progress transition tasks only after NPC check tasks. It is a level 2 OTF algorithm, and truly OTF since no Büchi automata is used.

DFS_{FIFO}'s advantages over NDFS are:

- more efficient since it requires only one pass over $\mathcal{A}_{\mathcal{S}}$, not two passes over $\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{\Diamond\Box\text{np\_}}$ like the NDFS;
- progress can also be modeled by progress transitions instead of progress states;
- lazy progress traversal makes the POR proviso C3 and C3' obsolete, resulting in stronger POR;
- DFS_{FIFO}'s BFS behavior on exploring after-progress states results in better parallel scalability;
- POR and parallelism are efficiently combinable;
- stronger on-the-flyness for the livelocks that occur in practice;
- it finds shortest counterexamples w.r.t. progress.

As only trade-off, DFS_{FIFO}'s counterexample solely contains the NPC. To construct a full counterexample, DFS_{FIFO} requires negligible additional time or space. DFS_{FIFO}'s disadvantage over NDFS is its restriction to livelock detection (more precisely: persistence properties). Unfortunately, DFS_{FIFO} cannot be generalized to full LTL MC.

Our experiments on four established protocols prove these advantages for our $\text{PDFS}_{\text{FIFO}}$ implementation with progress transitions:

- DFS_{FIFO}'s runtime is 1.6 to 3.2 times smaller than NDFS's for our experiments, and DFS_{FIFO}'s memory use is 1.5 to 2 times smaller (cf. Table 6.4);
- DFS_{FIFO}'s POR is over 3 to over 200 times stronger than NDFS's, resulting in reduced state spaces between 0.49% and 31.8% of the original size, which is only 1.0 to 2.0 times larger than that of a basic DFS with POR (cf. Table 6.5);
- $\text{PDFS}_{\text{FIFO}}$ is the fastest algorithm for NPC checks for the sequential case (about twice as fast, cf. Table 6.6) and has almost linear speedup (cf. Fig. 6.9), resulting in 3.4 to 16 times faster parallel NPC checks for our experiments (cf. Table 6.6);
- $\text{PDFS}_{\text{FIFO}}$ has 0.87 to 1.25 times the memory use of the sequential DFS, and requires 3 to 30 times less local memory than NDFS (cf. Table 6.7);
- $\text{PDFS}_{\text{FIFO}}$ in combination with POR handles 5 more orders of magnitude compared to NDFS, 4 more orders compared to DiVinE's OWCTY (cf. Table 6.8);
- for relevant livelocks of our experiments, $\text{PDFS}_{\text{FIFO}}$ has over 150 times stronger on-the-flyness for $P = 1$, 1.75 times for $P = 48$ (cf. Table 6.9);
- for our experiments, $\text{PDFS}_{\text{FIFO}}$'s counterexamples are up to 10 times shorter than those of NDFS.

6.9.2. Contributions

The main contribution in this chapter is the design, implementation and analysis of the algorithm DFS_{FIFO} for efficient livelock detection, its parallelization $\text{PDFS}_{\text{FIFO}}$, the adaption and integration of POR without the need of a cycle proviso, correctness proofs thereof, and experiments. The core ideas of DFS_{FIFO} have already been published in [Faragó, 2007], its correctness and integration with POR in [Faragó and Schmitt, 2009], its parallelization, implementation and measurements in [Laarman and Faragó, 2013].

More abstract contributions are:

- the introduction of a semantically more accurate way to model progress by using progress transitions;
- a lazy technique with advantages beyond those listed in Subsec. 3.6.3: It renders the one pass DFS NPC check complete: using lazy progress traversal, DFS_{FIFO} becomes a sound and complete livelock (more generally: persistence property) detection. Furthermore, lazy progress traversal improves optimizations (POR and parallelization), is an efficient search heuristic for livelocks, and yields shortest counterexamples;
- showing that there exist relevant subclasses of liveness properties that are worth being specialized on since this specialization enables many improvements compared to general LTL MC. Compared to DiVinE, which improves LTL MC for the larger subclass of weak LTL, $\text{PDFS}_{\text{FIFO}}$ focuses on a subclass of weak LTL and achieves strong level 2 on-the-flyness and almost linear speedup, whereas DiVinE has level 1 OTF and logarithmic speedup.

6.9.3. Future

Possible future work includes:

- integrating DFS_{FIFO} into SPIN, which Gerard Holzmann, the author of SPIN, is looking into;
- POR might be further improved for DFS_{FIFO} by weakening the visibility proviso: To find NPCs, we only need to distinguish π with infinite progress from π with finite progress. Thus POR needs not guarantee stutter equivalence, but only that NPC existence is preserved, i.e., that at least one progress is preserved per progress cycle and one NPC is preserved. One step in this direction is reducing the number of progresses on a cycle that are visible to POR using lazy full expansion due to progress (cf. Subsec. 6.6.3): This is formalized by $\text{C2}^{\mathfrak{T}}_{\text{lazy}}$ and $\text{C2}^{S}_{\text{lazy}}$; implementing and optimizing them by coupling the provisos to the exploration algorithm [Evangelista and Pajault, 2010] is interesting future work (cf. Note 6.22);
- investigate on-the-flyness of $\text{PDFS}_{\text{FIFO}}$ more thoroughly (cf. Note 6.34);
- this chapter has shown that focusing on special sub-classes of liveness properties, such as livelocks (more generally: persistence properties), can improve the efficiency and optimization of MC. Similar conclusions have been made for three other sub-classes: a fragment of CTL that restricts nesting [Saad et al., '12], weak LTL properties (cf. Subsec. 6.7.4) and response properties (cf. Note 6.17). Thus we should look for further sub-classes, how they are related, and how verification can be specialized. Exemplary future work is adapting DFS_{FIFO} to response properties (cf. Note 6.17);

- implement finite ∪ infinite trace semantics for DFS_{FIFO} and its POR. Similarly to on-the-fly LTL MC (cf. Subsec. 5.6.3), this would be simple: Solely a check for non-progress at end states is necessary. Allowing also finite traces, no work-arounds are necessary, whereas on-the-fly LTL MC with only infinite trace semantics must add self-loops that also make progress – or not add self-loops to end states at all for this special case (cf. 6.3.1);
- transform DFS_{FIFO} and strict $\text{PDFS}_{\text{FIFO}}$ into a conditional model checking tool by simply adding the contents of fifo as input C_{input} and as output C_{output} to DFS_{FIFO} (cf. Subsec. 6.4.3).

7. Testing with Software Bounded MC

To improve the feasibility of model checking, Sec. 5.4 introduced reduction techniques, with a focus on partial order reduction and symbolic model checking via bounded MC. Chapter 6 focused on liveness properties and achieved improvements via specialization on the livelock property and via partial order reduction. This chapter investigates the feasibility of software bounded MC (cf. Subsec. 5.2.3) for C and C++ source code, mainly using the tool CBMC and safety properties, which is the more common application [Yi et al., 2004; Prasad et al., 2005]. Experiments on a case study show that state space explosion occurs when nondeterminism is used too heavily in CBMC. So even SBMC, which is meant to be a more lightweight approach than exhaustive MC, can quickly become infeasible. We introduce an even more lightweight approach via testing, baptized **testing with SBMC**: the test engineer manually selects underspecified scenarios, which are checked via SBMC. This approach leverages SBMC and improves scalability.

This chapter is the first example in this thesis about combining a formal method with the testing approach. Therefore, it motivates Part III, which focuses on this combination. In this chapter, however, static testing and hence white-box testing is conducted. Nonetheless, several approaches introduced here will be applied in Part III, too: the integration of formal methods and testing is the main concept for Part III, bounded exploration one of the key design features, and one implementation will use similar techniques (SMT and compiler optimizations).

We describe a case study from the domain of wireless sensor networks (**WSNs**), using a large program of practical size ($\approx$ 21 000 LoC) and complexity. A WSN is a distributed embedded system consisting of autonomous sensors that cooperatively pass their measurements through the network to a main location. Trying to check the large program with CBMC causes state space explosion, so we used abstractions: lossy abstractions by removing technical details and by introducing nondeterminism (cf. Subsec. 3.7). But too much nondeterminism causes too high a combinatorial explosion, so we had to restrict the state space again. For this, we applied the **testing approach**: We manually selected scenarios that seemed most likely to reveal relevant bugs or raise our confidence in the correctness of the program. So we conducted experiments by choosing scenarios, similar to testing. But we did not perform dynamic testing, i.e., we did not execute the program; instead the properties were checked on the chosen scenario using CBMC, which covered all possible situations (paths and inputs) within the scenario due to underspecification, i.e., all remaining nondeterministic values. Therefore, we call this approach testing with SBMC. The only manual tasks for testing with SBMC are selecting scenarios and user-supplied properties to be checked; all other instrumentations, general abstractions and verifications can be automated.

Most of this work was published in [Werner and Faragó, 2010]; this chapter introduces SBMC, CBMC and the optimization heuristics more thoroughly, gives more related work and explains our lightweight approach in greater detail. For the case study, Frank

Werner [Werner, 2009] implemented all abstractions, scripts and code instrumentations, and conducted the first verification round of the case study. Hendrik Post implemented all optimization heuristics. Critical scenarios of the first round were inspected again by Frank Werner and David Faragó in a second round using specific scenarios.

In summary, verification of our large and complex system was in general possible with SBMC, but in a lightweight manner: we used rigorous static analysis via CBMC, but with a manual restriction to certain scenarios, just like testing does for execution.

Roadmap. In Sec. 7.1, we introduce SBMC: firstly in general, then the SBMC tool CBMC, its capability to use nondeterminism, and the complexity of SBMC. Then we describe the heuristic improvements we contributed to make CBMC cope with our protocol. Sec. 7.2 introduces wireless sensor networks, the ESAWN protocol [Calvert et al., 1999], the TinyOS platform and our abstraction from it. Then the desired properties of ESAWN are described and verified. Sec. 7.3 concludes this chapter.

7.1. Introduction to Techniques for SBMC

The overall approach of SBMC was introduced in Subsec. 5.2.3. But for SBMC, the user-supplied bound b usually does not directly describe the number of allowed steps on each path, but rather denotes the maximum number of allowed loop body executions on a path and the maximum recursive depth. The **recursive depth** of a path is the number of stack frames it contains (more precisely: the number of equal return addresses on the stack). For a given program, b still limits the number of statements on any path. As described in Subsec. 5.2.3, the bound b for BMC with bound check can iteratively be set large enough so that the SBMC tool becomes sound and complete for all properties the tool can check. To apply SBMC for static code analysis (usually in the language C or C++), the source code is transformed into a Kripke structure $\mathcal{S}$ (similarly to Subsec. 3.4.3). For this, the code is **unwound**: loops are **unrolled** up to the upper bound, as depicted in Listing 7.1; similarly, function calls are **inlined** up to the upper bound. The resulting Kripke structure is in $\mathbb{S}_{Kripke,(labeled),finite}$.

```
int x = 0;                          int x = 0;                          1
int y = 0;                          int y = 0;                          2
while (x < 2){                      if (x < 2){                         3
   y = y + x;                          y = y + x;                       4
   x++;                       ↦        x++;                             5
}                                   }                                   6
                                    if (x < 2){                         7
                                       y = y + x;                       8
                                       x++;                             9
                                    }                                   10
                                    assert (!(x<2));                    11
```

Listing 7.1: Exemplary loop unwinding for a bound of 2

SBMC for static code analysis is important since "static analysis has become one of the leading methods for software defect removal from source code" [Jones and Bonsignour,

2012].

To simplify static analysis, logical encoding and optimizations of the source code, many SBMC and other tools translate it into an **intermediate representation** (**IR**), which is an abstract assembler language with simpler syntax and semantics. Often the **static single assignment form** (**SSA**) [Rosen et al., 1988] is used (which is a special case of continuation-passing style [Kelsey, 1995]): Every assignment is replaced by a **versioned assignment** such that each identifier is assigned at most once. In order to transform a sequence of n assignments to an identifier, n new identifiers are introduced by appending a **version number** to the original identifier. Read accesses are replaced by read accesses to the currently active version. At program points where two control flows join, e.g., at the end of an **if** block, a new Φ **function** is introduced. A Φ function determines which version should be used for the consecutive read accesses, cf. Listing 7.2. Using control flow analysis, so-called dominance algorithms [Braun et al., 2013] determine where to insert Φ functions in the intermediate representation. Subsec. 7.1.3 shows an example for optimizations becoming much easier on SSA compared to C code: constant propagation, which employs use-definition chains. These are easily computed for SSA.

```
int x = 0;                          x0 = 0;                          1
if (x==0){                          if (x0==0){                      2
   x = x + 2;          ↦               x1 = x0 + 2;                  3
}                                   }                                4
assert(x!=0);                       // phi:                          5
                                    x2 = (x0==0) ? x1 : x0;          6
                                    assert(x2!=0);                   7
```

Listing 7.2: Exemplary translation from C code to SSA

With this, usually only Kripke structures in $\mathbb{S}_{Kripke,(labeled),finite,<\omega}$ are considered. An embedded system (e.g., one following the MISRA C standard [Association, 2004]) commonly is a **reactive system**, i.e., an event-driven system that continuously reacts to the stimuli of its environment [Manna and Pnueli, 1995; Clarke et al., 1999b]. Therefore, it indefinitely offers finite functions, i.e., it consists of an endless loop on the outside, but within that loop, all possible paths are bounded. Fortunately, SBMC can often show ultimate correctness for typical properties of such a system even when only considering one iteration through the loop (i.e., one loop unwinding, cf. Listing 7.1 below). So the inner, finite functions without the infinite loop around them are investigated (usually with generalized parameters), such that we have the situation above, with only finite paths and a finite number of them. For this **finitization**, we might require underspecified variable values via nondeterminism (cf. Subsec. 7.2.3), which many SBMC tools offer by:

- setting variable values nondeterministically with an **explicit** command, e.g., **havoc** or **nondet**;
- or by **implicitly** setting uninitialized input variables to nondeterministic values.

Nondeterminism can also be used to underspecify functions: Underspecified input parameters can cover all possible values. Alternatively, the whole function can be abstracted, for which its side effects are captured by nondeterministically setting the return value and variable values that the functions may mutate. Therefore, source code that interacts with the environment can be transformed to a closed system using nondeter-

minism (cf. Sec. 2.3 and Sec. 3.7), for instance nondeterministically setting the sensor values that the wireless sensors measure and communicate. So via nondeterminism, the system specification description can subsume all possible behaviors of the SUV. The description often contains even more behaviors than the SUV, i.e., it is truly underspecified: The resulting (labeled) Kripke structure exhibits more behavior than the SUV, possibly producing false positives, i.e., SBMC can become unsound. To avoid these false positives, abstraction must be reduced by allowing only nondeterministic values that the SUV exhibits too, e.g., by using assume(·) statements, (see next subsection) or by choosing specific scenarios manually. Unfortunately, nondeterminism causes the SBMC tool to consider even more combinations (before they are pruned by assume(·)), i.e., state space explosion is even more severe. For our large case study, the complexity became often too high to prove ultimate correctness. Instead, we had to restrict the cases by choosing specific scenarios manually: this testing with SBMC is described in Subsec. 7.2.3.

7.1.1. CBMC

The **C Bounded Model Checker** (**CBMC**) [URL:CBMC] is one of the most popular SBMC tools and implements SBMC for C programs. Properties have to be specified by **assert (·) statements**, which are not only used for user-supplied C assertions, but also to encode **built-in standard runtime error checks**: arithmetic errors like integer overflow and underflow and memory errors like array index out of bound and illegal pointer access [Clarke et al., 2004a].CBMC also offers **assume(f) statements**, which do not demand f to be **true**, but simply cause all outgoing transitions of the current state to be pruned (i.e., ignored) iff f evaluates to **false** (but we do not need assume(f) statements in Sec. 7.2). Nondeterministic values must be set explicitly, e.g., by `int x = nondet_int();`. In CBMC, the bound b can be set individually for each loop occurring in the program. The SSA statements have **guards** in CBMC, i.e., necessary and sufficient conditions for the statements to be executed.

Similarly to Subsec. 5.4.2, CBMC encodes the SBMC problem into a SAT instance that is checked using the SAT solver Minisat2 [Eén and Sörensson, 2003]. The SAT instance can also be exported in the DIMACS format. Alternatively, the SBMC problem is encoded into SMT-LIB using the logic QF_AUFBV and some support for lists, sets and maps. The solvers Z3, Yices or Boolector can be used.

If the SAT or SMT problem is satisfiable, CBMC generates a concrete counterexample from the satisfying assignment produced by the SAT or SMT solver. If the SAT or SMT problem is not satisfiable, the property holds for all program executions (since the bound check did not produce a counterexample, cf. Subsec. 5.2.3) and the program always terminates.

7.1.2. Complexities

The worst case time complexity of SAT is exponential in the number of variables of the SAT encoding, which is in $O(b \cdot (|\Sigma_{sv}(\mathcal{S})| + |\Sigma_{sv}(L)|))$ for the Kripke structure $\mathcal{S}$ and a liveness property description L (cf. Subsec. 5.4.2), and in $O(b \cdot \Sigma_{sv}(\mathcal{S}))$ for a reachability property. Runtime did not cause problems in our case study since real world instances of SAT problems can often be solved surprisingly fast in practice (cf. Subsec. 3.2.3).

The size of the encoding, however, did cause problems even for small bounds: The worst case size complexity of the encoding is in $O(b{\cdot}(C(\mathcal{A}_{\mathcal{S}}{\cap}\mathcal{A}_{never}){+}|\Sigma_{sv}(\mathcal{S})|{+}|\Sigma_{sv}(L)|{+}C(F)))$ for Büchi automata $\mathcal{A}_{never}$ for L with F acceptance states (cf. Subsec. 5.4.2), and in $O(b \cdot (C(\mathcal{S}) + \Sigma_{sv}(\mathcal{S})))$ for a reachability property. $C(\mathcal{S})$ already has at least as many variables as the number of bits potentially addressed in the C program. So for SBMC on the source code level, the state space and $C(\mathcal{S})$ become huge since many values of the heap, stack, registers and program counter need to be considered, especially when nondeterminism is used. Consequently, the generated encodings had many GiB of file size and caused failures in Minisat2 simply because of their size; the SAT instances that caused no such failure could be solved rather quickly by Minisat2. Alternatively, we exported the SAT instances as DIMACS files. But even for reachability properties, the size of the propositional formula that CBMC generated surpassed 4 GiB. Consequently, alternative SAT solvers and reductions and simplifications on these DIMACS files were also not able to handle their size. We solved this problem by integrating preprocessing in CBMC before calling Minisat2, as described in the next subsection. Even though the worst case size complexity of the encodings did not change, the encodings no longer became so big and could often be handled by Minisat2.

7.1.3. CBMC Optimization Heuristics

To decrease the size of the SAT instances, we engineered some preprocessing into CBMC that use heuristics from the field of compiler optimization on the most suitable level: the intermediate representation in SSA form (cf. Sec. 7.1).

CBMC already had some optimization heuristics, but only on the SAT level and only on primitive data types. Our heuristics also respect non-simple types like arrays, pointers and structures, and also use slicing rules (enabled by the option −slice) and **simplifications** (enabled by the option −use−sd). The simplifications use the following steps:

- constant propagation for arrays, pointers and structures, which can be computed efficiently in an unwound program (cf. Subsec. 7.1.3);
- expression simplification that uses the additional information generated by the constant propagation (cf. Subsec. 7.1.3);
- simplifying guards for statements by early satisfiability detection, using the above expression simplification (cf. Subsec. 7.1.3).

Hendrik Post implemented these heuristics within CBMC version 2.9, which are available on a side branch of CBMC's repository thanks to Daniel Kröning. Our preprocessing strongly reduces the problem size and complexity, enabling the verification of the ESAWN implementation in Sec. 7.2.

Field- and Array-Sensitive Constant Propagation

Many implementations (like the ESAWN protocol, cf. Subsec. 7.2.2) heavily use arrays, pointers and structures, which are not covered by CBMC's optimizations. Hence sequences as $\mathrm{a}[0] = 0$; **if** $(\mathrm{a}[0] == 0)$ ... are not simplified by CBMC's constant propagation. In contrast, we have implemented the propagation on the level of the SSA representation of the program by flattening these complex data types.

Expression Simplifier

Using the additional information generated by the constant propagation, we have added an expression simplifier. Any expression that can be simplified by one of the three following rules is replaced by its simplified expression. Note that all expressions are free of side effect at this level of encoding:

- Boolean expressions with Boolean operands: If an expression has Boolean type and any Boolean operand must evaluate to a constant `true` or `false`, the expression is simplified, e.g.,
  - `!true` becomes `false`,
  - `expr && false` becomes `false`,
  - `true => expr` becomes `expr`,
  - `false? expr1:expr2` becomes `expr2`.

  Additional cases where more than one operand evaluates to a constant are also simplified.
- Boolean comparisons: Operators like `<`, `>=`, `==` and `!=` are also simplified; e.g., `c <= c` becomes `true`, with `c` being a constant or versioned identifier.
- Integer, float and double expressions with constant integer or Boolean operands: Arithmetic expressions like `+,-,*,/,<<,>>` with all operands being constants are simplified according to their C semantics.

The last rule could easily be extended to `float` and `double` type variables. As the ESAWN implementation does not use such types, they are not yet implemented.

Early Satisfiability Detection

The simplifications from the previous two subsections can be effectively used to simplify the guards of the SSA statements: If the guard of a statement always evaluates to **false**, that statement can be removed from the encoding as it cannot be executed. If a guard evaluates to **true** and the statement is $\text{assert}(f)$ with f always evaluating to **true**, it can be removed. If a reachable guard evaluates to **true** and the statement is $\text{assert}(f)$ with f always evaluating to **false**, there is a counterexample for the corresponding property. The heuristics can often detect the reachability and stop further encoding with an appropriate message.

These heuristics detect that loop bounds are chosen too small if the loops are executed a fixed number of times (most of ESAWN's loops are). Since the unwinding bounds for loops are unknown a priori, many CBMC iterations are necessary. Thus the overall process of finding the correct loop bounds is greatly improved by early satisfiability detection.

7.2. Case Study

This section introduces a case study with a complex and large code basis of 21 000 LoC. It was published in [Werner and Faragó, 2010] and is from the field of wireless sensor network protocols. For verification, CBMC version 2.9 was used, the most recent version when our work started, together with our optimization heuristics. The major parts of the implementation and experiments were conducted by Frank Werner [Werner, 2009].

7.2.1. Introduction

A **wireless sensor network** is a network of autonomous sensors, called **nodes**. These are embedded devices for that monitor **environmental data** and communicate wirelessly. We strongly rely on monitoring applications and WSNs, which are used in highly distributed as well as safety-critical systems, such as structural health monitoring of bridges [Xu et al., 2004], intrusion detection [Ioannis et al., 2007], and in many industrial use cases, e.g., using SureCross from Banner Engineering Corp. [URL:SureCross] or Smart Wireless from Emerson Electric Co [URL:EmersonSmartWireless]. Hence WSNs must become more safe and dependable. The application of formal methods to embedded systems could be the key to solve this. Although WSN nodes carry only some hundred kilobytes of memory, their verification is neither simple nor easily automated (cf. related work below) because they run complex algorithms for the underlying protocols, distributed data management, and wireless communication. In this domain, powerful and extensible development and deployment frameworks are used, e.g., TinyOS (cf. Subsec. 7.2.3). By integrating formal methods seamlessly, i.e., fully automated, into such a framework, their usage by developers is most likely. Thus we decided for automatic generation of the model (cf. Subsec. 7.2.3) for our verification process, which also solves further problems:

- the development of manual artifacts is error-prone and costly since the model is only required for verification and cannot easily be used for further development;
- since the verification model and the implementation must stay in conformance, the model must rapidly change, especially during the design phase. Hence additional work is required;
- there is a high danger to abstract from fault-prone details, e.g., due to missing constructs in the modeling language, making it hard to model those details.

Roadmap. Subsec. 7.2.2 investigates a concast protocol implementation called ESAWN. It is from the domain of WSNs and uses the development and deployment platform TinyOS, introduced in Subsec. 7.2.3. Subsec. 7.2.3 describes the automatic generation of our abstract software behavior model from the given ESAWN implementation. Subsec. 7.2.4 describes how to specify desired properties and then enumerates properties for $STATUS$ packets and ESAWN packets. Subsec. 7.2.5 describes which properties could be proved and why the others could not.

7.2.2. The ESAWN Protocol

The system under verification is the **Extended Secure Aggregation for Wireless sensor Networks** (**ESAWN**) [Blaß et al., 2008]: It offers means to handle **concasts**, i.e., the transportation and aggregation of messages in the sensor network from many senders to one receiver. The path a message takes (ignoring witnesses, see below) is called **aggregation path**, the set of all aggregation paths is called **aggregation tree**. The aggregation tree is the topology of the WSN and set up in the initialization phase. So the protocol runs in two phases: First an **initialization** is necessary, before the actual **probabilistic concast** can be performed in the second phase. By using an end-to-end authenticity, the transport of sensible data is possible even in the presence of multiple

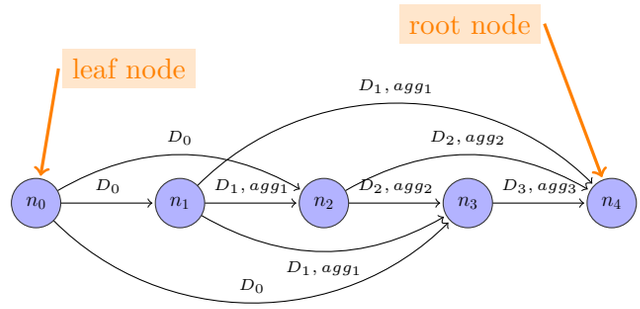


Figure 7.1.: ESAWN scenario of an aggregation tree with $w = 2$ witnesses.

malicious nodes under the control of an adverse acting entity.

Probabilistic Concast

We first consider the second phase, in which the actual sensor data is passed around and aggregated by the probabilistic concast, all along the way to the **sink**, i.e., the root node of the network, where the data is collected. We call the type of packets used in this phase **ESAWN packets**. Since the packets are relayed down the aggregation tree via intermediate nodes, their entries are encrypted such that only the destination can decode its contents (see Subsec. 7.2.4).

To enable nodes to check for authenticity of the aggregates they receive, each node sends its information to a fixed number w of additional child nodes, called **witnesses**. For the concast with probability, each node checks the authenticity of each received aggregate only with a given probability p, otherwise it just assumes that the aggregate is authentic. Since authentication is costly, this is a trade-off between low energy consumption (low p) and high probability of authenticity (high p). The employed concast saves additional energy by buffering packets and sending them all together later on, using an **aggregation function f_{agg}**.

Example. An exemplary setup is given in Fig. 7.1, where 5 nodes are used: The **leaf node** n_0, i.e., the node without predecessors, triggers the probabilistic concast by sending a packet (with its sensor data value D_0) to its successor on the aggregation path, n_1. Since this node could be cheating, additional packets are sent to node n_2 and n_3, which act as witnesses to assure the proper behavior of node n_1. The nodes n_i, with $i \in [1, \ldots, 3]$, are collecting all incoming packets, then check authenticity with probability p and finally, if all incoming packets were authentic, send out packet $agg_i = f_{agg}(agg_{i-1}, D_i)$ (with D_i being the new data value from n_i and $agg_0 := D_0$). The root node n_4 finally collects all data. It is located at the base station and accessible by the user.

Protocol Initialization

In the initialization phase, the parameter settings and the aggregation tree are made known to all nodes in the network. For this task, the ESAWN protocol uses *STATUS* **packets**, which are also encrypted (see Subsec. 7.2.4). The number of nodes, *num_nodes*,

probability p and the number of witnesses w are sent around in the network using *SET* **packets**. In addition, the aggregation tree is spread using *SETAGG* **packets**, which contain the *parent_ids* for each node. Finally, a *GO* **packet** triggers the second phase of the protocol. The *GO* packet contains a value specifying the **frequency** at which nodes send their data (0 means only one concast). Further packet types exist, which we do not consider since they play only a minor role for the verification of relevant global properties.

7.2.3. TinyOS Platform and Model Abstraction

The TinyOS Platform

TinyOS [URL:TinyOS] is an open source operating software for embedded devices. Its component based architecture and event driven execution model make it very suitable for resource constrained hardware systems with respect to memory, computation power and energy shortness. TinyOS is both an operating system and a software development platform that offers instruments to deploy the implementation on various hardware platforms through a modular design. So once a protocol like ESAWN is implemented, it can be deployed automatically to the desired sensor type. With many possible combinations of interacting components, automatic verification within TinyOS to check that the resulting composition behaves as expected is a solution for this **feature interaction problem**, i.e., for the combinatorial explosion of interactions amongst the offered features. In more detail, software in TinyOS is initially written in **nesC**, a C dialect having special constructs for embedded devices. Before the software can be deployed on sensor nodes, it is firstly translated from the modular description in nesC into an intermediate ANSI C representation, which includes specific constructs for interaction with the hardware. This C code could theoretically be used as model for the verification process already. But an abstraction is required when considering the size and complexity of the C code: 21 000 LoC that also include hardware parts with register assignments and interrupt handling inhibit verification because of state space explosion. Thus, the following subsection takes the approach of abstracting from the hardware part by generating an abstract behavior model.

A Behavior Model Abstraction

The NULL Platform. The **NULL platform** is a hardware model included in the TinyOS environment, which can be used to generate a hardware independent software behavior model: The model is a skeletal structure containing only the functionality of the protocol plus some overhead in form of the **scheduling** functions for jobs and the **job queue**. But all hardware specific functions, e.g., for the UART and LEDs, are simple functions that always terminate, and on a lower layer than the protocol. Thus we were able to remove them, i.e., generate mostly empty function bodies. This abstraction is lossy, but only irrelevant information is removed; the few relevant parts of the hardware specific functions that had been removed by using empty function bodies were reintroduced (see next subsection). Therefore, exactly the behavior relevant for our protocol verification is preserved, so that it is **sound** and **complete**, i.e., verifying the abstraction does not induce false positives or false negatives. Besides strongly reduc-

ing complexity, this abstraction has the major advantage that we do not have to take hardware platforms into account when specifying properties.

Abstract Behavior Model.

To enable verification with CBMC, five modifications to the NULL platform were made:

1. reintroduce relevant details the NULL platform abstracted from;
2. workarounds to avoid deficiencies of CBMC;
3. code instrumentation for the properties described in Subsec. 7.2.4;
4. enable a full but flexible initialization, such that we have a closed system (cf. Sec. 7.1) and are able to cover all relevant scenarios;
5. finitization to make CBMC sound and complete.

We call the result **abstract behavior model**, which is C code annotated with CBMC statements.

For Item 1, rudimentary packet sending and receiving was reintroduced into the sending and receiving functions since all WSN protocol behavior depends on it. This was necessary since the hardware independent NULL platform abstracted away the functionality that transports packets to the transceiver chip, i.e., removed the body of all sending and receiving functions. With the added details, our abstract behavior model is even able to detect erroneous packet fragmentation and reassembling errors.

Note. Sensors are also not present in the NULL platform. But the implementation of the ESAWN protocol was using the node's IDs as sensor data to be transmitted, anyways, for clarity reasons. We did this for verification, too, reducing its complexity (since the IDs are unique). Alternatively, we used nondeterministic values.

For Item 2, features and structures that CBMC cannot handle were transformed. These were simple workarounds, like the ones in Listing 7.3.

For Item 3, we manually **instrumented the code** with CBMC assertions, i.e., we added $\text{assert}(f)$ statements for the properties described in Subsec. 7.2.4. All other assertions are inserted automatically by CBMC.

For Item 4, we introduced an **autostart function**: it imitates some of the omitted hardware functionality, especially the input from the environment, resulting in a closed system for a specific scenario. The autostart function sets a node into a specific state by inserting *STATUS* packets (cf. Subsec. 7.2.4) into its receive queue. Hereby, parameters as p and w and the topology of the network are communicated to the node. Furthermore, tasks are enqueued into the node's task queue to let the node perform certain actions like starting the processing of packets. Using nondeterministic parameters for the autostart function would enable full verification by covering all scenarios. Unfortunately, CBMC cannot cope with the complexity, which causes severe state space explosion and problems with complex data structures, like packets, since CBMC does not directly offer nondeterminism for them. Consequently, we reduced nondeterminism to measurement inputs from the environment and simple parameters like p and w where appropriate; so we chose the testing approach for all other values. Two examples for the autostart function are given in Appendix A.2. As described in Subsec. 7.2.4, we choose an exemplary topology and node in the topology for each verification.

```
RoundRobinSIZE= OU ? (OU - 1) / 8 + 1 : 0;
              ↓
RoundRobinSIZE = 0;


typedef nx_struct msg_t {
  nx_uint8_t head[sizeof(msg_head_t )];
  nx_uint8_t data[28];
  nx_uint8_t foot[sizeof(msg_foot_t )];
  nx_uint8_t metadata[sizeof(msg_metadata_t )];
} msg_t;
              ↓
typedef nx_struct msg_t {
  nx_uint8_t header[16];
  nx_uint8_t data[28];
  nx_uint8_t footer[1];
  nx_uint8_t metadata[16];
} msg_t;
```

Listing 7.3: Exemplary workarounds for CBMC

For Item 5, a node's task loop must be bounded by changing the scheduler which periodically executes the task loop. This is necessary since the nodes are reactive systems that process packets indefinitely (cf. Sec. 7.1), i.e., the original task loop is infinite. By limiting the execution number of the task loop, the model will run either until all tasks from the task queue are processed or an upper bound is reached, which we compute with injected code and check via CBMC assertions. When the task loop finishes, the node terminates. With this finitization, we were also able to further reduce the scheduler's complexity by replacing the complex functions for initializing the scheduler queue and the assignment of the empty task element with the necessary core functionality in the autostart function. With the help of our autostart function and nondeterminism, it is sufficient to show that individual packets are transported and processed in accordance with the protocol. Hence regarding the finite task loop is sufficient.

We modified the NULL platform with manual intervention, but the modifications for the task loop finitization and packet transportation can be automated easily, e.g., by introducing a verification platform into TinyOS. The autostart function cannot be completely automated, since the initialization depends on the protocol and contains the configuration we want to consider (cf. Appendix A.2).

With these modifications to the NULL platform, we get our abstract behavior model. From originally 21 000 lines of C code, as in the example of a real hardware platform (MicaZ nodes), the abstract behavior model only contains 4 400 lines of C code annotated with CBMC statements, but fully comprises the protocol behavior of the sensor node. Fig. 7.2 depicts an overview of the abstract behavior model.

In summary, we have a work-flow of verification as depicted in Fig. 7.3.

Simulation. Besides verification, the abstract behavior model can also be used for **simulation**, i.e., for imitation of the real program's execution to test the abstract

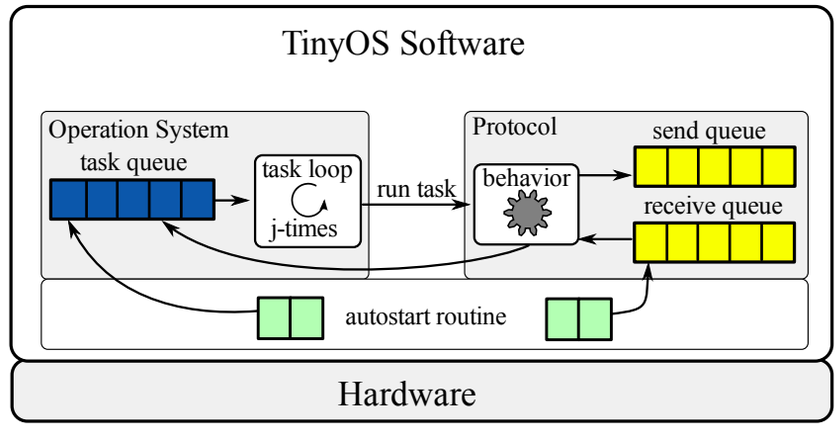


Figure 7.2.: Abstraction from TinyOS

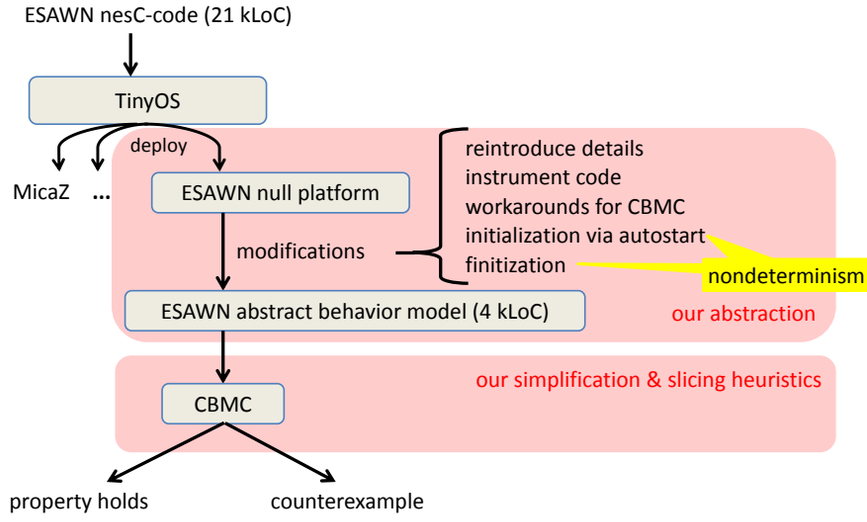


Figure 7.3.: Work-flow for our verification

behavior: We enriched the abstract behavior model with debugging statements and executed it. A few internal variables of TinyOS that were set nondeterministically in the model were now set to various specific values, removing all remaining underspecification. This simulation can be strengthened by setting these variables automatically (e.g., randomly, by tool environments like TEPAWSN [Man et al., 2009] or by a network simulators [Mahlknecht, 2010] like TOSSIM [Li and Regehr, 2010], NS-2 [Riley and Henderson, 2010] or OMNeT++ [Shing and Drusinsky, 2007]) until a desired coverage level is reached, to even better complement the verification process.

7.2.4. Property Specification

After acquiring the abstraction in Subsec. 7.2.3, we start specifying properties for the ESAWN protocol. Since the original code is intended for deployment, it has a limited scope of only a single node. Thus we need to specify **local properties** that do not consider distributed settings where messages are interchanged. This means we cannot specify properties that include two or more nodes, only the behavior and communication of one node at a time can be verified. Therefore, correctness is shown with local properties in an assume-guarantee style (cf. Subsec. 5.2.5). The environment is constructed using our powerful autostart function, partly by nondeterministically setting the network into all relevant states, partly by the testing approach, i.e., selecting an exemplary network topology and node in the topology. Then the corresponding desired behavior is checked at a single node (e.g., checking correct reception of a package instead of its full transportation in REQ4 below). This is achieved using `assert()` statements incorporated into the abstract behavior model, to be able to check the desired properties. At the end of this subsection, we will depict a solution for global properties.

We formulate the desired functional behavior as requirements (REQ), which are all translated into properties that are verifiable by $\text{assert}(f)$ statements. These assertions check whether the corresponding variables (e.g., a node's locally stored parameter w or outgoing packet queue) are set correctly. The assertions are located in the abstract behavior model either after the node's corresponding computation or within the alarm function that is built into the protocol. This instrumented function is then able to indicate wrong behavior of the protocol, potential attacks and also erroneous packets.

Besides these functional requirements, CBMC also covers the non-functional requirement of runtime correctness via its built-in standard runtime error checks (cf. Subsec. 7.1.1).

STATUS Packets

The entries of the *STATUS* packets are encrypted with an RC5 cipher. The encryption procedure was automatically removed in the NULL platform, which helps avoid state space explosion in the abstract behavior model without changing the underlying protocol. So nodes send their data as plain text.

Initialization of the network is done by our autostart function. Since full verification by covering all scenarios with nondeterministic parameters was too complex (data structures and state space, cf. Sections 7.1 and Subsec. 7.2.4), we took the testing approach and mostly picked one scenario that already exhibits most of ESAWN's computation

and communication operations: the topology described in Subsec. 7.2.2 (cf. Fig. 7.1) with $w = 2$ and $num\_nodes = 5$. For the probability value p, we firstly chose a nondeterministic value, but finally verified with $p = 1$: since we are checking functionality, not probabilities for desired results, and $p = 1$ is the strictest possible authentication, verified REQ1 to REQ6 below for $p = 1$ still hold for $p < 1$, i.e., when some checks are randomly omitted. But with $p = 1$, we can fully avoid the complexity caused by probability, i.e., the random variables, their computations, and pseudo-randomness. There is an alternative to using $p < 1$ for investigating cases where checks are omitted: Settling for qualitative instead of quantitative inspections in the abstract behavior model, i.e., using CBMC's nondeterminism instead of probabilistic choices. Hence we are still able to avoid probability computations and pseudo-randomness, and still investigate all possibilities of the probabilistic concast in a single verification run (and do not have to deal with probabilities converging zero since we only have finite executions [Faragó, 2007]). The disadvantages in this approach are the additional complexity that the nondeterministic choices might cause and the loss of quantitative results, i.e., all runs are treated equally no matter their probability. Hence this is future work.

Whether the values are set correctly by the autostart function is checked via the assertions for the following three requirements for the respective *STATUS* packets:

REQ1 covers packets of type *SET*, which are sent initially by the base station to make protocol parameters known to the network. REQ1 states that a node processes this type of packet correctly:

$$SET(num\_nodes, w, p) \text{ sets variables correctly} \tag{REQ1}$$

REQ2 considers packets that make the aggregation tree public using *SETAGG* packets. For this reason, each node is informed about its successor nodes that it will send packets to. The *SETAGG* packets contain the fields *node_id* and *parent_id* and must be sent to every node in the network:

$$SETAGG(node\_id, parent\_id) \text{ sets variables correctly} \tag{REQ2}$$

REQ3 is about protocol conform behavior after receiving a *GO* packet: Only leaf nodes initiate concasts and the frequency value f in the *GO* packet (cf. page 163) must be respected:

$$\text{correct action upon reception of } GO(f) \tag{REQ3}$$

We omit the trivial requirements for the packet type *RESET*, which causes a hard reset of the node, and for *ALARM*, which is simply forwarded.

ESAWN Packets

Entries of ESAWN packets are encrypted using symmetric keys (cf. SKEY [Zitterbart and Blaß, 2006]). Again, we consider unencrypted packets instead. Similarly to *STATUS* packets, we also split the correct handling of ESAWN packets into several requirements.

REQ4 requires that ESAWN packets are correctly transported. This also implies that packets have been correctly aggregated and are correctly forwarded (e.g., correct computation of the relay count). With our testing approach, we chose the sum as

aggregation function f_{agg} ($f_{agg}(a,b) = a +_{\text{int}} b$) and check REQ4 exemplarily for the packet P that contains D_1, agg_1 sent to n_2:

$$\text{correct reception of packet } P \tag{REQ4}$$

REQ5 requires that ESAWN packets are correctly authenticated (which also implies correct aggregation). For this, a node n_i has to alarm if any of the last w aggregates is incorrect (n_0 to n_w can only check fewer aggregates):

$$(\exists j \in \{1,..,w\} : agg_{i-j} - D_{i-j} \neq agg_{i-j-1}) \iff alarm_i \tag{REQ5}$$

Finally, REQ6 checks that this $alarm_i$, a certain alarm function built into n_i, behaves correctly, i.e., issues an $ALARM$ packet to be sent. We do this by checking whether $ALARM$ packets are put in the outgoing packet queue out_{n_i} of n_i:

$$alarm_i \implies ALARM \text{ packets in } out_{n_i} \tag{REQ6}$$

Global Properties

Global properties can achieve stronger and more concise formulations, for instance: `if some node alarms, then eventually the sink will receive an ALARM packet`. Since we are verifying the derived code that can be deployed on a sensor node, the verification process used local properties, which cannot handle multiple nodes.

To imitate multiple nodes in a distributed settings with packet communication, we implemented simple **multitasking** between nodes: When the current node sends out a packet, a **context switch** between nodes takes place. For this, we modified TinyOS's send routine: The local variables of the current node are saved and the local variables of the destination node are loaded. The packet being sent is enqueued into the receive queue. With this, a distributed network behavior can be imitated to some degree, with packets being sent to their destination without delay.

The trade-off for using global properties is an increased complexity. Therefore, we successfully verified only very simple properties and leave more powerful global properties as future work, after local properties no longer cause problems (cf. next subsection).

7.2.5. Verification Results

For the verifications, we used CBMC version 2.9 with our additional optimization heuristics (cf. Subsec. 7.1.3), some bug fixes related to complex data types and compiled for 64bit processors because some verifications required a lot of memory (see below).

As described in the previous subsection, the generated code is manually instrumented with the assertions that specify REQ1 to REQ6. All other assertions are inserted automatically by CBMC. For most verifications (cf. Subsec. 7.2.4), we fixed node n_2, which exhibits all behavior relevant to our verification.

Besides the REQs from Subsec. 7.2.4, we also checked whether the number of unwindings was sufficient, as well as code safety properties, array index out of bound and illegal pointer access. All checks have to be performed individually for each REQ since the autostart function is adjusted to each REQ.

SET packets			*SETAGG* packets			*GO* packets		
check	correct	|claims|	check	correct	|claims|	check	correct	|claims|
REQ1	yes	6	REQ2	"no"	4	REQ3	yes	4
unwind	yes	37	unwind	yes	37	unwind	yes	37
bounds	yes	60	bounds	yes	60	bounds	yes	59
pointer	"no"	181	pointer	"no"	177	pointer	"no"	175

Table 7.1.: Verification results for *STATUS* packets for a valid loop unwinding of 4.

To be sure the verification of all other properties is complete, the first verification step is finding and checking the required number of loop unwindings, using the corresponding assertions. Our abstract behavior model has 32 loops in total (which CBMC shows when the parameter −−show−loops is used). For one loop, we were able to infer the required unwindings a priori: It belongs to a memset function, which has to be iterated often when duplicating memory locations. Therefore, we set the required unwindings to a sufficiently high and safe value of 20. The unwindings for the other 31 loops had to be determined iteratively by automatically running the unwinding check provided by CBMC and incrementing the unwinding setting if the unwinding assertion failed. Our early satisfiability detection was critical to achieve this (cf. Subsec. 7.1.3).

Table 7.1 displays the performed verifications for the *STATUS* messages, their results and number of required claims, i.e., manually or automatically inserted CBMC assertions (which CBMC shows when the parameter −−show−claims is used). An unwinding depth of 4 was sufficient, and all unwinding checks passed successfully. Array index out of bound checks always passed successfully, too. The pointer checks were violated for every packet type: The cause seems to be no real violation, since debugging the source code of CBMC showed that this is not a failure of the protocol, but CBMC does not find correct symbols during its pointer-analysis. REQ1 and REQ3 were successfully proved correct. REQ2 was violated: The cause seems to be no real violation, but that CBMC is unable to handle arrays of structures, which are heavily used for the queues. This is one example where CBMC does not scale related to data type complexity.

Besides verifying these properties, we raised our confidence in the correctness of ESAWN by successful simulation (cf. Subsec. 7.2.3) and fault injections (cf. Sec. 2.5) in the code and in the assertions, all of which CBMC found.

Unfortunately, we were not able to verify REQ4 to REQ6 because the unwinding checks were problematic: At first we had difficulties setting the loop unwindings just as high as necessary, which is crucial. For instance, when we set the unwindings to 11 for all loops, CBMC requires 30GB of RAM (and over 3 hours) to detect that not enough unwindings were made. For 12 unwindings, CBMC gives segmentation faults because 32GB are exceeded. We solved the difficulty of finding the smallest possible unwinding value for each loop by searching automatically. But as the search is very time-consuming, it is important to start with sensible values. When we used 20 unwindings for the first loop (`memset`, which needs to be able to copy values sufficiently often) and 6 unwindings for all others (using the parameters `-unwind 6 -unwindset 1:20`), verification came much further with much less memory: With 2.5GB, CBMC reached the stage `passing to decision procedure`. Unfortunately, CBMC then halts with the error message `unexpected array expression: typecast`. Because CBMC

aborted with a typecast exception, we tested whether the unwindings might be sufficient by injecting a fault into one of our assertions for REQ4 to REQ6. But these verifications also caused typecast exceptions. This shows again that CBMC does not scale with data type complexity. CBMC offers two alternatives when enough unwindings cannot be reached efficiently: Firstly, paths with more unwindings can simply be ignored using assume(f) statements. But this leads to a bad path coverage: in our case, a lot of packets in the queue need to be processed for initialization; thus the processing of the ESAWN packets – and therefore their bugs – would not be reached. Secondly, we could have used nondeterminism at points where the maximum unwindings are reached. But this might cause true underspecification (cf. Sec. 7.1), and severe state space explosion. Furthermore, we would need to generate packets nondeterministically; because of CBMCs difficulties with complex data types, it cannot directly create them nondeterministically. Hence the only solution would be the cumbersome manual implementation of nondeterministically generating a protocol-conform sequence of packets whenever maximum unwindings are reached. But that would modify the abstract behavior model strongly and counteract our intent of an automatic verification process.

7.3. Conclusion

7.3.1. Summary

We have described a proof of concept for automatic SBMC verification for the realistically large (21 000 LoC) and complex WSN protocol ESAWN. The verification can be integrated into the WSN development and deployment platform TinyOS and thus into the software development process. To be able to handle such large scale programs, the verification and its integration must be automatic and require the abstractions and optimization heuristics we provided. These insights are used in Part III, where full automation and flexible heuristics are major design decisions.

Our process generates an abstract behavior model that is then verified by CBMC. The developer can automatically check for pointer dereference and array index out of bound errors, and for user-supplied functional properties described by assert(f) and assume(f) statements.

We were able to prove correctness for about half of the properties: for the *SET* and *GO* packets, but not for the *SETAGG* and ESAWN packets, due to technical difficulties in CBMC, e.g., unsupported arrays of structures, pointer bugs and typecast exceptions. It shows that, in our case, CBMC does not scale well with the complexity of data structures. Many of the technical difficulties in CBMC were caused by large function parameters ($\approx$ 500 byte) in the source code of ESAWN. So in general, our case study is a proof of concept that large programs of practical size can be handled by SBMC, but many technical difficulties exist. Unfortunately, we do not know whether SBMC could cope with the properties that now cause technical difficulties once these technical difficulties are fixed.

In some cases, the large function parameters can be considered a design flaw in ESAWN since frequent, unnecessary copying (because of C's call-by-value evaluation) is inefficient; we have informed the developer of ESAWN about this.

Our optimization heuristics (cf. Subsec. 7.1.3) improved the scalability in data type

complexity, and even more the scalability in the size of verifiable code: Without them, state space explosion prevents verification of even the simplest instances for the ESAWN protocol. A general lesson learned is that recent advances in compiler optimizations for the generation of runtime code can also improve static analysis mechanisms in real world settings. Our abstractions for the abstract behavior model (cf. Subsec. 7.2.3) also improved scalability in the size of the code and additionally allowed hardware independence. Using our optimization heuristics and abstractions, we have seen that CBMC can be employed in the verification process for large scale programs. But CBMC does not scale sufficiently with data type complexity, which can probably be solved by switching to other SBMC tools (cf. following subsection). Furthermore, SBMC with our optimization heuristics still does not scale sufficiently to handle all scenarios nondeterministically without severe state spaced explosion. Therefore, we switched to lightweight verification by adopting the testing approach to SBMC: Not all possible scenarios are considered at once by nondeterminism (see Subsec. 2.3). Instead, relevant scenarios are selected manually: SBMC does cover all possible situations within the selected scenarios, resulting in a sensible overall coverage, and the selection enables CBMC with our optimization heuristics to check our large code bases. This approach motivates Part III of this thesis, which focuses on this the combination of formal methods and testing.

7.3.2. Related Work

After enumerating alternative SBMC tools that could be used for our approach, we describe work related to our approach of testing with SBMC. Thereafter, we enumerate work that applied verification to WSNs.

The award-winning [Beyer, 2013] **Low-Level Bounded Model Checker** (**LLBMC**) [URL:LLBMC; Falke et al., 2013a; Faragó et al., 2014] implements the approach of applying a compiler frontend to verify an intermediate language (cf. Subsec. 7.3.3). LLBMC is a SBMC tool with a bit-precise memory model that employs the ACM award-winning [URL:ACMAWARD] **LLVM compiler infrastructure** [URL:LLVM]. LLVM provides modern compiler, optimizer (e.g., promote memory to register transform pass) and static analysis (e.g., scalar evolution analysis) tools built around the established SSA-based **LLVM intermediate representation** (**LLVM IR**). LLVM provides compiler frontends for many languages, such as ActionScript, Ada, C, C++, C#, Common Lisp, D, Fortran, GLSL, Go, Haskell, Java bytecode, Julia, Lua, Objective-C, Python, R, Ruby, Rust, Scala, and Swift; and compiler backends for various instruction sets, such as x86/x86-64, ARM, and PowerPC [URL:LLVM; Berg, 2014]. Since LLVM is a very strong compiler and optimizer, many of the difficulties in this chapter (too complex data structures and requiring optimization heuristics) can be avoided. Therefore LLBMC can support all C constructs, even not so common features such as bitfields. After translating LLVM IR to the **intermediate logic representation** (**ILR**), LLBMC's additional optimization heuristics use rewrite-based simplifications that are not already covered by the LLVM compiler [Sinz et al., 2012; Falke et al., 2013b]. To avoid inefficient translations to SAT, LLBMC translates ILR to SMT-LIB, performs some final optimizations and uses the SMT solver Boolector or STP. These optimizations on three different levels perform better than optimizations on fewer levels. LLBMC offers both explicit and implicit nondeterminism (cf. Sec.7.1) and user-supplied checks via $\text{assert}(f)$ and $\text{assume}(f)$. The

built-in standard checks are integer overflow and underflow, division by zero, invalid bit shift and memory errors (invalid memory access, array index out of bound, stack buffer overflows, illegal pointer access, invalid frees).

The **Verifier for Concurrent C** (**VCC**) [URL:VCC] is a SBMC tool currently developed at Microsoft Research. It is roughly similar to CBMC and LLBMC, but offers specifications via design-by-contract and translates the annotated source code to **BoogiePL** [URL:BoogiePL], an intermediate representation that is a typed procedural language for program analysis and verification [Cohen et al., 2009].

Besides CBMC, LLBMC and VCC, **FAuST** [Holzer et al., 2008] is another SBMC tool: It is a highly customizable SBMC algorithm which integrates verification, bug finding, equivalence checking, test case generation and execution (cf. Note 7.1). Like LLBMC, FAuST builds its verification on top of LLVM IR. Furthermore, it uses LLVM's JIT compiler for execution of generated test cases.

There are other approaches that combine testing with constraint solving; consequently, they are affected by the deficits of the applied constraint solver. We start with related work that performs static testing; but the related work shows that the boundary between static and dynamic methods are no longer sharp.

[Merz et al., 2010] introduces **abstract testing**, which are contracts derived from requirements. Preconditions are formulated by $\mathrm{assume}(\cdot)$, postconditions by $\mathrm{assert}(\cdot)$. Both are checked using CBMC, to perform static unit testing with high coverage. [Merz et al., 2010] also contains a case study with a safety-critical automotive software project. In comparison, [Merz et al., 2010] has a similar lightweight approach as this chapter, but its motivation is to increase the coverage of combinatorial testing for unit tests, whereas this chapter's motivation is reducing complete verification to a more lightweight approach to increase feasibility of SBMC while retaining generality for specific, manually chosen scenarios, i.e., full path and input coverage for relevant scenarios. The approach in this chapter can be considered static acceptance, system or integration testing.

Testing with SBMC has similarities to execution-based model-checking (EMC) [Kundu et al., 2011]: both explore all possible behaviors for a given input. But EMC tools like **Verisoft** [Godefroid, 1997] and **Java Pathfinder** [Visser and Mehlitz, 2005] do not use implicit state MC, but implement execution-based explicit state MC (cf. Subsec. 5.2.1) using an own runtime-scheduler, usually also adding reductions (cf. Sec. 5.4). Therefore, execution is **simulated** in an own environment, e.g., (**Symbolic**) **Java Pathfinder** (**JPF**) instruments and (symbolically) executes Java bytecode in an own Java Virtual Machine to perform EMC. JPF is a good example that the boundaries between static and dynamic methods are no longer sharp.

Techniques based on black-box testing will be investigated in Part III. Among the vast amount of dynamic white-box testing tools [URL:whiteboxToolsHP], several are based on the idea of instrumenting the source code such that its execution generates its own TCs automatically. They instrument the source code and run it with some of the inputs being symbolic. Hence this approach is sometimes called **dynamic symbolic execution** (**DSE**) [Godefroid et al., 2005; Sen et al., 2005b; Su et al., 2015b,a].

EXE [Cadar et al., 2008b] is an efficient white-box test generation tool with search heuristics to achieve high execution path coverage of the source code. For each input i, the test engineer chooses i to be either a concrete value (random or user-supplied) or to be symbolic. Other memory locations that do not correspond to input can be

treated similarly. During execution, instructions containing symbolic values are **executed symbolically**, i.e., the constraints describing the control flow and side effects of the instructions are tracked along all paths (resulting in **symbolic paths**). All other instructions are executed concretely. Symbolic branch points are resolved by forking program execution. Due to the high number of symbolic paths that often occur (called **path explosion**), they are not all investigated concurrently, but prioritized by so-called search heuristics: paths that are likely to increase code coverage are preferred. By default, a DFS is performed by randomly picking one of the two branches to investigate first; alternatively, a mix between DFS and best-first is performed, where heuristics pick a symbolic path based on its run so far, e.g., how many steps and increments of the coverage level it had so far. Symbolic inputs enable to cover multiple values and thus multiple concrete paths within one symbolic path. Searching through symbolic paths in this way is sometimes called **explicit symbolic path model checking**, shortly called **explicit path MC** (which is similar to JPF's approach with symbolic execution). EXE checks for the usual runtime errors (arithmetic errors and memory errors, cf. Subsec. 7.1.1) and generates a test suite that exhibit these errors and high code coverage. For this, EXE solves the constraints on the corresponding symbolic path with bit-precision using the SMT solver STP. Besides bit-precision, EXE's strength are high code coverage over a broad range of source code. Its weakness is the inherent path explosion, which often breaks completeness.

The open source tool **KLEE** [URL:Klee; Cadar et al., 2008a] is a redesign of EXE and part of the LLVM Compiler Infrastructure [URL:LLVM]. It is implemented as a virtual machine for LLVM IR. Since prioritization of paths to find bugs and increase code coverage is a challenge, the heuristics have been improved by computing weights for each path based on its run so far, and by sophisticated scheduling. Compared to testing with SBMC, KLEE has the advantage that the environment can be set up via parameters and files, without the need of an initialization function. Our Abstract Behavior Model offers, however, a flexible initialization that can also cope with a very complex setup, as is required for ESAWN and WSN topologies. KLEE's heuristics automatically search for paths that seem best, while testing with SBMC selects scenarios manually. But when path explosion occurs, KLEE's verification is no longer complete due to its heuristics, not even for the scenarios selected in this chapter. KLEE has difficulties with symbolic pointers and function pointers [Qu and Robinson, 2011]. Like testing with SBMC, KLEE can now perform functional verification, via klee_assert(·) and klee_assume(·).

The **directed automated random testing** tool **DART** [URL:KoushikSensProjects; Godefroid et al., 2005] is based on a similar explicit path MC approach, called **concolic testing**: it performs a *conc*rete execution with symb*olic* execution on top, for input variables defined as symbolic by the test engineer. Concolic testing starts execution with a random concrete input; for successive runs, iteratively a constraint of the current path condition is negated and the resulting path condition solved to get a new concrete input, which is used for the next iteration and to generate a TC. The negations are chosen in a way to drive concolic testing through new execution paths in a bounded DFS fashion, which leads to test suites with higher code coverage than random testing (cf. Sec. 2.5). DART checks for the usual runtime errors (arithmetic errors and memory errors, cf. Subsec. 7.1.1) and non-termination, similar to SBMC (cf. Subsec. 5.2.3).

The **concolic unit testing engine** (**CUTE**) [URL:KoushikSensProjects; Sen et al.,

2005b,a] is a successor of DART that also handles data structures: **logical input maps** represent all inputs, including memory graphs, using a collection of scalar symbolic variables and approximate pointer constraints (e.g., for alias analysis). After simplifications, the approximate pointer constraints have the form $x = y$ or $x \neq y$, which often suffice in practice. But sometimes, pointers with symbolic offsets are needed, which CUTE cannot handle [Cadar et al., 2008b], hindering interprocedural analysis and completeness. Additionally, CUTE also has KLEE's deficits compared to testing with SBMC. Like KLEE and testing with SBMC, CUTE can now perform functional verification, via CUTE_assert(·) and CUTE_assume(·).

Most work on applying SBMC for testing instruments the code and then applies CBMC to generate test cases for dynamic testing that satisfy some coverage criterion like statement coverage, decision coverage or MC/DC:

The tool **FSHELL** [Holzer et al., 2008] is a frontend for CBMC for white-box testing: It offers a language to query program paths, based on regular expressions and coverage criteria, enabling test engineers to formulate, manage and execute test jobs. This frontend turns CBMC into a versatile testing environment for interactive exploration or automatic test case generation.

In [Angeletti et al., 2009c; Rosa et al., 2010; Angeletti et al., 2009a,b], CBMC is applied to achieve branch coverage on C code from the railway domain. One project [Angeletti et al., 2009a] has 250 000 LoC, and for 13% out of 31 modules CBMC gives a timeout, for the others full branch coverage is achieved, and the approach yields a dramatic increase in the productivity. Another project [Rosa et al., 2010] has 30 000 LoC, and CBMC gives timeouts in less than 3% of the cases.

In [Venkatesh et al., 2012], CBMC is applied for MC/DC coverage based on an instrumentation approach from [Bokil et al., 2009]. The industrial project from the automotive domain has 50, 000 LoC of C code, but CBMC crashed for about 20% of the 1241 coverage tasks, and gave a timeout for one.

The German master's thesis [Liu, 2015] does not use SBMC as an aid for test case generation to achieve some coverage criterion, but extends the SBMC tool LLBMC to generate a test case from a counterexample from LLBMC. By choosing functions to be mocked and an entry point in the counterexample, where the appropriate state is reconstructed for the test case to start, LLBMC can generate unit, integration or system tests. So the focus of the work is on tightly integrating SBMC and dynamic testing in the software development process, for instance to dynamically check, debug or regression test mistakes found by SBMC.

The work described in this chapter was the first published work [Werner and Faragó, 2010] known to the author of this thesis that uses software bounded model checking for verification of WSNs. But other verification techniques were also applied to WSNs at that time, mainly based on MC or symbolic execution. Most approaches for protocol verification in WSNs use either a heavy abstraction from the actual implementation or only consider parts of the model behavior.

The **T-Check** tool [Li and Regehr, 2010] builds on TOSSIM and provides state space exploration and early bug finding. Similar to our approach, the authors use a combination of model checking and more lightweight techniques to combat the complexity due to nondeterminism. Their lightweight techniques are random walks and heuristics. The results show the applicability of the tool and that it can find violated properties. But

this random search is not as exhaustive as our approach, and strongly depends on the implemented heuristics [Killian et al., 2007], yet the user's experience is still required.

The **Anquiro** tool [Mottola et al., 2010] is used for the verification of WSN software written for the Contiki OS. To cope with the state space explosion, the user can select from different levels of abstraction, depending on his property of interest. In comparison, our abstraction only eliminates direct function calls to the hardware and assembler constructs. Thereby, our abstraction is even able to detect erroneous packet fragmentation and reassembling errors. This is closer to the actual implementation – at the cost of complexity. In addition, since we use CBMC and its transformation mechanisms, we are able to directly point to the fault, i.e., the violating line of code, instead of trying to backtrack the sequence of actions for the violated property.

In [Kothari et al., 2008], an approach is described that automatically derives a high-level program representation from low-level nesC implementations in TinyOS. For this, symbolic execution is adopted to handle the event-driven nature of the TinyOS framework. The approach is implemented in a tool called **FSMGen**. In detail, generic components are provided that approximate the behavior of sensor network components. The resulting finite state machine representation of components is obtained by predicate abstraction. Since the approach uses coarse approximations of the event-driven execution model of TinyOS, not all possible execution paths are represent. Furthermore, the approach is in general not applicable for low-level interrupt driven code. The largest conducted case study (FTSP) consists of 255 states.

The work of [Hanna, 2007] proposes a tool called **Slede** for bug finding in security protocols. This tool extracts a model from a provided nesC implementation in TinyOS. From the protocol specification, an intrusion model is automatically generated. Finally, PROMELA code (cf. Subsec. 3.4.3) is generated and verified with SPIN (cf. Subsec. 5.5.1). We performed some experiments with Slede and found out that only old protocol implementations written in TinyOS 1.0 can be used as input for the verification tool. Furthermore, the implemented intrusion model is restricted to the Dolev-Yao threat model. Finally, the real implementation is heavily abstracted from.

KleeNet [URL:KleeNet] is a DSE tool based on KLEE, which is described above, enabling test case generation for distributed systems. [Sasnauskas et al., 2010] integrated KleeNet into the Contiki OS to detect bugs due to nondeterministic events in WSNs by automatically injecting nondeterministic failures. The case study detected four insidious bugs in the μIP TCP/IP protocol stack. An advantage of KleeNet is its support to check global properties (cf. Subsec. 7.2.4) using distributed assertions. Though the search heuristics achieve high coverage, they cannot be influenced by the user to achieve 100% input and path coverage even for small parts, as SBMC can.

The **FeaVer** verification system [Holzmann and Schmith, 2000] focuses on coping with the feature interaction problem (cf. Subsec. 7.2.3). This has long been a problem in telecommunications systems [Keck and Kühn, 1998] and is currently being researched in product lines [Padmanabhan and Lutz, 2005] and feature oriented software development [Apel and Kästner, 2009]. FeaVer offers means to mechanically extract a reduced verification model from implementations in C, controlled by a user-supplied lookup table of feature requirements: Parts relevant for the currently investigated features are included and possibly abstracted using nondeterminism, whereas irrelevant parts are sliced from the model. The verification is performed by SPIN. Due to the abstractions

and slicing, FeaVer may become unsound, i.e., yield false counterexamples.

The work [Bucur and Kwiatkowska, 2009] and [Bucur and Kwiatkowska, 2010] considers the application of verification techniques to software written in TinyOS (more precisely, in the TosThreads C API). Instead of analyzing an integrative model with an operating system part and a protocol implementation, low level services are modeled and statistically verified individually against safety specifications. For verification, **SATABS** [URL:SATABS] was employed, which performs predicate abstraction using SAT and can handle ANSI-C and C++ programs. In this work, the overall checked model size is at most 440 LoC. In our approach, we apply our abstraction to the more complex ESAWN protocol consisting of 21 000 LoC and obtain a model with about 4 400 LoC, which we subsequently check.

Insense [Sharma et al., 2009] is a composition-based modeling language which translates models in a concurrent high-level language to PROMELA code to enable verification of WSN software by SPIN. A complete model of the protocol under investigation has to be created, though, even if an implementation, e.g., in TinyOS, already exists. This is very time intensive and error-prone. Though SPIN is very well capable of analyzing concurrent and distributed settings, [Werner and Steffen, 2009] shows severe state space explosion when model checking an abstract behavior model of ESAWN with small topologies in SPIN.

7.3.3. Contributions

Most related work that combines model checking with a testing approach generates dynamic tests with MC/DC or branch coverage. But it strongly depends on the situation whether these coverage criteria yield meaningful test suites (cf. Sec. 2.5 and Subsec. 12.3.1). Our testing with SBMC checks all situations for manually chosen scenarios (i.e., achieves path and input coverage for them), but not outside these scenarios. Together with our abstract behavior model and optimization heuristics, verification of WSN code of practical size of originally 21 000 LoC reaches feasibility.

Our case study shows that scalability with the complexity of data structures is very important for the successful verification of programs of practical size. CBMC (and VCC) can improve by not only supporting flat C data types, such as a single struct or array, but also their closure, i.e., nested types. A different solution is using a compiler frontend to translate C code into an intermediate language, which is also in line with the lesson we learned about recent advances in compiler optimizations also improving static analysis, and with the current trend in formal methods to decouple input processing from the actual analysis by introducing an intermediate representation [Blom et al., 2010; URL:BoogiePL; Cohen et al., 2009; Ulbrich, 2014; Falke et al., 2013a].

7.3.4. Future Work

As further SBMC tools emerge and improve, our case study can be used as benchmark for them:

Aarti Gupta has shown interest in our work to benchmark NEC's F-Soft [URL:F-Soft],which has grown into a full framework providing various static analyses and model checking techniques where the size and complexity can be reduced across multiple

stages [Gupta, 2008].

We alternatively tried applying the Verifier for Concurrent C, but experienced similar problems as in the first steps with CBMC: Pointer constructs present in the generated model could not be handled correctly and resulted in a syntax error while parsing. This shows that handling complex data types in SBMC tools is problematic, but a necessary improvement for verifying programs of practical size.

To check whether our case study can in general be conducted with LLBMC, we recently verified REQ1 with LLBMC version 2013.1 after removing some workarounds required for CBMC (cf. Subsec. 7.2.3) and changing some CBMC specific syntax to the LLBMC counterpart. All of LLBMC's built-in standard checks, plus unwind and REQ1 were verified on a Mac mini, 2.6 GHz Intel Core i7, 16 GB RAM, requiring 232 seconds (mainly 183 seconds for transformations and 37 seconds with STP and MiniSat) and less than 8 GB memory. Using LLVM's promote memory to register and scalar evolution analysis, the maximal trip count of many loops can be derived automatically, i.e., the maximal number of times the loop exit condition chooses to stay in the loop, which is equal to the required number of loop unwindings. This is very helpful since it was difficult and time-consuming to find the smallest possible unwinding value for each loop with CBMC (cf. Subsec.7.2.5). Because of this efficiency, it is interesting future work to conduct the whole case study in LLBMC with increased nondeterminism and a reduced testing approach, for higher coverage. Furthermore, the SBMC tool FAuST can be considered, which is similar to LLBMC.

For a fair comparison, the adapted case study should also be repeated with the newest version of CBMC, which might yield better results. We had checked for improvements with CBMC version 3.6, but encountered segmentation faults, e.g., when passing the problem to propositional reduction, already with 4 unwindings. This indicates that CBMC does not include our optimization heuristics (cf. Subsec. 7.1.3) or similar ones. However, CBMC offers encodings for SMT solvers instead of SAT solver since version 4.0, so this is interesting future work. In general, SBMC tools have evolved strongly in the last few years: while our case study with 21 000 LoC was one of the largest when the case study was conducted in 2010, competition benchmarks from 2013 contain several examples with 10 000 LoC to 80 000 LoC and even beyond, but the larger examples are meant as smoke test, not for successful verification [Beyer, 2013]. But for practical code, state space explosion still causes feasibility problems quickly.

We can also consider exhaustive model checking tools, e.g., use our generation of the abstract behavior model and apply SATABS afterwards (cf. Subsec. 7.3.2). If this approach is infeasible, a combination of SBMC and predicate abstractions (cf. [Post et al., 2008]) might be able to cope with our large protocol.

We successfully verified simple global properties, but left more powerful global properties as future work (cf. Subsec. 7.2.4) since they are more complex and the complexity of local properties already caused problems. Many extensions and improvements are possible for verifying global properties, e.g., enhancing our abstract behavior model by improving multitasking between nodes: We can reduce the large memory requirement by not storing the local variables (e.g., w, p and the whole aggregation tree) of all simulated nodes independently, but only once. We can also implement a more general multitasking that allows several leaf nodes and delayed transmission of packets. This can be achieved by using one extended scheduling function that comprises all jobs of all simu-

lated nodes. With these improvements, all distributed properties verified in [Werner and Steffen, 2009] with SPIN (cf. Subsec. 5.5.1) using hand-written models will be verifiable automatically.

Thoroughly investigating the protocol's robustness and fault-tolerance is another important research direction, made possible by the powerful autostart function. Since it can set all state variables to arbitrary values, also hazardous situations can be constructed. In this situation, settling for qualitative inspections instead of using a probabilistic value p (cf. Subsec. 7.2.4) is also interesting future work.

Incorporating our generation scheme (simulation features inclusive) as a **verification platform** into TinyOS will help developers of WSN protocols to verify correctness of their implementations.

Notes 7.1. Since our testing with SBMC does not achieve path and input coverage outside the chosen scenarios, integrating dynamic testing with SBMC and coverage-based test case generation with SBMC is interesting future work, although it deviates from our static approach. This coverage-based test case generation with SBMC is a current trend and planned as extension of LLBMC and [Liu, 2015]. To generate more meaningful test suites, the extension to OMC/DC is intended (cf. Sec. 2.5). The extension is alleviated by the SSA-based LLVM IR.

As alternative to coverage criteria, LLBMC aiding mutation testing is also interesting future work, since this can further increase meaningfulness of test suites. LLBMC can help detect equivalent mutants, which is often costly and prohibits a broad success of mutation testing [Frankl et al., 1997; Grün et al., 2009; Madeyski et al., 2014]. LLBMC's equivalence checks [Falke et al., 2013a] are a promising approach to tackle this problem. As potential further improvement, a single meta-mutant in schema-based mutation can be used, where all mutants are encoded in the meta-mutant [Offutt and Untch, 2001; Jia and Harman, 2008, 2011]. LLBMC can then efficiently check equivalence by asserting equivalent original input (i.e., modulo the input for triggering still active mutants) but different output.

Such an approach has already been implemented in FAuST [Riener et al., 2011], using an LLVM transform pass to generate the meta-mutant. FAuST cannot, however, handle pointers and arrays, and was only applied to source code of less than 1 000 LoC.

Part III.

Model-based Testing

8. Input-output conformance theory

8.1. Introduction

This chapter introduces the **ioco theory**, which formalizes black-box, functional testing, enabling to precisely formulate

- when an SUT **conforms** (i.e., complies) to a specification;
- an abstract test case generation algorithm;
- when a test suite (e.g., generated by the algorithm) is sound and complete (i.e., exhaustive, cf. Sec. 8.8).

Therefore, the ioco theory is a tool-independent foundation for **conformance testing** [ITU-T Study Group 10, 1997; ISO Information Technology, 1992], i.e., for checking whether the observable behavior of the SUT conforms to its expected functional behavior, described by a system specification. The ioco theory helps design concrete test case generation algorithms and compare them, the overall methods and the tools for test case generation.

The **ioco** relation determines which SUTs from the set $\boldsymbol{SUT}$ of all possible SUTs conform to the specification. The *ioco* relation can be used in test generation algorithms to derive a test suite from the specification to check the SUT for conformance. By traversing the specification $\mathcal{S}$ and nondeterministically choosing amongst all controllable choices, but keeping all uncontrollable choices, i.e., choices that the SUT can take nondeterministically, the ioco algorithm generates test cases which are themselves LTSs. They are sound, and the generated test suite containing all possible test cases is complete. A test case is run by executing it synchronously in parallel with the SUT.

To be able to also treat the real, physical SUT formally, it needs to be abstracted to a mathematical structure (just like other sciences use mathematical objects to talk about physical objects).

Though the ioco theory is covered by many publications [Brinksma and Tretmans, 2000; ITU-T Study Group 10, 1997], this chapter contributes several new results, mainly:

- the aspects of abstractions covered by the testing hypothesis, the test adapter and nondeterminism;
- more flexible semantics: different kinds of LTSs and semantics for internal transitions;
- investigating reduced exhaustive test suites, covering a deterministic test case generation algorithm, coverage criteria, fairness, and exhaustiveness thresholds.

The other parts are roughly based on [Tretmans, 2008; Frantzen, 2016], but more flexible and concise with the help of the general formalisms introduced in Sec. 3.4.

Roadmap. Subsec. 8.1.1 introduces the most visible abstractions: those on the interface. Thereafter, Subsec. 8.1.2 investigates the full testing hypothesis. Subsec. 8.1.3 gives an overview of the remainder of this chapter.

8.1.1. Interface Abstraction

The testing hypothesis helps bridge the gap between the formal specifications and the informal, real system under test. The first step is abstracting the SUT's interface, described in Def. 8.1.

Definition 8.1. We **abstract the SUT's interface** by restricting the possible
- **stimuli** for the SUT: We consider all stimuli as **input** to the SUT since they are **under the tester's control**; we only use a countable set $\boldsymbol{I}$ of relevant input;
- **observations** from the SUT: We consider all observations as **output** of the SUT since they are **under the SUT's control**; we only use a countable set $\boldsymbol{U}$ of relevant output.

We also include the observation that is made when some $i \in I$ is inhibited by the SUT, i.e., we do not abstract from it. Thus, the SUT always accepts all inputs (cf. Subsec. 8.2.6), but may return an output indicating that the SUT inhibits the input (cf. coffee machine example below).

Example 8.2. For a coffee machine (one of the most popular toy examples in MBT), we can choose the expected input $I = \{10cent, 20cent, \text{buttonPushed}\}$, but omit unexpected input like inserting $2rupee$, turning the machine off or on, turning the machine upside down and environmental influences like radiation or abrasion. Likewise, we choose the expected output $U = \{10cent, \text{coffee}, \text{tea}\}$, but omit unexpected output like getting $2rupee$, an error screen, sounds and smells the machine makes, and that the machine turns itself off.

If the machine inhibits the inputs $10cent$, respective $20cent$, in some states by a closed coin slot, we add the output $\widetilde{10cent}$, respective $\widetilde{20cent}$, which corresponds to the case that $10cent$, respective $20cent$, is pushed against the coin slot as input, but refused by a closed coin slot.

Notes 8.3. Detecting refusal like $\widetilde{10cent}$ has been researched in **refusal testing** [Phillips, 1987], which allows to detect refusal of any subset of $I \cup U$. We take the more practical approach, since usually the SUT does not show the set of actions it currently cannot perform: We only allow to detect the refusal $\widetilde{i}$ directly after giving the input $i \in I$ (and the refusal $\widetilde{U}$ after a timeout, cf. Subsec. 8.2.3), and consider a refusal as an own output.

In the ioco domain, U (short for the dutch word for output: uitvoer) is the standard name for the set of outputs.

Physical aspects are relevant in software, too, since software needs hardware to run. But even if we exclusively consider software, interface abstractions are required, e.g., omitting as input certain events from the environment, like exceptions or manipulations of memory or files, likewise as output own exceptions and side effects in memory or files.

Since these interface abstractions focus on input and output, their application to MBT is often called behavioral MBT [Anand et al., 2013].

These interface abstractions are lossy and solve the epistemological frame problem [Dennett, 1984; Chow, 2013] of having to take everything into account ("Hamlet's problem viewed from an engineer's perspective" [Fodor, 1987]): by deciding on the scope $I \cup U$ of what is relevant from the physical system, the abstraction turns the informationally unencapsulated physical system into an abstract system that only considers

- the input stimuli in I (helping to solve the qualification side of the frame problem [McCarthy, 1959]);
- the observations in U (helping to solve the qualification and the ramification side of the frame problem [Papadakis et al., 2008]).

Besides these interface abstractions, the input and output can also be made more abstract by the test adapter (cf. Subsec. 8.7.2).

8.1.2. Testing Hypothesis

Interface abstraction is not sufficient to fully formalize the SUT: the rules for the causality between the elements in $I \cup U$ need to be formalized. The specifications, i.e., mathematical structures specifying this, are called **models** in the ioco theory [Tretmans, 2008] (not to be confused with model in the logical sense, e.g., in Def. 3.4 and Def. 4.12). $\boldsymbol{MOD}$ is the set of all models, e.g., $MOD = \mathbb{S}_{LTS}$ or $\mathbb{S}_{Kripke,labeled}$.

The ioco theory represents traces (cf. Def. 3.23 and Def. 8.14), i.e., sequences of elements in $I \cup U$ that occur in the SUT, by words $t \in (I \cup U)^*$ or variants thereof. So with the word $t = (t_j)_{j\in[1,\dots,n]}$, the input or output t_n is an outcome of $(t_j)_{j\in[1,\dots,n-1]}$. The set of all outputs that nondeterministically occur in the SUT as direct consequence of t is $out(\mathrm{SUT}, t) := \{u \in U \mid$ directly after starting the SUT and successively giving the stimuli and observations $(t_j)_{j\in[1,\dots,n]}, u$ is observed$\}$. For $\mathbb{M} \in MOD$ instead of the SUT, $out(\mathbb{M}, t)$ is defined analogously (using $L_I \rightleftharpoons I, L_U \rightleftharpoons U$, cf. Def. 8.14). With this introduction, we can define the **testing hypothesis** in Def. 8.4, which states that accurate abstractions from SUT to MOD are possible.

Definition 8.4. The **testing hypothesis** for given SUT, MOD, I and U is:

$$\forall \mathbb{S} \in SUT\ \exists \mathbb{M} \in MOD : \big(\mathbb{S} \text{ and } \mathbb{M} \text{ accept all inputs in all states and } \forall t \in (I \cup U)^* : out(\mathbb{S}, t) \rightleftharpoons out(\mathbb{M}, t)\big).$$

Notes. This shows that the purpose of our testing hypothesis is to enable formal reasoning about the SUT, and not to restrict SUT for better test case generation (e.g., as in [Gaudel, 1995] via selection hypotheses, which guarantees that a finite set of test cases is sufficient for completeness).

In the standard literature [Tretmans, 2008; Frantzen, 2016], the testing hypothesis does not consider all words in $(I \cup U)^*$, only those represented by $\boldsymbol{TEST}$, the set of test cases generated out of the specification, which is a weaker testing hypothesis iff the test generation algorithm is not sound or not complete (i.e., exhaustive, cf. Sec. 8.8). Since we focus on sound and exhaustive test generation algorithms, we can use our simpler definition that does not require the set $TEST$. Furthermore, the testing hypothesis in the standard literature does not explicitly demand $\mathbb{S} \in SUT$ to accept all inputs in all states. But for sound and exhaustive $TEST$, for each sequence t in a test case, with input at the end, there is also an extension t' to t with an output or quiescence (cf. Subsec. 8.2.3 and Sec. 8.8). So for $out(\mathbb{S}, t') \rightleftharpoons out(\mathbb{M}, t')$ to hold, $\mathbb{S}$ must accept all inputs in all states if $\mathbb{M}$ does. Since this is an important property (cf. Sec. 8.4), this thesis includes the condition explicitly in the testing hypothesis.

For $|out(\mathrm{SUT}, t)| > 1$, the SUT behaves nondeterministically; each $u \in out(\mathbb{S}, t)$ must occur recurrently after finitely many tries (cf. fairness below or [Tretmans, 2008]).

Besides interface abstraction, *MOD* can abstract from further aspects. In the standard ioco theory and in this thesis, we will also abstract from

- continuous or hybrid behavior, so we only use **discrete state spaces** in *MOD*;
- **probability**, but we allow possibilistic models, i.e., use nondeterministic, unquantified choices (see enumeration below);
- **timing values**, so we do not specify and measure precise time (except timeouts, see enumeration below);
- other **quality** aspects like fault-tolerance, performance, usability.

This thesis and most work on *ioco* makes use of the following aspects, so they are not abstracted from. We enumerate all those aspects explicitly:

- infinite systems, so $\mathbb{M} \in MOD$ may have infinitely many states, which is in contrast to FSM-based testing [Lee and Yannakakis, 1996], but much more practical [Huima, 2007];
- nondeterministic systems, so we allow all kinds of nondeterminism (cf. Subsec. 8.2.5). We demand **fairness of the SUT**, i.e., that all nondeterministic choices must occur. Deviant from other literature, we investigate what this means exactly, which leads to three main variants of fairness (described in detail in Subsec. 8.8.3):
  - **fairness**$_{model}$ demands that the SUT exhibits all nondeterministic behaviors of its model $\mathbb{M} \in MOD$ recurrently;
  - **fairness**$_{test}$ demands that $\mathbb{M} \in MOD$ exhibits all its nondeterministic behaviors in each test case;
  - **fairness**$_{spec}$ demands that $\mathbb{M} \in MOD$ exhibits all its nondeterministic behaviors in each *Strace* in $\mathcal{S}$.
- **real-time constraint on the $\rightarrow$ relation**: even though we do not measure time values, we demand that the SUT performs actions before some predetermined **timeout**, i.e., if the SUT accepts an input or makes an output, it does so before the timeout;
- quiescence (cf. Def. 8.8, Def. 8.9 and Def. 8.10), so idleness of the SUT is considered in $\mathbb{M} \in MOD$ (in contrast to FSM-based testing [Lee and Yannakakis, 1996]);
- refusal $\widetilde{i}$ directly after giving the input $i \in I$. So the SUT gives an input refusal as output and continues to run afterwards. In contrast to the standard ioco literature and FSM-based testing, we use this concept to carry out that the SUT accepts all inputs in all states (cf. Subsec. 8.2.6);
- a **reliable reset capability** is required to enable recurrent restarts of the SUT. An **implicit reset** is not reified, i.e., not part of $\mathbb{M} \in MOD$, but abstracted away by the interface abstraction. Contrarily, an **explicit reset** is part of L_I and I. Resets are not mentioned in the standard literature on the ioco theory [Tretmans, 2008; Frantzen, 2016; Brinksma and Tretmans, 2000; ITU-T Study Group 10, 1997], but required. Explicit resets are included in FSM-based testing theory.

8.1.3. Overview

With these abstractions, we can use LTSs, labeled Kripke structures or some variations to formally define all artifacts involved in the ioco theory: *MOD*, system specifications, and test cases (cf. Fig. 8.5 on page 224). Using the same kind of structure for these artifacts, a verification engineer needs to deal with one kind of structure only. Furthermore, their

relations and interactions are simplified: The ioco theory can formally reason about an SUT being conform to a specification, an algorithm can correspondingly generate test cases by unwinding the specification in a specific way, and test execution can simply be formalized by the synchronous product and derive verdicts in accordance to the *ioco* relation.

Roadmap. Sec. 8.2 introduces several variants of and operations on LTSs used in the ioco theory. It contributes new results by thoroughly differentiating various kinds of nondeterminism and internal transitions, as well as consequences thereof for quiescence and livelock detection. Sec. 8.3 defines the set *SPEC* of possible specifications, Sec. 8.4 the set *MOD* of possible models; multiple structures that can all be used for the same ioco theory are given, which has only been done limitedly in other ioco literature [Jard and Jéron, 2005; Volpato and Tretmans, 2013; Tretmans, 2008]. Sec. 8.5 defines the implementation relation *ioco*, furthermore faultable states and traces used later for reduced exhaustive test suites. Sec. 8.6 defines test cases as trees (which is the most suitable structure to apply our heuristics). Sec. 8.7 defines test execution on $\mathbb{M} \in MOD$. Unlike other ioco literature, it additionally makes the full connection to the test execution on the SUT, the testing hypothesis and further abstractions. Sec. 8.8 covers test case generation: besides introducing a nondeterministic algorithm, it makes new contributions by defining a deterministic algorithm that generates a reduced test suite for a given bound, and investigates exhaustiveness thresholds for our fairness constraints. Sec. 8.9 summarizes the chapter, our contributions and possible future work.

8.2. Labeled Transition Systems for ioco

All formalisms used for ioco (cf. Fig. 8.5) are based on labeled transition systems with input and output (and with further elements x_i like internal transitions and quiescence), as defined in Def. 8.5.

Definition 8.5. Let $L \supseteq L_I \dot\cup L_U$, LTS $\mathcal{S} = (S, \rightarrow, L, S^0) \in \mathbb{S}_{LTS}$, $n \in \mathbb{N}_{\geq 0}$ and pairwise different elements $x_1, \ldots, x_n \notin L$. Then:

- $\boldsymbol{L_{x_1 \ldots x_n}} \supseteq L_I \dot\cup L_U \dot\bigcup\limits_{i \in [1,\ldots,n]} \{x_i\}$, with $|L_{x_1 \ldots x_n}| - |L_I \dot\cup L_U \dot\bigcup\limits_{i \in [1,\ldots,n]} \{x_i\}| \in \mathbb{N}_{\geq 0}$;
- $\mathcal{S}$ is called a **labeled transition system with inputs and outputs** (**LTS with I/O**), where $\boldsymbol{L_I}$ specifies input, $\boldsymbol{L_U}$ output;
- $\boldsymbol{\mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n)}$ denotes the set of all LTSs with $L = L_I \dot\cup L_U \dot\bigcup\limits_{i \in [1,\ldots,n]} \{x_i\}$;
- $\boldsymbol{\mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n, +)}$ denotes the set of all LTSs $\mathcal{S}$ with $\exists m \in \mathbb{N}_{>0}$, $\exists y_1, \ldots, y_m : \mathcal{S} \in \mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n, y_1, \ldots, y_m)$;
- $\boldsymbol{\mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n, *)} := \mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n) \dot\cup \mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n, +)$, i.e., with $L = L_{x_1 \ldots x_n}$.

For simplicity and wlog, the set S^0 is always given implicitly (cf. Subsec. 3.4.1) and unreachable states are ignored (i.e., $S = S_{\rightarrow^*}$).

To avoid case distinctions later on, this thesis considers $\mathcal{LTS}(L_I, L_U, x_1, \ldots, x_{n-1}) \subseteq \mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n)$ by identifying $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, x_1, \ldots, x_{n-1})$ with the corresponding $\mathcal{S}' \in \mathcal{LTS}(L_I, L_U, x_1 \ldots, x_n)$ with the label x_n unused.

Often all elements in L_I have the prefix **?**, all elements in L_U the prefix **!**.

8.2.1. Internal Transition

Sometimes it is convenient to use an additional label $\boldsymbol{\tau} \notin L_I \dot{\cup} L_U$ for internal transitions, resulting in Def. 8.6.

Definition 8.6. $\boldsymbol{\tau}$ represents an **internal transition**. Then:

- $\mathcal{S} \in \boldsymbol{\mathcal{LTS}(L_I, L_U, \tau)}$ is called a **labeled transition system with inputs and outputs and internal transition** (**LTS with I/O and** τ).
- $\mathcal{S}$ is **convergent** $:\Leftrightarrow \forall s \in S\, \forall \pi \in paths^{\omega}(\mathcal{S}, s) : trace(\pi) \neq \tau^{\omega}$.

Internal means that the transition is under the control of the SUT, but not observable by the tester. Therefore, testing should be able to abstract from τ. This is done by building the reflexive and transitive τ-closure $\overset{}{\underset{\tau^*}{\Longrightarrow}}$ of $\rightarrow$, as defined in Def. 8.7.

Definition 8.7. Let $\mathcal{S} = (S, \rightarrow, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau, *)$. Then:
$\overset{}{\underset{\boldsymbol{\tau^*}}{\boldsymbol{\Longrightarrow}}} \subseteq S \times (L_\epsilon) \times S$ is the reflexive and transitive $\boldsymbol{\tau}$**-closure** of $\rightarrow$, i.e. $\forall s, s' \in S \forall l \in L$:

- $s \overset{\epsilon}{\underset{\tau^*}{\Longrightarrow}} s' :\Leftrightarrow \exists n \in \mathbb{N}_{>0} : s \overset{\tau^n}{\longrightarrow} s'$ or $s = s'$;
- $s \overset{l}{\underset{\tau^*}{\Longrightarrow}} s' :\Leftrightarrow \exists s_1, s_2 \in S : s \overset{\epsilon}{\underset{\tau^*}{\Longrightarrow}} s_1 \overset{l}{\longrightarrow} s_2 \overset{\epsilon}{\underset{\tau^*}{\Longrightarrow}} s'$.

The τ **abstraction** of $\mathcal{S}$ is $\boldsymbol{\mathcal{S}_{\tau^*}} := (S, \overset{}{\underset{\tau^*}{\Longrightarrow}}, L_\epsilon) \in \boldsymbol{\mathcal{LTS}(L_I, L_U, \epsilon, *)}$.
$\forall \ddot{s} \in 2^S \setminus \{\varnothing\} : \ddot{s}$ is τ**-closed** $:\Leftrightarrow \ddot{s} = dest_{\mathcal{S}_{\tau^*}}(\ddot{s}, \overset{\epsilon}{\underset{\tau^*}{\Longrightarrow}})$.

Now the demands on τ for the testing hypothesis's real-time constraint for $\rightarrow$ can be formulated: $\forall s, s' \in S\, \forall l \in L_\epsilon$: if $s \overset{l}{\underset{\tau^*}{\Longrightarrow}} s'$, then the corresponding action of the SUT must be performed before the predetermined timeout.

Example. Fig. 8.1 on page 192 shows $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ (Subfig. 8.1a), its τ abstraction $\mathcal{S}_{\tau^*}$ (Subfig. 8.1b) and an alternative automaton (Subfig. 8.1e) without τ, but similar behavior (cf. Def. 9.3).

Even though the tester can neither control nor observe internal transitions, τ can be useful in specifications. For instance, having one initial state *init* is sufficient if the other states can be reached by τ. The semantics of τ can be classified along two dimensions:

- whether the SUT has to treat τ **internally like output** ($\boldsymbol{\tau_u}$) or not ($\boldsymbol{\tau_{\not u}}$): The SUT must eventually take τ_u if no other transition is taken. Contrarily, $\tau_{\not u}$ is fully internal, so the SUT may take or ignore such internal transitions as long as this does not contradict $\overset{}{\underset{\tau^*}{\Longrightarrow}}$ (e.g., in Subfig. 8.1a with $a, b \in L_I$, giving input b in s_0 should cause $s_0 \overset{\tau}{\longrightarrow} s_2 \overset{b}{\longrightarrow} s_3$); Subsec. 8.2.3 about quiescence covers this in detail);
- whether τ **may consume time** ($\boldsymbol{\tau_t}$) or not ($\boldsymbol{\tau_{\not t}}$): $\tau_{\not t}$ in the specification corresponds to a **no operation** (**NOP**) in the SUT, i.e., the SUT does not change state in correspondence. τ_t vs. $\tau_{\not t}$ is particularly relevant for livelocks (cf. Subsec. 8.2.2).

These two dimensions lead to the internal transitions $\boldsymbol{\tau_{\not u \not t}}$, $\boldsymbol{\tau_{\not u t}}$, $\boldsymbol{\tau_{ut}}$, and $\boldsymbol{\tau_{u \not t}}$. So $\boldsymbol{\tau_{\not u}}$ means $\tau_{\not u \not t}$ or $\tau_{\not u t}$, analogously for $\boldsymbol{\tau_u}$, $\boldsymbol{\tau_t}$, and $\boldsymbol{\tau_{\not t}}$.

The semantics of τ have an effect on the ioco theory (cf. Subsec. 8.2.2, Subsec. 8.2.3, and Fig. 8.1) and should be chosen according to the application of τ:

- to easily and clearly **model nondeterminism of the LTS** (cf. Subsec. 8.2.5) and **structure the LTS**, in which case $\tau_{\not t}$ is suitable. The strongest variant of structuring is using **equality** between two states s, s', i.e., the states and their incoming and outgoing transitions can be merged together in $\mathcal{S}$. In this case, a deliberate τ-cycle (cf. Subsec. 8.2.2) $s \xrightarrow{\tau_{\not t\not t}} s' \xrightarrow{\tau_{\not t\not t}} s$ can be used;
- to naturally **hide an action** for abstraction: The straight forward way to hide an action is to replace it with $\tau_{\not t t}$ if it is an input, and with τ_{ut} if it is an output. If the output should be completely removed from $\mathcal{S}$, e.g., because neither the output nor any information about internal transitions in the SUT is present (any longer), $\tau_{\not t\not t}$ should be used; for **process algebraic specifications**, many actions are meant from the beginning only for synchronization and later action hiding, in which case $\tau_{\not t}$ is most suitable;
- for **unobservable communication** if the tester knows that **hidden I/O between components** must take place. Sometimes, the SUT is not fully black-box and the tester has this insight, e.g., during **compositional** or **integration testing** that is not fully black-box: the tester knows how the SUT is composed of components, and their respective interfaces. Thus the tester knows when a synchronous product $C_1||C_2$ hides I/O, caused by an output of C_1 being consumed as input of C_2 without revealing it to its environment. Therefore, τ_{ut} should be used;
- if no information on the application of τ is given, we do not know whether it consumes time, so τ_t should be used.

As default, this thesis uses $\tau = \tau_{ut}$ (i.e., $\boldsymbol{\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau_{ut}, *)}$) since this is a common case and leads to the usual definition of quiescence (cf. Def. 8.8). If another semantics of internal transitions is used, we make this explicit by replacing τ in $\mathcal{S}$, e.g., use $\boldsymbol{\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau_{\not t}, *)}$.

8.2.2. Livelocks

Since τ corresponds to internal actions of the SUT, they do not contribute to the SUT's observable behavior, i.e., its progress. Consequently, they can cause livelocks in the SUT: If infinitely many τ of a trace τ^ω in the system specification $\mathcal{S}$ are τ_t, i.e., not NOPs in the SUT, the trace corresponds to a livelock.

The only traces τ^ω that an on-the-fly traversal algorithm over $\mathcal{S}$ can detect are **τ-cycle**, i.e., cycles in $\mathcal{S}$ that only consists of τ transitions. For other τ^ω traces, black-box testing cannot differentiate between quiescence (cf. Subsec. 8.2.3), deadlocks and livelocks. Luckily, if $\mathcal{S}$ only has finitely many τ_t, a livelock τ^ω stems from a τ-cycle. This leads to the following three solutions:

- use the precondition that $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau, *)$ is convergent, which is what most MBT tools do, e.g., the original (J)TorX (cf. Subsec. 10.3.2 and Subsec. 10.3.3);
- only allow $\tau_{\not t}$. The test case generation algorithm must still check for τ-cycles to avoid looping endlessly, but $\tau_{\not t}$-cycles are not livelocks and thus acceptable. This is what JTorX currently does by default. STG (cf. Subsec. 10.3.4), an extension of the tool TGV, performs τ abstraction only for internal transitions that have non-internal sibling transitions. This handles some but not all τ-cycles; e.g., for $s_1 \xleftarrow{\tau} s_2$ with both s_1 and s_2 having non-internal sibling transitions, τ abstraction does not terminate;

- allow infinitely many $\tau_{\not t}$ in $\mathcal{S}$, but only finitely many τ_t; check for τ-cycles to resolve endless loops in test case generation algorithms (cf. Subsec. 8.8 and Chapter 11). If the τ-cycle contains τ_t transitions, give a livelock warning to the user, since livelocks are faults and also do not meet the testing hypothesis's real-time constraint for $\rightarrow$. This approach is a generalization of what the tool TGV does (cf. Subsec. 10.3.1): guarantee only finitely many τ_t transitions by demanding $\mathcal{S}$ to have only finitely many states. TGV returns quiescence instead of a livelock warning (cf. Note 8.13 and Subsec. 8.2.3 below). TGV uses Tarjan's DFS to detect τ-cycles. This requires the decomposition of $\mathcal{S}$ into SCCs (cf. Subsec. 5.3.1) and additional memory. Chapter 6 introduces other livelock detection algorithms, which are compatible with optimizations, especially partial order reduction. DFS_{FIFO} is particularly apt because testing often requires strong on-the-flyness (cf. Subsec. 6.8.6) and short τ-traces (cf. Subsec. 6.4.3) are better understandable. Compared to the NDFS, DFS_{FIFO} is also simpler since exactly the τ_t transitions can be set as non-progress transitions for DFS_{FIFO}. Though DFS_{FIFO} does not preserve the full lasso starting from *init*, this is no drawback since usually the paths with internal transitions are not needed later on, only a set of states on the paths (cf. Sec. 8.3 and Sec. 8.8).

Note. Note that livelocks are detected in $\mathcal{S}$ during its traversal; they cannot be detected in the SUT during its executing, since liveness properties cannot be detected by dynamic testing (cf. Subsec. 10.2.1).

8.2.3. Quiescence

The testing hypothesis for ioco does not abstract from idleness and the real-time constraint for $\rightarrow$. So when the tester does not give an input but waits for an output, he either observes an output in L_U or **quiescence** (also called **suspension**), i.e., that the SUT gives no output until the timeout. Therefore, quiescence can be considered as a special kind of output (or as the refusal of all output, $\widetilde{L_U}$, cf. Note 8.3). Def. 8.8 reifies quiescence in $\mathcal{LTS}(L_I, L_U, \tau_u, *)$, Def. 8.9 in $\mathcal{LTS}(L_I, L_U, \tau_{\not t}, *)$, Def. 8.10 for mixed τ semantics in $\mathcal{LTS}(L_I, L_U, \tau_u, \tau_{\not t}, *)$.

Definition 8.8. Let $\tau = \tau_u$, $\mathcal{S} = (S, \rightarrow, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau, *)$, $\delta \notin L_\tau$ and $s, s' \in S$.

Then:

- s is **quiescent** $:\Leftrightarrow \forall u \in L_U \dot\cup \{\tau\} : s \overset{u}{\not\longrightarrow}$. This is written $\boldsymbol{\delta_u(s)}$ or $\boldsymbol{\delta(s)}$ if it is clear that$\tau = \tau_u$;
- $\boldsymbol{\mathcal{S}_\delta} := (S, \overset{}{\underset{\delta}{\rightarrow}}, L_{\tau\delta}) \in \boldsymbol{\mathcal{LTS}(L_I, L_U, \tau, \delta, *)}$
  is $\mathcal{S}$'s **suspension automaton**, where
  $\overset{}{\underset{\delta}{\rightarrow}} :=\rightarrow \dot\cup \{(s, \delta, s) \mid s \in S \text{ with } \delta(s)\}$ reifies quiescence in $\mathcal{S}$;
- $\boldsymbol{\mathcal{S}_{\tau^*\delta}} := (S, \Rightarrow, L_{\epsilon\delta}) \in \boldsymbol{\mathcal{LTS}(L_I, L_U, \epsilon, \delta, *)}$
  is $\mathcal{S}$'s **suspension automaton after τ abstraction**,
  where $\Rightarrow$ is also written $\overset{}{\underset{\tau^*\delta}{\rightarrow}}$ and
  $\Rightarrow :=\overset{}{\underset{\tau^*}{\rightarrow}} \dot\cup \{(s, \delta, s) \mid s \in S \text{ with } \delta(s)\}$ reifies quiescence in $\mathcal{S}_{\tau^*}$;
- $\boldsymbol{\delta_{\circlearrowleft}}$:= δ that is used only in self-loops (s, δ, s);
- $\boldsymbol{\delta_{\curvearrowright}}$:= δ that is not used only in self-loops, i.e., also (s, δ, s') for some $s \neq s'$;
- $\boldsymbol{\mathcal{S}_{\delta\tau^*}} := (S, \Rightarrow, L_{\delta\epsilon}) \in \boldsymbol{\mathcal{LTS}(L_I, L_U, \delta, \epsilon, *)}$
  is $\mathcal{S}$'s τ **abstraction after reifying quiescence**,
  where $\Rightarrow$ is also written $\overset{}{\underset{\delta\tau^*}{\rightarrow}}$ and $\delta_{\curvearrowright}$ is possible.

Def. 8.8 of $\delta(s)$ is the default for this thesis and the definition found in the ioco literature [Tretmans, 2008]. It forbids outgoing τ_u since τ_u performs hidden output. But for $\tau = \tau_{\not u}$, τ transitions are fully internal and arbitrary as long as they do not contradict $\overset{}{\underset{\tau^*}{\rightarrow}}$ (cf. Subsec. 8.2.1). Therefore, $\delta(s)$ must also allow arbitrary $\tau_{\not u}$ as long as they do not contradict $\overset{}{\underset{\tau^*}{\rightarrow}}$, leading to Def. 8.9. So when the tester observes quiescence, outgoing τ_u must have been taken, whereas outgoing $\tau_{\not u}$ may have been taken.

Definition 8.9. Let $\tau = \tau_{\not u}, \mathcal{S} = (S, \rightarrow, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau, *)$, $\delta \notin L_\tau$ and $s \in S$. Then:

- s is **quiescent** $:\Leftrightarrow \forall u \in L_U : s \overset{u}{\not\underset{\tau^*}{\rightarrow}}$. This is written $\boldsymbol{\delta_{\not u}(s)}$ or $\boldsymbol{\delta(s)}$ if it is clear that $\tau = \tau_{\not u}$.

In case both τ semantics τ_u and $\tau_{\not u}$ are mixed in $\mathcal{S}$, quiescence must also consider their combinations, i.e., forbid quiescence if τ_u is reachable via $\tau_{\not u}^*$, leading to Def. 8.10.

Definition 8.10. Let $\mathcal{S} = (S, \rightarrow, L_{\tau_{\not u}\tau_u}) \in \mathcal{LTS}(L_I, L_U, \tau_{\not u}, \tau_u, *)$, $\delta \notin L_{\tau_{\not u}\tau_u}$ and $s \in S$. Then:

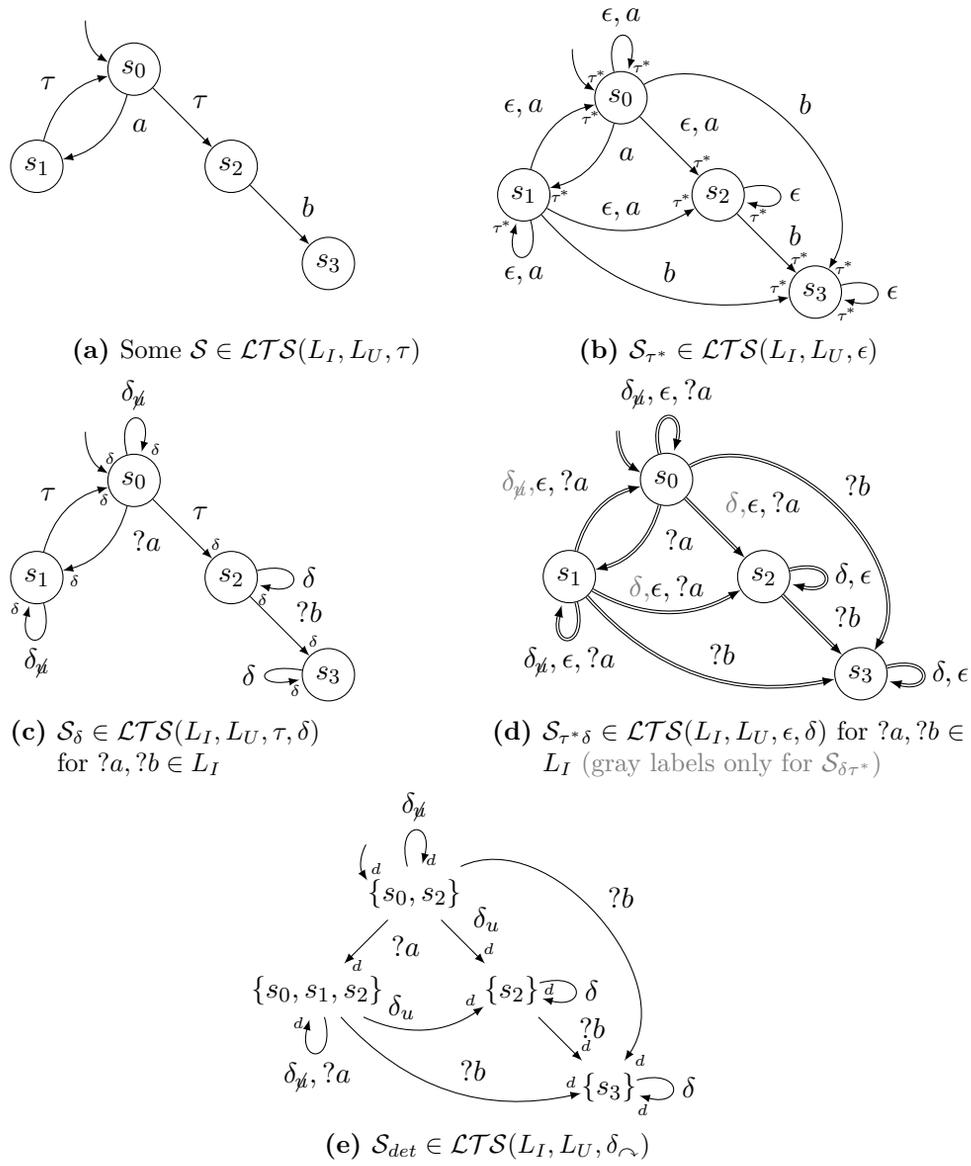
- s is **quiescent** $:\Leftrightarrow \forall u \in L_U \dot\cup \{\tau_u\} : s \overset{u}{\not\underset{\tau_{\not u}^*}{\rightarrow}}$. This is written $\boldsymbol{\delta_{\not u u}(s)}$ or $\boldsymbol{\delta(s)}$ if it is clear that τ_u and $\tau_{\not u}$ may occur .

Example 8.11. Subfig. 8.1c reifies quiescence for $\mathcal{S}$ of Subfig. 8.1a and $?a, ?b \in L_I$. $\delta_{\not u}$ (respectively δ_u) in a state s means s is quiescent if all $\tau = \tau_{\not u}$ (respectively all $\tau = \tau_u$). δ means that $\delta_u, \delta_{\not u}$ and $\delta_{\not u u}$ hold. For τ_u, if we initially observe δ, then $?a$ cannot be observed afterward since δ_u implies that $s_0 \xrightarrow{\tau_u} s_2$ was taken. For $\tau_{\not u}$, initially observing $\delta_{\not u}$ and then $?a$ is possible because of $\delta_{\not u}(s_0)$.

The suspension automaton after τ abstraction $\mathcal{S}_{\tau^*\delta}$ of Subfig. 8.1d reifies quiescence (δ_u or $\delta_{\not u}$ or $\delta_{\not u u}$) for the τ abstraction $\mathcal{S}_{\tau^*}$ in Subfig. 8.1b: $\delta(s)$ can be checked easily

by replacing τ with ϵ: For $\tau = \tau_{\not\iota}, \delta_{\not\iota}(s)$ iff $enabled_{\mathcal{S}_{\tau^*}}(s) \cap L_U = \varnothing$; for $\tau = \tau_u, \delta_u(s)$ iff $enabled_{\mathcal{S}_{\tau^*}}(s) \cap L_U = \varnothing$ and $dest_{\mathcal{S}_{\tau^*}}(s, \xrightarrow{\epsilon}_{\tau^*}) = \{s\}$. So we only get self-loops $\delta_{\circlearrowleft}$. For $\mathcal{S}_{\delta\tau^*}$, the τ abstraction is made after reifying quiescence, i.e., from Subfig. 8.1c. So the τ abstraction is also made for $\delta_{\circlearrowleft}$, resulting in the additional gray quiescent transitions that are no longer self-loops but $\delta_{\frown}$.

If the semantics were mixed by having $s_1 \xrightarrow{\tau_{\not\iota}} s_0 \xrightarrow{\tau_u} s_2$, then neither $\delta_{\not\iota u}(s_0)$ nor $\delta_{\not\iota u}(s_1)$.



(a) Some $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$

(b) $\mathcal{S}_{\tau^*} \in \mathcal{LTS}(L_I, L_U, \epsilon)$

(c) $\mathcal{S}_\delta \in \mathcal{LTS}(L_I, L_U, \tau, \delta)$ for $?a, ?b \in L_I$

(d) $\mathcal{S}_{\tau^*\delta} \in \mathcal{LTS}(L_I, L_U, \epsilon, \delta)$ for $?a, ?b \in L_I$ (gray labels only for $\mathcal{S}_{\delta\tau^*}$)

(e) $\mathcal{S}_{det} \in \mathcal{LTS}(L_I, L_U, \delta_{\frown})$

Figure 8.1.: Exemplary $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ and its transformations ($\delta_{\not\iota}$ for $\tau = \tau_{\not\iota}$, δ_u for $\tau = \tau_u$, δ for both)

Lemma 8.12 shows the relationship between $\delta_u(s)$ and $\delta_{\not\mu}(s)$ when the kind of τ is left open.

Lemma 8.12. *Let* $\mathcal{S} = (S, \to, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau, *)$ *and* $s \in S$.

Then $\delta_u(s) \implies \delta_{\not\mu}(s)$, *but* $\delta_u(s) \not\impliedby \delta_{\not\mu}(s)$ *is possible.*

Proof. $\delta_u(s) \Longrightarrow \tau \notin \mathit{enabled}_{\mathcal{S}}(s)$ (for $\tau = \tau_u$) $\Longrightarrow \mathit{enabled}_{\mathcal{S}}(s) \subseteq L_I \Longrightarrow \forall u \in L_U : s \overset{u}{\not\Rightarrow}$ (for $\tau = \tau_{\not\mu}$) $\implies \delta_{\not\mu}(s)$. Subfig. 8.1c with $s = s_0$ is an example for $\delta_u(s) \not\impliedby \delta_{\not\mu}(s)$. □

Note 8.13. Some tools, e.g., TGV and JTorX, do not differentiate between quiescence and livelocks (cf. Subsec. 8.2.2): Instead of detecting livelock and warning the user, they detect a timeout and return quiescence. This means that $\delta_{\circlearrowleft}$ is added to all states that are on a τ-cycle [Jard and Jéron, 2005; Belinfante, 2014; Stokkink et al., 2013]. This chapter differs since livelocks often indicate some erroneous behavior that the test engineer should be informed about, and quiescence should indicate idleness, and output should not follow δ (if τ-cycles are not excluded from the real-time constraint on $\to$). To fix that output should not follow δ, [Stokkink et al., 2013] recently changed the semantics of the LTS by modifying it: Instead of adding $\delta_{\circlearrowleft}$ to each state s that is on a τ-cycle, a transition $\delta_{\curvearrowright}$ is added, pointing to a copy s' of s, which is added, too. s' inherits outgoing input and quiescence from s, resulting in $\mathit{enabled}_{\mathcal{S}_\delta}(s') = \{\delta_{\circlearrowleft}\} \cup (\mathit{enabled}_{\mathcal{S}_\delta}(s) \cap L_I)$. Belinfante adapted this fix in JTorX, but only optionally since he is "not yet sure about its usability" [Belinfante, 2014]. Furthermore, he pointed out the problem that τ transitions leaving a τ-cycle are not considered [Stokkink et al., 2013, Def.4.1]: if δ is observed in s, only the inputs may follow that directly leave the τ-cycles which s is on. To allow exiting a τ-cycle not only through such a direct input, but also through some $\tau^* \cdot i$, the LTS has to be modified further by also adding all $i \in L_I$ with $s \overset{i}{\underset{\tau^*}{\Rightarrow}}$ to the copy s' as outgoing transition.

8.2.4. Operations

Most of the following operations on LTSs can be formulated already with the operations introduced in Sec. 3.4. To be able to formulate even more concisely and use the standard notation [Tretmans, 2008; Frantzen, 2016], we give Def. 8.14.

Definition 8.14. Let $\mathcal{S} = (S, \rightarrow, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau, *)$ and $s \in S$. Then:

- $\boldsymbol{traces_{\mathcal{S}_{\tau^*}}(s)} := traces^{fin}(\mathcal{S}_{\tau^*}, s)$;
- $\boldsymbol{Straces_{\mathcal{S}_{\tau^*\delta}}(s)} := traces^{fin}(\mathcal{S}_{\tau^*\delta}, s)$ (called **suspension traces**);
- $\boldsymbol{after_{\mathcal{S}_{\tau^*\delta}}} : S \times L_\delta^* \rightarrow 2^S, (s, \sigma) \mapsto dest_{\mathcal{S}_{\tau^*\delta}}(s, \overset{\sigma}{\Longrightarrow}^*)$
  (often written infix as $s\ after_{\mathcal{S}_{\tau^*\delta}}\ \sigma$);
- $\boldsymbol{in_{\mathcal{S}_{\tau^*}}(s)} := enabled_{\mathcal{S}_{\tau^*}}(s) \cap L_I$;
- $\boldsymbol{out_{\mathcal{S}_{\tau^*\delta}}(s)} := enabled_{\mathcal{S}_{\tau^*\delta}}(s) \cap (L_U \dot\cup \{\delta\})$;
- $\boldsymbol{branchout_{\mathcal{S}}} = \underset{s \in S}{supremum}(|out_{\mathcal{S}_{\tau^*\delta}}(s)|)$;
- $\boldsymbol{\approx_{Straces}} \subseteq \mathcal{LTS}(L_I, L_U, *)^2$(called **Strace equivalence**), with
  $\mathcal{S}' \approx_{Straces} \mathcal{S}'' :\Leftrightarrow Straces_{\mathcal{S}'_{\tau^*\delta}}(init_{\mathcal{S}'}) = Straces_{\mathcal{S}''_{\tau^*\delta}}(init_{\mathcal{S}''})$;
- $\boldsymbol{\approx_{Straces}} \subseteq (2^{\mathcal{LTS}(L_I, L_U, *)})^2$ lifts Strace equivalence to sets, with
  $\mathcal{L} \approx_{Straces} \mathcal{L}' :\Leftrightarrow \forall \mathcal{S} \in \mathcal{L} \exists \mathcal{S}' \in \mathcal{L}' : \mathcal{S} \approx_{Straces} \mathcal{S}'$, and vice versa.

If the τ abstraction is made after reifying quiescence, the index $\mathcal{S}_{\delta\tau^*}$ is used instead. We sometimes drop the index if it is clear from the context, e.g., from a state given as parameter.

Lemma 8.15 shows the relationship between various *Straces*.

Lemma 8.15. *Let $\mathcal{S} = (S, \rightarrow, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau, *)$ and $s \in S$. Then*
$Straces_{\mathcal{S}_{\delta_u \tau_u^*}}(s) = Straces_{\mathcal{S}_{\tau_u^* \delta_u}}(s) \subsetneq Straces_{\mathcal{S}_{\tau_{\not u}^* \delta_{\not u}}}(s) = Straces_{\mathcal{S}_{\delta_{\not u} \tau_{\not u}^*}}(s)$.

Proof. Since both $s_1 \xrightarrow{\delta_{\curvearrowright}} s_2$ and $s_1 \xrightarrow[\tau^*]{\epsilon} s_1' \xrightarrow{\delta_{\circlearrowleft}} s_1' \xrightarrow[\tau^*]{\epsilon} s_2$ result in $s_1 \Rightarrow s_2$, the equalities hold. The subset relation is a direct consequence of Lemma 8.12, with Subfig. 8.1d giving an example. □

8.2.5. Nondeterminism

We use **nondeterminism** for desired or forced multiple choices within one situation. We do not determine whether nondeterminism is caused by epistemological restrictions (cf. Subsec. 8.1.1), or by indeterminism of nature itself, or by both [Bohm, 1971].

Nondeterminism can be differentiate by two aspects:

- whether the nondeterministic choices at a choice point are **controllable by the tester** (sometimes called **external nondeterminism**), i.e., related to input, or **uncontrollable by the tester** (sometimes called **internal nondeterminism**), i.e., related to output;
- whether the uncontrollable nondeterministic choice is **immediately observable** by the tester (sometimes called **external nondeterminism**) or known to the tester only later on, if at all (sometimes called **internal nondeterminism**).

On the whole, there are three different types of nondeterminism:

- nondeterminism that is controllable by the tester comprises two cases: Firstly, the nondeterministic choice point whether the tester tries to give a stimuli to the SUT or only waits to observe the SUT; secondly, if the tester gives a stimuli, he can also nondeterministically choose amongst the inputs $in(s)$ if $|in(s)| \gneq 1$. These

nondeterministic choices are carried over into the test case generation algorithms in Sec. 8.8;

- nondeterministic choices uncontrollable by the tester, but immediately observable, occur iff $|out(s)| \gneq 1$. Thus we call it **nondeterminism on output**;
- nondeterministic choices uncontrollable by the tester and not immediately observables stem from **nondeterminism of the LTS**, i.e., iff $\exists \sigma \in traces_{\mathcal{S}_{\tau^*}}(init) : |init\ after\ \sigma| \gneq 1$. There are two different causes for this: Firstly, if $\exists l \in L_\tau : \xrightarrow{l}$ is not right-unique. This is called **nondeterministic branching** and wlog subsumes nondeterministic initialization, i.e., if $|S^0| \gneq 1$. Secondly, if $\mathcal{S}$ has internal transitions, since the SUT might or might not take them (with $\tau_{\not u}$ being more nondeterministic than τ_u since the SUT may, but need not, take $\tau_{\not u}$ while the tester waits). In short, we have nondeterminism of the LTS iff $\exists l \in L_\epsilon : \xrightarrow{l}_{\tau^*}$ is not right-unique.

In practice, controllable nondeterminism occurs in almost every system, since usually the user can make some choices. Systems with uncontrollable nondeterminism are often called **nondeterministic systems**. Uncontrollable nondeterminism also occurs in almost all complex systems, since they usually build upon lower layers, e.g., the operating system or network [Fraser et al., 2009; ETSI, European Telecommunications Standards Institute, 2011]. Since these lower layer are not under the tester's control (e.g., containing asynchronous communication, scheduling, parallel interleavings, possibly race conditions) or too complex to model and monitor, they are abstracted from: several behaviors are comprised into one abstract behavior (such as a nondeterministic choice), where we need not care about which concrete behavior actually occurred (e.g., which of the many interleaving (cf. Table 6.1) occurred or what concrete data is present in the underlying database). For instance, $out(s)$ might contain one output from the happy path and several exceptional cases (e.g., when the underlying database contains null). So this abstraction can be forced upon the tester, but is often also desired to simplify testing: specifications are simpler to understand, sometimes test case generation becomes easier and test case execution more efficient. Besides reducing complexity and covering unpredictability, nondeterminism can also postpone implementation decisions, which is particularly useful for iterative software development (cf. Sec. 14.2).

Depending on the type of nondeterminism, different methods are used for **resolving nondeterminism**:

Controllable nondeterminism is resolved by the tester making a choice, or considering multiple choices (cf. Subsec. 8.8.3 and Subsec. 10.2.5).

Nondeterminism on output is resolved the moment the SUT picks one output of the nondeterministic choices, i.e., during test case execution. Until then, all choices should be respected for exhaustive testing, leading to test cases being test trees (cf. Sec. 8.6) as opposed to sequences (called linear test cases). If nondeterminism on output occurs often and with many choices, test trees can become very broad. Then offline MBT needs to create huge test suites from which most parts have to be discarded during test case execution because of the SUT made other nondeterministic choices (cf. Subsec. 10.2.5).

Uncontrollable and not immediately observable nondeterminism occur when a trace does not fully determine the final state. Thus multiple states are possible, which must all be included in subsequent computations. Therefore, a suspension trace does not lead

to a single state, but to a set of potential states, one of which really represents the SUT's state. This nonempty set of states $\ddot{s}$ is called **superstate** (or **meta-state** [Jard and Jéron, 2005]), and **real superstate** if $|\ddot{s}| > 1$. So this kind of nondeterminism can be resolved by keeping track of superstates, where a transition between superstates, i.e., $\ddot{s} \xrightarrow{l} \ddot{s}'$ for $l \in L_\delta$, is the result of all potential transitions in $\mathcal{S}_\delta : \ddot{s}' = \bigcup_{s \in \ddot{s}} dest_{\mathcal{S}}(s, \xrightarrow{l}_\delta)$. This eliminates nondeterministic branching, but a δ transition from superstate $\ddot{s}$ can now occur in combination with other output. Furthermore, δ is no longer a self-loops $\delta_{\circlearrowleft}$ iff $\{s \in \ddot{s} | \delta(s)\} \neq \ddot{s}$, so we use a transition $\delta_{\curvearrowright}$ instead. A subsequent τ-closure for each $\ddot{s} \xrightarrow{l} \ddot{s}'$ also eliminates internal transitions. The result is $\mathcal{S}_{det}$, given in Def. 8.16.

Definition 8.16. Let $\mathcal{S} = (S, \rightarrow, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau)$. Then:

- $\boldsymbol{\mathcal{S}_{det}} := (S_{det}, \xrightarrow[d]{}, L_{\delta_{\curvearrowright}}) \in \mathcal{LTS}(L_I, L_U, \delta_{\curvearrowright})$ is the **determinized suspension automaton** of $\mathcal{S}$, with
- $\boldsymbol{S_{det}} := 2^S_{\xrightarrow[d]{}^*}$;
- $\boldsymbol{init_{\mathcal{S}_{det}}} := init_{\mathcal{S}}\ after_{\mathcal{S}_{\tau^*}}\ \epsilon$;
- $\boldsymbol{\xrightarrow[d]{}} := \{(\ddot{s}, l, \bigcup_{s \in \ddot{s}} dest_{\mathcal{S}_{\tau^*\delta}}(s, \stackrel{l}{\Longrightarrow})) \mid \ddot{s} \in S_{det}, l \in enabled_{\mathcal{S}_{\tau^*\delta}}(\ddot{s})\}$;
- transforming $\mathcal{S}$ into $\mathcal{S}_{det}$ is called **determinization** of $\mathcal{S}$.

Notes 8.17. Determinization of $\mathcal{S}$ resolves all nondeterminism of the LTS in $\mathcal{S}$.

In some literature [Tretmans, 2008], suspension automaton refers to $\mathcal{S}_{det}$.

Nondeterminism of the LTS can fully abstract away certain differentiating behavior in a specific state $s \in S$, resulting in counter-intuitive specifications: For instance, let s differentiate odd from even via $Straces_{\mathcal{S}_{\delta_u \tau_u^*}}(s) = \{?i?i\}^* \cdot ?end!even \dot{\cup} \{?i?i\}^* \cdot ?i?end!odd$, but s only occurs in superstates that also contain a state that allows all behavior (e.g., state χ). Then $\mathcal{S}_{det}$ cannot differentiate odd from even for all nondeterministic resolutions.

Example. Subfig. 8.1e shows the determinized suspension automaton $\mathcal{S}_{det}$ for $\mathcal{S}$ of Subfig. 8.1a and $?a, ?b \in L_I$.

Complexity analyses for the time and space requirements of determinizing $\mathcal{S}$ are very rough, i.e., the typical complexity measures in Landau notation are only moderately meaningful (cf. Subsec. 10.2.5). Thus they are derived only shortly:

The worst case time complexity per determinization step, i.e., per considered superstate $\ddot{s} \in S_{det}$, depends on:

- $|S_{\rightarrow^*}|$ to traverse through all the states $s \in \ddot{s}$ and to include all the states $s \in \ddot{s}\ after_{\mathcal{S}_{\tau^*\delta}}\ a$ (where $a \in enabled_{\mathcal{S}_{\tau^*\delta}}(\ddot{s})$), i.e., also those states reachable via subsequent τ transitions;
- $branch_{\mathcal{S}_{\rightarrow^*}}$ to consider all outgoing transitions of each such state s.

This results in a worst case time complexity in $O(|S_{\rightarrow^*}| \cdot branch_{\mathcal{S}_{\rightarrow^*}})$ per step.

Since $S_{det} \subseteq 2^S$, S_{det} can have up to $2^{|S_{\rightarrow^*}|}$ many states, the **overall worst case time complexity of this determinization** of $\mathcal{S}$ is in $O(2^{|S_{\rightarrow^*}|} \cdot |S_{\rightarrow^*}| \cdot branch_{\mathcal{S}_{\rightarrow^*}})$.

The worst case space complexity of determinization of $\mathcal{S}$ depends on:

- $|S_{det}|$, the possible number of superstates to store;

- $|S_{\to^*}|$, the possible number of states per superstate;
- $branch_{\mathcal{S}_{det}} \leq max(|L_\delta|, |S_{det}|)$, the possible number of outgoing transitions in $\mathcal{S}_{det}$ per superstate.

This results in an **overall worst case space complexity of this determinization** of $\mathcal{S}$ in $O(2^{|S_{\to^*}|} \cdot (|S_{\to^*}| + branch_{\mathcal{S}_{det}}))$.

Often, $branch_{\mathcal{S}_{\to^*}}$ is small (cf. Subsec. 11.3.3) and can be ignored. Then the overall worst case time complexity of determinization of $\mathcal{S}$ is in $O(2^{|S_{\to^*}|} \cdot |S_{\to^*}|)$.

Notes. $branch_{\mathcal{S}_{det}}$ can become large in spite of a small $branch_{\mathcal{S}_{\to^*}}$; Fig. B.1 on page 386 gives an example if only the maximal available `licenseID` were allowed as parameter for `?removeLicenseInput`.

Determinization requires $ld(branch_{\mathcal{S}_{\to^*}})$ of memory to be able to iterate for the current state s over its possible outgoing transitions, but this is negligible in Landau notation since $branch_{\mathcal{S}_{\to^*}} \in O(branch_{\mathcal{S}_{det}} \cdot |S_{\to^*}|)$. In summary, focusing on the strongest growth, the overall worst case time and space complexities of determinization of $\mathcal{S}$ are exponential in $|S_{\to^*}|$.

All these complexities are rough because the structure of $\mathcal{S}_{det}$ is usually much simpler than the worst case, often with only a few large superstates, so determinization requires much less runtime and memory.

This determinization corresponds to the popular Rabin-Scott powerset construction [Rabin and Scott, 1959]. Though it does not yield minimal LTSs, it can be used as an on-the-fly determinization of the LTS; on-the-fly determinization is a requirement to perform on-the-fly MBT. Furthermore, only reachable states are considered (so $S_{\to^*}$ was used in the complexity formulas instead of S). For an imperative Rabin-Scott powerset construction, the LTS $\mathcal{S}$ must have $|init_{\mathcal{S}}| < \omega$ and be **image finite**: $\forall s \in S \, \forall \sigma \in L^* : |s \; after \; \sigma| \in \mathbb{N}$.

τ abstractions are mainly used for formal reasoning about traces. Contrarily, the Rabin-Scott powerset construction is mainly used for implementations. The Lemmas 8.18 and 8.21 show the relationship.

Lemma 8.18. *Let $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau, *)$. Then*
$Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}) = Straces_{\mathcal{S}_{det}}(init_{\mathcal{S}_{det}})$.

Proof. The proof uses induction over the length of the trace σ.

For the base case, $\sigma = \epsilon$, so $\sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}) \cap Straces_{\mathcal{S}_{det}}(init_{\mathcal{S}_{det}})$.

For the induction step from n to $n+1$, let $\sigma := \sigma' \cdot l$ with $l \in L_{\delta_\frown}$ and $\sigma' \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}) \cup Straces_{\mathcal{S}_{det}}(init_{\mathcal{S}_{det}})$. Because of the induction hypothesis, $\sigma' \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}) \cap Straces_{\mathcal{S}_{det}}(init_{\mathcal{S}_{det}})$. Let $\ddot{s} := init_{\mathcal{S}} \; after \; \sigma$. Therefore, $\sigma \cdot l \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}) \Leftrightarrow \exists s \in \ddot{s} : l \in enabled_{\mathcal{S}_{\tau^*\delta}}(s) \Leftrightarrow \ddot{s} \xrightarrow[d]{l} \Leftrightarrow \sigma \cdot l \in Straces_{\mathcal{S}_{det}}(init_{\mathcal{S}_{det}})$. □

We say a deterministic $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \delta_\frown)$ **reifies quiescence** if $\exists \mathcal{S}' \in \mathcal{LTS}(L_I, L_U, \tau) : \mathcal{S} = \mathcal{S}'_{det}$. Such $\mathcal{S}'$ need not exist if $\mathcal{S}$ uses δ as arbitrary label, for instance $\mathcal{S}$ with $S = \{init_{\mathcal{S}}\}$ and no transition. Lemma 8.19 gives a criterion, Cor. 8.20 the direct consequence for Strace equivalence. Thereafter, Lemma 8.21 shows that $\delta_\frown$ is required.

Lemma 8.19. *Let* $\mathcal{S} = (S, \xrightarrow[d]{}, L) \in \mathcal{LTS}(L_I, L_U, \delta_{\circlearrowleft}, \delta_{\curvearrowright})$ *be deterministic with:* $\forall \ddot{s}, \ddot{s}' \in S : \ddot{s} \xrightarrow[d]{\delta_{\circlearrowleft}} \ddot{s}' \Rightarrow \ddot{s} = \ddot{s}'$ *and* $\ddot{s} \xrightarrow[d]{\delta_{\curvearrowright}} \ddot{s}' \Rightarrow \ddot{s} \neq \ddot{s}'$. *Then* $\mathcal{S}$ *reifies quiescence iff*

$$\bullet\ \forall \ddot{s} \in S : out_{\mathcal{S}}(\ddot{s}) \subseteq L_U \dot\cup \{\delta_{\curvearrowright}\} \text{ or } out_{\mathcal{S}}(\ddot{s}) = \{\delta_{\circlearrowleft}\} \quad (1)$$

$$\bullet \text{ and } \forall \ddot{s}, \ddot{s}' \in S : \ddot{s} \xrightarrow[d]{\delta_{\curvearrowright}} \ddot{s}' \Rightarrow$$

$$(out_{\mathcal{S}}(\ddot{s}') = \{\delta_{\circlearrowleft}\} \quad (2)$$

$$\text{and } \exists L' \subseteq L_U : L' \neq \varnothing \text{ and } out_{\mathcal{S}}(\ddot{s}) = L' \dot\cup \{\delta_{\curvearrowright}\} \quad (3)$$

$$\text{and } \forall i \in L_I\ \forall \ddot{s}'' \in S : (\ddot{s}' \xrightarrow[d]{i} \ddot{s}'' \Rightarrow \ddot{s} \xrightarrow[d]{i} \ddot{s}'') \quad (4)\ )$$

Proof. Let $\ddot{s}, \ddot{s}' \in S$ and $\mathcal{S}$ reify quiescence with $\mathcal{S}' = (S', \rightarrow, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau)$ so that $\mathcal{S}'_{det} = \mathcal{S}$. If $out_{\mathcal{S}}(\ddot{s}) \not\subseteq L_U \dot\cup \{\delta_{\curvearrowright}\}$, then we have in $\mathcal{S}'$: $\forall s \in \ddot{s} : \delta(s)$, i.e., (1) holds since $\ddot{s} \neq \varnothing$. If $\ddot{s} \xrightarrow[d]{\delta_{\curvearrowright}} \ddot{s}'$, then firstly $out_{\mathcal{S}'}(\ddot{s}') \cap L_U = \varnothing$, i.e., (2) holds. Secondly, since $\ddot{s} \neq \ddot{s}', \exists s \in \ddot{s}$ with $out_{\mathcal{S}'}(s) \cap L_U \neq \varnothing$, i.e., (3) holds. Thirdly, if additionally for $i \in L_I, \ddot{s}'' \in S$ we have $\ddot{s} \xrightarrow[d]{\delta_{\curvearrowright}} \ddot{s}' \xrightarrow[d]{i} \ddot{s}''$, then $\ddot{s} \xrightarrow[d]{i} \ddot{s}''$ since $\ddot{s}' \subseteq \ddot{s}$, i.e., (4) holds.

Let $\mathcal{S}$ meet all (1), (2), (3) and (4) and $\mathcal{S}'$ result from $\mathcal{S}$ by removing all $\delta_{\circlearrowleft}$ and hiding all $\delta_{\curvearrowright}$ (i.e., exchanged by τ). Then $\mathcal{S}'$ has no nondeterministic branching, only nondeterminism by internal transitions. We show by induction over the steps of the determinization that $\mathcal{S}'_{det} \rightleftharpoons \mathcal{S}$, each superstate of $\mathcal{S}'$ contains one state or two states connected by τ, and each outgoing transition leads to one state. (The induction can be generalized to also hold for not image finite $\mathcal{S}'$.)

For the base case, let $s := init_{\mathcal{S}}$, $s' := init_{\mathcal{S}'}$, $\ddot{s}' := s'$ *after* τ and $s'_{det} := init_{\mathcal{S}'_{det}}$. If $\tau \in enabled_{\mathcal{S}'}(s')$, then because of (2) $|\ddot{s}'| = 2$, and the two states are connected by τ. Because of (2) and (4), $\forall l \in L_I \dot\cup L_U \dot\cup \{\delta_{\curvearrowright}\} : |\ddot{s}' \text{ after}_{\mathcal{S}'_\delta} l| = 1$. Because of (3), $|\ddot{s}' \text{ after}_{\mathcal{S}'_\delta} \delta_{\circlearrowleft}| = 1$. Because of (1), $\delta_{\circlearrowleft} \in out_{\mathcal{S}'_{det}}(s'_{det})$ iff $\delta_{\circlearrowleft} \in out_{\mathcal{S}}(s)$, so $enabled_{\mathcal{S}}(s) = enabled_{\mathcal{S}'_{det}}(s'_{det})$.

For the induction step from n to $n+1$, let $\ddot{s}'$ be the superstate of $\mathcal{S}'$ in step $n+1$, s'_{det} the corresponding state in $\mathcal{S}'_{det}$, and s the corresponding state in $\mathcal{S}$. Each outgoing transition in step n led to one state, $s' \in S'$. Besides this initialization, proving the induction step is identical to the proof for the base case. □

Corollary 8.20. $\mathcal{LTS}(L_I, L_U, \tau) \approx_{Straces} deterministic\ \mathcal{LTS}(L_I, L_U, \delta_{\curvearrowright})$ *with reified quiescence.*

Lemma 8.21. *For Strace equivalent determinization, reified* $\delta_{\curvearrowright}$ *is required, so reified* $\delta_{\circlearrowleft}$*, or* δ *deduced instead of reified, is not sufficient:* $\mathcal{LTS}(L_I, L_U, \tau) \not\approx_{Straces} deterministic\ \mathcal{LTS}(L_I, L_U, \delta_{\circlearrowleft})$ *with reified quiescence, and* $\mathcal{LTS}(L_I, L_U, \tau) \not\approx_{Straces} deterministic\ \mathcal{LTS}(L_I, L_U)$.

Proof. For Subfig. 8.1a with $\tau = \tau_u$ and $a, b \in L_I$, $Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}) = a^*\delta^*b\delta^*$. Assume there were a deterministic Strace equivalent $\mathcal{S}_{det} \in \mathcal{LTS}(L_I, L_U, \delta_{\circlearrowleft})$ with reified quiescence or $\mathcal{S}_{det} \in \mathcal{LTS}(L_I, L_U)$ with δ deduced (cf. Def. 8.8), then $\delta \in enabled_{\mathcal{S}_{det}}(init_{\mathcal{S}_{det}})$ since $\delta b \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$. Furthermore, $a \in enabled_{\mathcal{S}_{det}}(init_{\mathcal{S}_{det}})$ since $ab \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$. But δ deduced or $\delta_{\circlearrowleft}$ does not change the current state, so δab or $\delta_{\circlearrowleft} ab$ would also be in $Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$, which is a contradiction. □

8.2.6. Enabledness

Definition 8.22. Let $\mathcal{S} = (S, \rightarrow, L) \in \mathcal{LTS}(L_I, L_U, *)$, $A \subseteq L$ with $\tau \notin A$, and $s \in S$. Then:

- s is A-**enabled** (also called A-**triggered**) iff $A \subseteq \mathit{enabled}_{\mathcal{S}_{\tau^*}}(s)$; otherwise, s is A-**inhibited**;
- $\mathcal{S}$ is A-**enabled** (also called A-**complete**) iff all its states are A-enabled;
- $\boldsymbol{\mathcal{IOTS}_A(L_I, L_U)}$ denotes the set of all A-enabled $\mathcal{S} \in \mathcal{LTS}(L_I, L_U)$ (analogously for $\boldsymbol{\mathcal{IOTS}_A(L_I, L_U, x_1, \ldots, x_n)}$, $\boldsymbol{\mathcal{IOTS}_A(L_I, L_U, x_1, \ldots, x_n, +)}$ and $\boldsymbol{\mathcal{IOTS}_A(L_I, L_U, x_1, \ldots, x_n, *)}$);
- for $A = L_I$, A-enabled s (respective $\mathcal{S}$) are called **input-enabled**;
- for $A = L_U$, A-enabled s (respective $\mathcal{S}$) are called **output-enabled**.

Any $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, *)$ can be transformed into a $\mathcal{S}' \in \mathcal{IOTS}_{L_I}(L_I, L_U, *)$ by **input completion**, e.g., by angelic completion or by demonic completion:

Angelic completion of $\mathcal{S}$ makes the lax assumption that the SUT is able to simply ignore input not specified in $\mathcal{S}$ and continue as if the input had not been given. This transformation corresponds to adding self-loops, as defined in Def. 8.23.

Definition 8.23. Let $\mathcal{S} = (S, \rightarrow, L) \in \mathcal{LTS}(L_I, L_U, *)$. Then:

$$\boldsymbol{\mathcal{S}_{\uparrow}} := (S, \underset{\uparrow}{\rightarrow}, L) \in \mathcal{IOTS}_{L_I}(L_I, L_U, *) \text{ is } \mathcal{S}\text{'s } \textbf{angelic completion}, \text{ with}$$
$$\underset{\uparrow}{\rightarrow} := \rightarrow \dot{\cup} \{(s, \xrightarrow{i}, s) \mid s \in S, i \in L_I \text{ with } i \notin \mathit{in}_{\mathcal{S}_{\tau^*}}(s)\}.$$

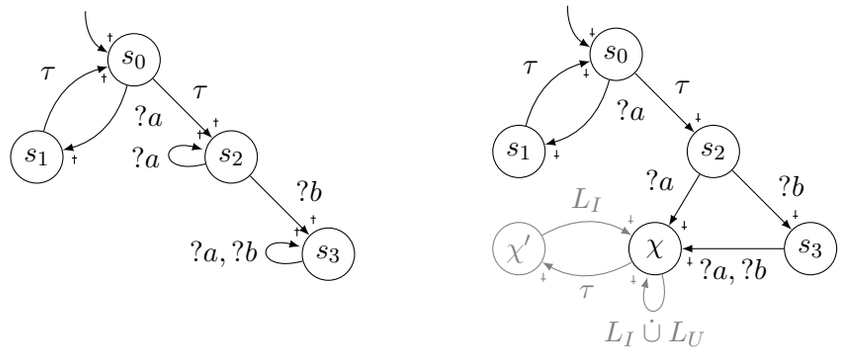
Demonic completion of $\mathcal{S}$ conservatively makes no assumptions on how the SUT handles input not specified in $\mathcal{S}$, so any behavior may occur. Consequently, the transformation adds transitions to a chaos state, from which any behavior may occur, as defined in Def. 8.24.

Definition 8.24. Let $\mathcal{S} = (S, \rightarrow, L) \in \mathcal{LTS}(L_I, L_U, *)$. Then:

$$\boldsymbol{\mathcal{S}_{\downarrow}} := (S \dot{\cup} \{\chi\}, \underset{\downarrow}{\rightarrow}, L) \in \mathcal{IOTS}_{L_I}(L_I, L_U, *) \text{ is } \mathcal{S}\text{'s } \textbf{demonic completion}, \text{ with}$$
$$\underset{\downarrow}{\rightarrow} := \rightarrow \dot{\cup} \{(s, \xrightarrow{i}, \chi) \mid s \in S, i \in L_I \text{ with } i \notin \mathit{in}_{\mathcal{S}_{\tau^*}}(s)\}, \text{ where}$$
$$\chi := \text{ the } \textbf{chaos state}, \text{ with } \mathit{in}_{\mathcal{S}_{\tau^*}}(\chi) = L_I \text{ and } \mathit{out}_{\mathcal{S}_{\tau^*\delta}}(\chi) = L_U \dot{\cup} \{\delta\},$$
$$\text{all self-loops, i.e., leading back to } \chi.$$

Example 8.25. Subfig. 8.2a shows the angelic completion for $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ from Subfig. 8.1a. Subfig. 8.2b shows the demonic completion for $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ from Subfig. 8.1a. Chaos χ can itself be modeled by an $\mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$, e.g., as depicted gray in Subfig. 8.2b, or in [Tretmans, 2008, Fig. 5] using three states. So in these cases, S is not extended by χ, but by those two or three states.

Note. The angelic and demonic completions might add more transitions than necessary to $\mathcal{S}$, because of τ, but this does not cause any problems.



(a) Angelic completion $\mathcal{S}_{\uparrow}$ **(b)** Demonic completion $\mathcal{S}_{\downarrow}$

Figure 8.2.: Angelic and demonic completions of $\mathcal{S}$ from Subfig. 8.1a

8.3. System Specifications

As described in Sec. 2.4, **system specifications** (**specifications** for short) define what behavior the SUT should conform to (cf. Sec. 8.5) and are the source for test case generation algorithms (cf. Sec. 8.8 and Fig. 8.5). The default definitions for specifications are given in Def. 8.26, alternatives are enumerated afterward.

Definition 8.26. *SPEC* denotes the set of all specifications.
$SPEC = \mathcal{LTS}(L_I, L_U, \tau)$.

This does not determine the semantics of τ. To do so, we use $\tau_{ut}, \tau_{u\not t}, \tau_{\not u t}$ or $\tau_{\not u \not t}$ instead, or multiple ones to mix semantics, e.g., $SPEC = \mathcal{LTS}(L_I, L_U, \tau_{ut}, \tau_{\not u \not t})$.

Similarly to behavioral properties for temporal logics (cf. Sec. 4.2), the ioco theory (cf. Sec. 8.5) does not consider the states in $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$, only the derived set $Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$ of suspension traces (or variations thereof, cf. Chapter 9). So as long as the resulting set of suspension traces stays the same, $\mathcal{S}$ can be replaced by another specification, without having to modify any other part of the ioco theory, enabling alternative definitions of *SPEC*, e.g., one of the following:

- $SPEC$ = deterministic $\mathcal{LTS}(L_I, L_U, \delta_{\curvearrowright})$ with reified quiescence (cf. Cor. 8.20);
- $SPEC = \mathcal{LTS}(L_I, L_U, \epsilon, \delta_{\circlearrowleft})$ with reified quiescence (cf. Lemma 8.15).

Contrarily, the meaning of *ioco* would change for $SPEC$ = deterministic $\mathcal{LTS}(L_I, L_U, \delta_{\circlearrowleft})$ with reified quiescence or $SPEC$ = deterministic $\mathcal{LTS}(L_I, L_U)$ since they are not Strace equivalent to $\mathcal{LTS}(L_I, L_U, \tau)$ (cf. Lemma 8.21). Many more variations could be investigated (deterministic vs. nondeterministic, finite vs. infinite, τ_u vs. $\tau_{\not u}$, $\delta_{\circlearrowleft}$ vs. $\delta_{\curvearrowright}$ vs. δ deduced). But the ones above are the most relevant since they are used in practice and also as alternatives for *MOD* (with the restriction of being input-enabled, cf. Sec. 8.4). Many Strace equivalences are direct implications of the ones above, e.g., $\mathcal{LTS}(L_I, L_U, \tau) \approx_{Straces} \mathcal{LTS}(L_I, L_U, \epsilon)$ with δ deduced, as described in Example 8.11. Therefore, we only show one more extreme case in Lemma 8.27.

Lemma 8.27. *$\mathcal{LTS}(L_I, L_U, \tau) \approx_{Straces} \mathcal{LTS}(L_I, L_U)$, but $\mathcal{S}' \in \mathcal{LTS}(L_I, L_U)$ that is trace equivalent to $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ might require nondeterministic branching, infinite state space and more than one initial state, even if $\mathcal{S}$ does not.*

Proof. Since $\mathcal{LTS}(L_I, L_U) \subsetneq \mathcal{LTS}(L_I, L_U, \tau)$ (cf. Sec. 8.2), we only need to show that for a $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$, there is a $\mathcal{S}' \in \mathcal{LTS}(L_I, L_U)$ with $\mathcal{S} \approx_{Straces} \mathcal{S}'$, which we do by constructing $\mathcal{S}'$ as computation tree (cf. Def. 4.4) that contains exactly the same *Straces*, using induction over the length of the suspension trace $\sigma \in Straces_{\mathcal{S}}(init_{\mathcal{S}})$: For the base case, $\sigma = \epsilon$, so $\sigma \in Straces_{\mathcal{S}'}(init_{\mathcal{S}'})$. For the induction step, let $\sigma = \sigma' \cdot l$ with $\sigma, \sigma' \in Straces_{\mathcal{S}}(init_{\mathcal{S}})$ and $l \in L_\delta$. Because of the induction hypothesis, $\sigma' \in Straces_{\mathcal{S}'}(init_{\mathcal{S}'})$. Let $\pi \in paths(\mathcal{S}')$ with $trace(\pi) = \sigma'$. We extend $\mathcal{S}'$ to contain σ: If $l = \delta$, we create a copy s_δ of $dest_{\mathcal{S}'}(\pi)$ by making a copy of the last transition $s_{|\pi|-1} \xrightarrow{l_{|\pi|}} s_{|\pi|}$, i.e., $s_{|\pi|-1} \xrightarrow{l_{|\pi|}} s_\delta$. If $l \in L_U$, we attach new outgoing transitions l to new states for all $s \in init_{\mathcal{S}'}\ after\ \sigma' \setminus s_\delta$, so that $\delta(s_\delta)$ is retained. If $l \in L_I$, we attach new outgoing transitions l to new states for all $s \in init_{\mathcal{S}'}\ after\ \sigma'$ (cf. Sec. 9.2).

This adds exactly all trace $\sigma' \cdot l \in Straces_{\mathcal{S}}(init_{\mathcal{S}})$ to $\mathcal{S}'$, so $\mathcal{S} \approx_{Straces} \mathcal{S}'$.

Nondeterministic branching is introduced by making a copy s_δ of a state. If $\mathcal{S}$ contains a loop, the state space of $\mathcal{S}'$ becomes infinite, even if the state space of $\mathcal{S}$ is finite. If $out(init_{\mathcal{S}}\ after\ \tau)$ contains δ and some $u \in L_U$, we need two initial states in $\mathcal{S}'$ (one state $init_\delta$ for δ). □

Many languages can be used to describe these specification. Some popular description languages are described in Subsec. 3.4.3.

8.4. *MOD*

Subsec. 8.1.2 has shown that we abstract from *SUT* to the set of all models, *MOD*, to enable formal reasoning: The test hypothesis guarantees the existence of an appropriate $\mathbb{M} \in MOD$ to replace the SUT, such that conformance between the SUT and the specification can be formalized (cf. Fig. 8.5). But $\mathbb{M}$ is not an artifact that is operated on or at all available, just a theoretical construct. Therefore we need to execute the SUT to find out its behavior (cf. Sec. 8.7).

We consider an $i \in L_I$ to be the attempt to give that input, not its actual processing (e.g., Example 8.28). Therefore, all $\mathbb{M} \in MOD$ and all $\mathbb{S} \in SUT$ are input-enabled, resulting in Def. 8.29. Input-enabledness facilitates test case execution (cf. Sec. 8.7), underspecification (cf. Sec. 9.2), and defining the testing hypothesis, though the implementation of the test adapter becomes more difficult (cf. Subsec. 8.7.2).

Example 8.28. In Example 8.2, input 10*cent* means that we are pushing 10 cent against the coin slot, not that 10 cent are actually inserted and processed. Thus if the machine closes its slot, the input is still accepted by the machine, with the observation $\widetilde{10cent}$.

Definition 8.29. $MOD = \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$.

With these considerations, we can fulfill the testing hypothesis by choosing $L_I \rightleftharpoons I$ and $L_U \rightleftharpoons U$. Since we are black-box testing, the states of $\mathbb{S} \in SUT$ are not observable. The tester only considers $I \cup U$. Accordingly, our formal reasoning does not investigate the states in $\mathbb{M} \in MOD$, so $\mathbb{M}$ only defines the rules for the causality between the actions. So as for *SPEC*, only the derived set $Straces_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}})$ of suspension traces (or variations thereof, cf. Chapter 9) are considered in the ioco theory. Since the Strace

equivalence used in Sec. 8.3 also hold for input-enabled LTSs with I/O, *MOD* also has many alternatives, e.g.,:

- $MOD =$ deterministic $\mathcal{IOTS}_{L_I}(L_I, L_U, \delta_{\curvearrowright})$ with reified quiescence;
- $MOD = \mathcal{IOTS}_{L_I}(L_I, L_U, \epsilon, \delta_{\circlearrowleft})$ with reified quiescence;
- $MOD = \mathcal{IOTS}_{L_I}(L_I, L_U)$.

8.5. Implementation Relation ioco

Having formalized *SUT* to *MOD* with the help of the testing hypothesis, conformance can now be defined as relation over $MOD \times SPEC$, called **implementation relation**.

This section introduce the *ioco* relation, which has its roots in the theory of testing- and refusal-equivalences for transition systems [Tretmans, 2008]. It decides when a model $\mathbb{M} \in MOD$ conforms to a specification $\mathcal{S} \in SPEC$ (cf. Fig. 8.5). For this, traces in a set $\mathcal{F} \subseteq L_\delta^*$ are considered, along which the input and output behavior of $\mathbb{M}$ must conform to that of $\mathcal{S}$:

1. the input considered by the paths in $\mathcal{F}$ must be specified by $\mathcal{S}$;
2. all possible output of $\mathbb{M}$ (including δ) must be specified by $\mathcal{S}$.

Because of Item 1, the largest possible set without senseless traces is $\mathcal{F} = Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$, which is what the default ioco theory in Def. 8.30 uses. Chapter 9 will introduce generalizations and variants.

Definition 8.30. Let model $\mathbb{M} \in MOD$, specification $\mathcal{S} \in SPEC$, $\mathcal{F} = Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$.

Then $\mathbb{M}$ ***ioco*** $\mathcal{S} :\Leftrightarrow \forall \sigma \in \mathcal{F}:$

$$out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}}\ after_{\mathbb{M}_{\tau^*\delta}}\ \sigma) \subseteq out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}\ after_{\mathcal{S}_{\tau^*\delta}}\ \sigma)$$

Notes. Since $init_{\mathbb{M}}\ after_{\mathbb{M}_{\tau^*\delta}}\ \sigma$ is already τ-closed, Def. 8.30 remains unchanged if $out_{\mathbb{M}_{\tau^*\delta}}(\cdot)$ does not use τ abstractions, i.e., $out_{\mathbb{M}_\delta}(\cdot)$ instead. The same applies for $\mathcal{S}$.

So $\mathbb{S} \in SUT$ conforms to $\mathcal{S}$ iff $\mathbb{M} ioco \mathcal{S}$ for a corresponding model $\mathbb{M} \in MOD$ according to the testing hypothesis. To denote that conformance is checked by the *ioco* relation, we also say $\mathbb{S} ioco \mathcal{S}$.

So for $\mathbb{M} \not{ioco} \mathcal{S}$, there must be a $\sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$ and a $u \in out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}} after_{\mathbb{M}_{\tau^*\delta}} \sigma) \setminus out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}\ after_{\mathcal{S}_{\tau^*\delta}}\ \sigma)$. This can only be the case if $out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}\ after_{\mathcal{S}_{\tau^*\delta}}\ \sigma) \neq L_U \dot\cup \{\delta\}$, leading to Def. 8.31.

Definition 8.31. Let $\mathcal{S} = (S, \rightarrow, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau, *)$ and $s \in S$.

Then $\boldsymbol{faultable(S_{det})} := \{\ddot{s} \in S_{det} | out_{\mathcal{S}_{det}}(\ddot{s}) \neq L_U \dot\cup \{\delta\}\}$, baptized **faultable states** of $\mathcal{S}_{det}$.

We can reduce $\mathcal{F}$ by excluding suspension traces $\sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$ that are not faultable or redundant:

- σ with $init_{\mathcal{S}}\ after\ \sigma \notin faultable(S_{det})$ are not faultable since no output leads to fail;
- σ that end with δ are not faultable because δ only reaches quiescent states, so the testing hypothesis guarantees that no output $u \in L_U$ occurs in them. Therefore, no output leading to fail can occur in the SUT;

- σ that contain $\delta \cdot \delta$ are redundant because after the first δ, only quiescent states are reached, so the testing hypothesis guarantees that no output $u \in L_U$ occurs. Therefore, the second δ cannot cause fail and simply makes a self-loop (cf. Lemma 8.19).

Thus $\mathcal{F}$ in Def. 8.30 can be reduced from $Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$ to $faultable(Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}))$, as given in Def. 8.32, which is similar to *Rtraces* of [Volpato and Tretmans, 2013] used on *Utraces* (cf. Sec. 9.2) for coverage (cf. Chapter 2 and Subsec. 8.8.4 and Chapter 12).

Definition 8.32. Let $\mathcal{S} = (S, \rightarrow, L) \in \mathcal{LTS}(L_I, L_U, *)$ and $\ddot{s} \in S_{det}$.
Then $\boldsymbol{faultable(Straces_{\mathcal{S}_{\tau^*\delta}}(\ddot{s}))} := \{\sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(\ddot{s}) \mid \sigma = (l_i)_{i\in[1,\dots,|\sigma|]}$ and $\nexists k \in [1, \dots, |\sigma| - 1] : l_k = l_{k+1} = \delta$ and $l_{|\sigma|} \neq \delta$ and $init_{\mathcal{S}}$ *after* $\sigma \in faultable(S_{det})\}$, baptized **faultable reduced suspension traces**.

Example 8.33. Since input completion in $\mathcal{S}_{\downarrow}$ from Subfig. 8.2a only adds input to $\mathcal{S}$ from Subfig. 8.1a, but does not change the outputs on suspension traces already present, $\mathcal{S}_{\downarrow} ioco \mathcal{S}$. For $\mathcal{S}_{\downarrow}$ and its τ interpreted as τ_u, we have $init\ after_{\mathcal{S}_{\downarrow}}\ \delta?a = \{\chi, \chi'\}$, so $out_{\mathcal{S}_{\downarrow}}(init after \delta?a) = L_U \dot{\cup} \{\delta\}$. For $\mathcal{S}$ and its τ interpreted as $\tau_{\not\mu}$, we have $init_{\mathcal{S}} after \delta_{\not\mu}?a = \{s_0, s_1, s_2\}$, so $out_{\mathcal{S}}(init\ after\ \delta_{\not\mu}?a) = \{\delta_{\not\mu}\}$. Therefore ($\mathcal{S}_{\downarrow}$ with $\tau = \tau_u$) $\not{ioco}$ ($\mathcal{S}$ with $\tau = \tau_{\not\mu}$) if $L_U \neg \varnothing$.

8.6. Test Cases

Whereas $\mathcal{S}$ specifies how the SUT should behave, a test case $\mathbb{T}$ specifies how the test, an experiment on the SUT, should behave during its execution of $\mathbb{T}$. Since we want to check whether $\mathbb{M}$ *ioco* $\mathcal{S}$, each test case follows suspension traces of $\mathcal{S}$. Therefore, test cases can also be described by LTSs in $\mathcal{LTS}(L_I, L_U, *)$, but of a special kind that meets the following demands: Since test cases will be executed (cf. Sec. 8.7 and Fig. 8.5), nondeterminism of $\mathcal{S}$ must be resolved. As described in Subsec. 8.2.5, controllable nondeterminism is resolved by either trying to give one input or waiting for an output. If an input is given, the tester picks one amongst all the enabled inputs. Since the SUT can preempt test input by giving an output, all outputs must always be accounted for, such that verdicts can be made no matter what output occurs. Thus test cases are output-enabled and not traces, but contain branching; because of nondeterminism on output, test cases are not degenerated: more than one output can lead to subsequent test steps instead of the verdict fail. Nondeterminism of the LTS is also resolved similarly to Subsec. 8.2.5 by considering all possible nondeterministic cases and states the SUT is potentially in.

Depending on where uncontrollable nondeterminism occurs, and on what choices the SUT has made so far, it might or might not be useful to revisit a certain state of the model during test case execution. Therefore, a test case $\mathbb{T}$ forbids loops, but rather unwinds $\mathcal{S}$ to control how often a specific state can be revisited, to make $\mathbb{T}$ as meaningful (cf. Subsec. 2.5) as possible. Besides unwinding loops, we also replicate other superstates, such that $\mathbb{T}$ becomes a tree (cf. Def. 4.4). Without this replication, test cases would more generally be single-rooted, connected, directed, acyclic graphs, i.e., states would be shared amongst different traces, saving memory. But often test trees are used [Tretmans, 2008], and they are necessary when different traces must be treated differently, e.g., for our heuristics (cf. Chapter 12) of our lazy on-the-fly MBT (cf. Chapter 11).

Since test cases are executed, only finite traces are relevant. Thus $\mathbb{T}$ should not contain infinite paths.

We guarantee that each test case results in a verdict by demanding that each leaf is a **verdict leaf**: a pass or a fail .

Test cases in $\mathcal{LTS}(L_I, L_U, *)$ that meet all the demands above result in Def. 8.34. Def. 8.35 shows how test cases can be extended by concatenation.

Definition 8.34. Let $\mathbb{T} = (S, \rightarrow, L_{\delta_\frown}) \in \mathcal{IOTS}_{L_U}(L_I, L_U, \delta_\frown)$. Then:

$\mathbb{T}$ is a **test case** (**TC**) iff all of the following points hold:

- **pass**, **fail** $\in S$ are two different states and called **verdict states**;
- $\mathbb{T}$ is a tree with the leafs pass and fail ;
- $\forall i \in L_I \; \forall s \in S : \; s \not\xrightarrow{i}$ fail ;
- $\mathbb{T}$ only has finite paths: $\forall \pi \in paths_{max}(\mathbb{T}) : |\pi| \in \mathbb{N}$;
- $\forall s \in S \setminus \{\text{pass}, \text{fail}\} \;\; \exists \boldsymbol{i_\delta} \in L_I \dot\cup \{\delta_\frown\} : \; enabled_{\mathbb{T}}(s) = L_U \dot\cup \{i_\delta\}$.

$\boldsymbol{\mathcal{TTS}(L_I, L_U, \delta)}$ denotes the set of all test cases in $\mathcal{IOTS}_{L_U}(L_I, L_U, \delta_\frown)$.

A countable subset $\ddot{\mathbb{T}} \subseteq \mathcal{TTS}(L_I, L_U, \delta)$ is called a **test suite** (**TS**).

Definition 8.35. Let $\mathbb{T}_1, \mathbb{T}_2 \in \mathcal{TTS}(L_I, L_U, \delta)$, with $\mathbb{T}_1 = (S_1, \rightarrow_1, L_{\delta_\frown})$ and $\mathbb{T}_2 = (S_2, \rightarrow_2, L_{\delta_\frown})$ where $init_{\mathbb{T}_2} \in S_1$ and $init_{\mathbb{T}_2} \rightarrow_1$ pass in $\mathbb{T}_1$.

Then the **concatenation of TCs** $\mathbb{T}_1 \cdot \mathbb{T}_2$ is the TC $(S_1 \cup S_2, \rightarrow, L_{\delta_\frown})$ with $\rightarrow = \rightarrow_2 \dot\cup (\rightarrow_1 \setminus \{init_{\mathbb{T}_2} \rightarrow_1 \text{pass}\})$, i.e., the pass after $init_{\mathbb{T}_2}$ in $\mathbb{T}_1$ is replaced by $\mathbb{T}_2$.

Notes 8.36. Since we always use these kind of trees as test cases, we always set $TEST = \mathcal{TTS}(L_I, L_U, \delta)$; so unlike *SPEC* and *MOD*, we do not need the abstraction *TEST*.

Since test execution (cf. Sec. 8.7) terminates in pass and fail anyways, there is no need for the verdict states to be output-enabled. Formally, we can define $\mathbb{T}$ to be a tree besides the verdict states, and on each path a verdict state must eventually occur. Similarly to chaos (cf. Def. 8.24), pass and fail can be rendered output-enabled by demanding $out_{\mathbb{T}}(\text{fail}) = out_{\mathbb{T}}(\text{pass}) = L_U \dot\cup \{\delta\}$, all self-loops. By including δ, the constraint on $enabled_{\mathbb{T}}(s)$ of Def. 8.34 also holds for $s =$ pass and $s =$ fail . Besides pass and fail , TCs remain to be trees.

As always, the single initial state is called *init* or $init_{\mathbb{T}}$ if not named explicitly.

Test cases have the following properties:

- Even though $\mathbb{T} = (S, \rightarrow, L_\delta) \in \mathcal{TTS}(L_I, L_U, \delta)$ only contains finite paths, the depth of $\mathbb{T}$, $\boldsymbol{depth(\mathbb{T})}$, can be infinite for infinite $|L_U|$. If $\mathbb{T}$ is **finitely branching**, i.e., $branch_{\mathbb{T}}$ is finite, then the depth of $\mathbb{T}$ is finite [König, 1936], as is its **size**, i.e., $|\mathbb{T}| = |S| + |\rightarrow| \in \mathbb{N}$;
- Having resolved nondeterminism of the LTS, $\mathbb{T}$ is a deterministic LTS;
- pass states are used to indicate termination of test runs. State fail indicates a failure of the SUT and also the termination of the test run, since a failure might cause an **inconsistent state** of the SUT.

Since failures often occur close to each other ("bugs are social" [Kervinen et al., 2005; Anand et al., 2013]), it would be helpful to continue at the failure state. This could be done by hardening the test hypothesis with some constraint on the SUT: that it copes with failures in a specific way (e.g., via fault-tolerance [Dubrova, 2013] or strong exception safety [Abrahams, 1998] or via failure recovery like Erlang's backward recovery [Nyström, 2009]), or that it can be guided to some consistent state (e.g., via homing

sequences [Lee and Yannakakis, 1996]). So many solutions are possible. But continuing in a state after a failure is only an optimization and not necessary in this thesis because it designs a **robust testing process** that restarts after failures during test execution (cf. reset capability in the testing hypothesis in Subsec. 8.1.2 and test case sequences in Def. 11.8). Therefore, fault-tolerance of the SUT is future work.

Example 8.37. Fig. 8.4 on page 214 depicts the beginning of three TCs: $\mathbb{T}_1$, $\mathbb{T}_2$ and $\mathbb{T}_3$.

8.7. Test Case Execution

Subsec. 8.7.1 formally describes how a TC is executed in an abstract way on a model $\mathbb{M} \in MOD$. Subsec. 8.7.2 shows how a test adapter binds abstract models in MOD to SUTs in SUT. Therefore, test cases can be executed on the SUT.

8.7.1. Test Case Execution on MOD

In a **test case execution** (test execution for short) of the test case $\mathbb{T} \in \mathcal{TTS}(L_I, L_U, \delta)$ on the model $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ (cf. Sec. 8.4),

- $\mathbb{T}$ chooses the input, which **drives** $\mathbb{M}$;
- $\mathbb{M}$ chooses the output, leading to a successive state of $\mathbb{T}$, possibly a verdict, resulting in the **test oracles**.

Therefore, a test execution of $\mathbb{T}$ on $\mathbb{M}$ corresponds to their synchronous parallel product $\mathbb{T}||\mathbb{M}_{\delta\tau^*}$ (cf. Def. 3.32). Since $\mathbb{M}$ is input-enabled, it always accepts $\mathbb{T}$'s input; since $\mathbb{T}$ is output-enabled, it always accepts $\mathbb{M}$'s output. Thus $\mathbb{T}||\mathbb{M}_{\delta\tau^*}$ never deadlocks and yields a test run σ of $\mathbb{T}$ on $\mathbb{M}$: a maximal trace through both $\mathbb{T}$ and $\mathbb{M}$ that terminates in a leaf of $\mathbb{T}$, yielding the verdict of σ.

Due to uncontrollable nondeterminism, different test executions of $\mathbb{T}$ on $\mathbb{M}$ can result in different test runs. Since $\mathbb{M}$ should behave according to $\mathbb{T}$ for all nondeterministic resolutions, all possible test runs must pass for the whole test case $\mathbb{T}$ to pass. These concepts are defined in Def. 8.38 and Def. 8.39. For full certainty, the test case have to be repeatedly executed an unknown number of times, but finite due to fairness. Under which circumstances the SUT must exhibit all nondeterministic resolutions depends on the king of fairness included in the testing hypothesis. These will be described in Subsec. 8.8.4 after having introduced test case generation algorithms and exhaustiveness.

Definition 8.38. Let $\mathbb{T} \in \mathcal{TTS}(L_I, L_U, \delta)$ and $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$. Then:

- $(\mathbb{T}||\mathbb{M}_{\delta\tau^*}) \in \mathcal{LTS}(L_I, L_U, \delta)$ models all possible **test executions** of $\mathbb{T}$ on $\mathbb{M}$;
- a path $\pi \in paths^{fin}(\mathbb{T}||\mathbb{M}_{\delta\tau^*})$ resembles one nondeterministic resolution of $\mathbb{T}||\mathbb{M}_{\delta\tau^*}$ and is called **test run path**;
- a path $\pi \in paths^{fin}_{max}(\mathbb{T}||\mathbb{M}_{\delta\tau^*})$ is called **maximal test run path**;
- a suspension trace $\sigma \in traces^{fin}(\mathbb{T}||\mathbb{M}_{\delta\tau^*})$ is called a **test run trace** (**test run** for short) of $\mathbb{T}$ on $\mathbb{M}$;
- a test run $\sigma \in traces^{fin}_{max}(\mathbb{T}||\mathbb{M}_{\delta\tau^*})$ is called a **maximal test run trace** (**maximal test run** for short) of $\mathbb{T}$ on $\mathbb{M}$;
- $\boldsymbol{\mathcal{F}_{mod}} : paths^{fin}(\mathbb{T}||\mathbb{M}_{\delta\tau^*}) \to paths^{fin}(\mathbb{T}), ((t_{i-1}, s_{i-1}) \xrightarrow{l_i} (t_i, s_i)) \mapsto (t_{i-1} \xrightarrow{l_i} t_i)$ forgets the model $\mathbb{M}$ component of the test run path, and is called **forgetful transformation for model component**;
- $\boldsymbol{\mathcal{F}_{TC}} : paths^{fin}(\mathbb{T}||\mathbb{M}_{\delta\tau^*}) \to paths^{fin}(\mathbb{M}_{\delta\tau^*}), ((t_{i-1}, s_{i-1}) \xrightarrow{l_i} (t_i, s_i)) \mapsto (s_{i-1} \xrightarrow{l_i} s_i)$ forgets the test case $\mathbb{T}$ component of the test run path, and is called **forgetful transformation for TC component**;
- the **verdict** of a maximal test run path π is $dest(\mathcal{F}_{mod}(\pi))$;
- the **verdicts** of $\mathbb{T}$ on $\mathbb{M}$ are $\boldsymbol{verd_{\mathbb{M}}(\mathbb{T})} := dest(\mathcal{F}_{mod}(paths^{fin}_{max}(\mathbb{T}||\mathbb{M}_{\delta\tau^*})))$.

Definition 8.39. Let model $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$, test case $\mathbb{T} \in \mathcal{TTS}(L_I, L_U, \delta)$, TS $\ddot{\mathbb{T}} \subseteq \mathcal{TTS}(L_I, L_U, \delta)$, maximal test run path $\pi \in paths^{fin}_{max}(\mathbb{T}||\mathbb{M}_{\delta\tau^*})$ and maximal test run $\sigma \in traces^{fin}_{max}(\mathbb{T}||\mathbb{M}_{\delta\tau^*})$. Then:

- $\mathbb{M}$ **passes** $\pi :\Leftrightarrow dest(\mathcal{F}_{mod}(\pi)) = \text{pass}$;
- $\mathbb{M}$ **passes** $\sigma :\Leftrightarrow \forall \pi' \in paths^{fin}_{max}(\mathbb{T}||\mathbb{M}_{\delta\tau^*})$ with $trace(\pi') = \sigma : \mathbb{M}$ passes π';
- $\mathbb{M}$ **passes** $\mathbb{T} :\Leftrightarrow verd_{\mathbb{M}}(\mathbb{T}) = \{\text{pass}\}$ (i.e., $\forall \pi' \in paths^{fin}_{max}(\mathbb{T}||\mathbb{M}_{\delta\tau^*}) : \mathbb{M}$ passes π');
- $\mathbb{M}$ **passes** $\ddot{\mathbb{T}} :\Leftrightarrow \forall \mathbb{T}' \in \ddot{\mathbb{T}} : \mathbb{M}$ passes $\mathbb{T}'$;
- $\mathbb{M}$ **fails** σ (resp. $\mathbb{T}$, resp. $\ddot{\mathbb{T}}$) $\Leftrightarrow \mathbb{M}$ ~~passes~~ σ (resp. $\mathbb{T}$, resp. $\ddot{\mathbb{T}}$);
- π (resp. σ, resp. $\mathbb{T}$) is a **counterexample-path** (resp. **-trace**, resp. **-test case**) to $\mathbb{M}$ $\Leftrightarrow \mathbb{M}$ fails π (resp. σ, resp. $\mathbb{T}$).

Notes. $Straces_{\mathbb{T}||\mathbb{M}_{\delta\tau^*}}(init) = Straces_{\mathbb{T}||\mathbb{M}_{\tau^*\delta}}(init)$ because of Lemma 8.15. We use $\mathbb{M}_{\delta\tau^*}$ to have $\delta_{\frown}$ as for TCs.

A test run path $\pi \in paths^{fin}(\mathbb{T}||\mathbb{M}_{\delta\tau^*})$ resolves nondeterminism on output by picking one. Nondeterministic branching must be resolved by considering all possibilities (cf. $\mathbb{M}$ passes σ in Def. 8.39) since they can lead to differing behavior.

$\mathbb{M}$ fails path $\pi \Leftrightarrow dest(\mathcal{F}_{mod}(\pi)) =$ fail .

Some papers call L_I output and L_U input (and correspondingly for enabledness and completion) when talking about a TC $\mathbb{T}$, due to the symmetry of $\mathbb{T}||\mathbb{M}_{\delta\tau^*}$. Since it is

simpler, we consistently call L_I input and L_U output, independent of the artifact we are talking about.

As Sec. 8.4 has shown, alternatives to $MOD = \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ can also be chosen.

8.7.2. Test Adapter

The last subsection has shown how to execute a test case on a model $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$, but $\mathbb{M}$ is just imaginary, for formalization. So we actually need to execute the test case on $\mathbb{S} \in SUT$, resulting in Def. 8.40.

Definition 8.40. To indicate that a test case $\mathbb{T} \in \mathcal{TTS}(L_I, L_U\delta)$ is executed on a model, it is also called **abstract test case**.

To execute abstract test cases on an SUT, a **binding $b(\cdot)$** is used: It consists of a mapping $\boldsymbol{b_I(\cdot)}$ between L_I and the SUT's input I and a mapping $\boldsymbol{b_U(\cdot)}$ between the SUT's output U with quiescence and $L_U \dot\cup \{\delta\}$. $b(\cdot)$ is implemented by a **test adapter**.

A test adapter that binds all actions in the abstract test case $\mathbb{T}$ to actions in $I \cup U \dot\cup \{\text{quiescence}\}$ yields a **concrete test case** $b(\mathbb{T})$ (also called **executable test case**).

The **verdicts** of executing $b(\mathbb{T})$ on $\mathbb{S}$ are $\boldsymbol{verd_{\mathbb{S}}(\mathbb{T})}$.

To fulfill the testing hypothesis's interface abstraction, we chose $L_I \rightleftharpoons I$ and $L_U \rightleftharpoons U$; consequently, the mapping is a lossless abstraction and the test adapter only needs to implement the bijection. More precisely, test execution on the SUT only requires the directions $b_I : L_I \to I$ and $b_U : U \dot\cup \{\text{quiescence}\} \to L_U \dot\cup \{\delta\}$. Depending on the SUT, this can be simple or technically challenging. The main technicalities for the test adapter to fulfill the testing hypothesis are:

- implementing the mapping b_U from a refusal directly after $b_I(i)$ for an $i \in L_I$ to $\widetilde{i} \in L_U$ (cf. Subsec. 8.1.2 and Sec. 8.4);
- handling race conditions between giving an input and receiving an output. Usually, input and output is implemented as atomic transactions, where an input is interruptible by an output if parts of giving an input can be undone;
- implementing waiting with timeouts, such that the system does not deadlock but detect quiescence. δ in a TC $\mathbb{T}$ causes the tester to wait, since $\delta \in enabled_{\mathbb{T}}(s)$ in a state s of $\mathbb{T}$ implies $L_I \cap enabled_{\mathbb{T}}(s) = \varnothing$. If no other output occurs until the timeout, then quiescence is observed and δ is taken in $\mathbb{T}$.

As mentioned in Subsec. 8.1.1, the SUT's interface cannot only be abstracted using interface abstractions, but also by **abstractions by the test adapter**: In this case, $b_I(\cdot)$ binds an element $i \in L_I$ not to an atomic input of the SUT, but to a more complex structure; likewise, $b_U(\cdot)$ can bind a more complex structure to an element $u \in L_U$. Hereby, the difference between abstract and concrete test cases becomes much larger, distributing the complexity between the model and the test adapter [van der Bijl et al., 2005; Prenninger and Pretschner, 2005]. While the interface abstractions are lossy, abstractions by the test adapter are usually lossless. More precisely, $b_U(\cdot)$ performs abstraction, $b_I(\cdot)$ refinement. Abstractions by the test adapter are hence often called **action refinement**.

Notes 8.41. There are many variations of this kind of abstraction and action refinement for conformance testing in practice, but few have been published [van der Bijl et al.,

2005; Fraser et al., 2009; Nieminen et al., 2011]. The properties of most of them are not yet well researched, although action refinement is highly relevant in practice. One solid reference is [van der Bijl et al., 2005], which investigates a more general action refinement than described above: It allows replacing an abstract atomic transition (called atomic refinement) or even a whole suspension trace with another suspension trace (called linear refinement) or even with a whole tree. These refinements can become very complex, so a prominent use-case is atomic linear refinement (called atomic linear input-inputs refinement in [van der Bijl et al., 2005]), which maps an atomic input to a trace of inputs (similar to keywords in keyword driven testing [Brandes et al., 2015]). Having thoroughly introduced the testing hypothesis, abstractions and the test adapter, action refinement can be investigated on top of this foundation as future work; one promising approach is binding an element $i \in L_I$ to a sequence of atomic inputs and fully determined atomic outputs of the SUT, and likewise binding a sequence of atomic outputs and fully determined atomic inputs to an element $u \in L_U$.

The term "**test case refinement**" sometimes stands for the mapping of abstract to concrete test cases, sometimes for the abstractions by the test adapter.

The binding does not determine which nondeterministic resolutions the SUT actually takes under which circumstances. This is covered by fairness criteria, investigated in Subsec. 8.8.4.

Test adapters, even when not doing elaborate abstractions, are still technical and time-consuming in practice: the cost of general test adapters can become the major part in MBT, cf. [Grieskamp et al., 2011] and the German articles [Weißleder et al., 2011; Faragó et al., 2013].

8.8. Test Case Generation

This section introduces abstract test case generation algorithms (**test generation** for short) that check *ioco* (cf. Fig. 8.5).

Roadmap. Subsec. 8.8.1 introduces the concept of test case generation algorithms. Subsec. 8.8.2 introduces the nondeterministic test case generation algorithm genTC, which is roughly similar to the standard literature [Tretmans, 2008; Frantzen, 2016], and shows some of its properties. Subsec. 8.8.3 shows that genTC contains redundant TCs and introduces the deterministic test suite generation algorithm genTS, an adaption of genTC. Subsec. 8.8.4 details the fairness constraints introduced in Subsec. 8.1.2 and properties they imply. Subsec. 8.8.5 introduces exhaustiveness thresholds for genTS and shows approximations for all kinds of fairness (cf. Table 8.1).

8.8.1. Introduction

Test case generation algorithms generate TCs (cf. Sec. 8.6) that check *ioco* (cf. Sec. 8.5). They not only help in implementing concrete algorithms, but also in comparing different concrete algorithms, as well as tools and testing processes that employ them. Test case generation algorithms are defined in Def. 8.42, Listing 8.1 determines their input, output and contract.

Definition 8.42. Let $\mathcal{S} \in SPEC$ be a system specification.

Then a **test case generation algorithm** (**gen** for short) is a computable algorithm that takes (at least) $\mathcal{S}$ as input to generate

- nondeterministically a single test case $\mathbb{T} \in \mathcal{TTS}(L_I, L_U, \delta)$ from $\mathcal{S}$. In this case, termination and the result of gen depend on the resolution of gen's nondeterministic choice points. We often identify $\text{gen}(\mathcal{S})$ with the set $\{\mathbb{T} \in \mathcal{TTS}(L_I, L_U, \delta) | \mathbb{T} = \text{gen}(\mathcal{S})$ for some nondeterministic resolution$\}$;
- or deterministically a test suite $\ddot{\mathbb{T}} \subseteq \mathcal{TTS}(L_I, L_U, \delta)$ from $\mathcal{S}$.

In later chapters, some test case generation algorithms integrate test execution and hence take $\mathbb{S} \in SUT$ as additional parameter. This thesis will use the notation $\mathbf{gen}_{exec}(\mathcal{S}, \mathbb{S})$ if this needs to be made explicit.

```
// PRE: S is a well-formed system specification in LTS(L_I, L_U, τ) (or
//      Strace equivalent structure or description thereof);
// POST: If gen(S) is nondeterministic, it need not terminate;
//      If it does, it returns a TS containing a single TC;
//      If gen(S) is deterministic, it must terminate and return a TS.
2^{TTS(L_I,L_U,δ)} gen(S)

// PRE: S is a well-formed system specification in LTS(L_I, L_U, τ) (or
//      Strace equivalent structure or description thereof);
//      S ∈ SUT;
// POST: If gen(S, S) is nondeterministic, it need not terminate;
//      If it does, it returns a TS containing a single TC;
//      If gen(S, S) is deterministic, it must terminate and return a TS;
//      For each returned TC T, a verdict in verd_S(T) may be given.
2^{TTS(L_I,L_U,δ)} gen_exec(S, S)
```

Listing 8.1: Contract for test case generation

Usually, only sound gen are considered. In the field of *ioco*, the term "**exhaustive**" is often used instead of complete, and some say complete to express both soundness and exhaustiveness. To avoid confusion, we will only use the term "exhaustive" in *ioco* and MBT, as defined in Def. 8.43. Unfortunately, exhaustiveness of gen does not have strong practical implications in the ioco theory: exhaustive test suites usually have infeasibly large size (cf. exhaustiveness threshold in Subsec. 8.8.3), and each of its test cases needs to be executed an unknown number of times (cf. Subsec. 8.7.1). But at least exhaustiveness implies that gen can potentially find every fault eventually, i.e., no faults are in advance ruled out to be detected.

Definition 8.43. Let TC $\mathbb{T} \in \mathcal{TTS}(L_I, L_U, \delta)$, TS $\ddot{\mathbb{T}} \subseteq \mathcal{TTS}(L_I, L_U, \delta)$, specification $\mathcal{S} \in LTS(L_I, L_U, \tau)$, SUT $\mathbb{S}_1 \in SUT$, $\mathbb{M}_1 \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ a corresponding model according to the testing hypothesis (cf. Def. 8.4), and $\text{gen}(\cdot)$, $\text{gen}_{exec}(\cdot, \cdot)$ test case gen-

eration algorithms. Then we call:

$$
\begin{aligned}
&\mathbb{T} \textbf{ sound for } \mathcal{S} && :\Leftrightarrow \forall \mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau) : (\mathbb{M} \text{ fails } \mathbb{T} \Rightarrow \mathbb{M} \not\mathit{ioco} \mathcal{S});\\
&\ddot{\mathbb{T}} \textbf{ sound for } \mathcal{S} && :\Leftrightarrow \forall \mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau) : (\mathbb{M} \text{ fails } \ddot{\mathbb{T}} \Rightarrow \mathbb{M} \not\mathit{ioco} \mathcal{S});\\
&\ddot{\mathbb{T}} \textbf{ sound for } \mathcal{S} \text{ and } \mathbb{S}_1 \\
& && :\Leftrightarrow (\mathbb{M}_1 \text{ fails } \ddot{\mathbb{T}} \Rightarrow \mathbb{M}_1 \not\mathit{ioco} \mathcal{S});\\
&\ddot{\mathbb{T}} \textbf{ exhaustive for } \mathcal{S} \\
& && :\Leftrightarrow \forall \mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau) : (\mathbb{M} \text{ fails } \ddot{\mathbb{T}} \Leftarrow \mathbb{M} \not\mathit{ioco} \mathcal{S});\\
&\ddot{\mathbb{T}} \textbf{ exhaustive for } \mathcal{S} \text{ and } \mathbb{S}_1 \\
& && :\Leftrightarrow (\mathbb{M}_1 \text{ fails } \ddot{\mathbb{T}} \Leftarrow \mathbb{M}_1 \not\mathit{ioco} \mathcal{S});\\
&\text{gen}(\cdot) \textbf{ sound} && :\Leftrightarrow \forall \mathcal{S} \in LTS(L_I, L_U, \tau) : \text{gen}(\mathcal{S}) \text{ is sound for } \mathcal{S};\\
&\text{gen}(\cdot) \textbf{ exhaustive} && :\Leftrightarrow \forall \mathcal{S} \in LTS(L_I, L_U, \tau) : \text{gen}(\mathcal{S}) \text{ is exhaustive for } \mathcal{S};\\
&\text{gen}_{exec}(\cdot,\cdot) \textbf{ sound} && :\Leftrightarrow \forall \mathcal{S} \in LTS(L_I, L_U, \tau)\ \forall \mathbb{S} \in SUT : \\
& && \qquad \text{gen}_{exec}(\mathcal{S}, \mathbb{S}) \text{ is sound for } \mathcal{S} \text{ and } \mathbb{S};\\
&\text{gen}_{exec}(\cdot,\cdot) \textbf{ ex-} && :\Leftrightarrow \forall \mathcal{S} \in LTS(L_I, L_U, \tau)\ \forall \mathbb{S} \in SUT : \\
&\textbf{haustive} && \qquad \text{gen}_{exec}(\mathcal{S}, \mathbb{S}) \text{ is exhaustive for } \mathcal{S} \text{ and } \mathbb{S}.
\end{aligned}
$$

Notes. Thus, a false positive from an unsound gen is a $\mathbb{T} \in \text{gen}(\mathcal{S})$ for which $\mathbb{M}$ fails (i.e., $\mathbb{T}$ is a witness for $\mathbb{M} \not\mathit{ioco} \mathcal{S}$), when in fact $\mathbb{M}\ \mathit{ioco}\ \mathcal{S}$ and $\mathbb{T}$ should pass. Conversely, a false negative from an inexhaustive gen is a TS $\text{gen}(\mathcal{S})$ for which $\mathbb{M}$ passes (i.e., the statement that $\mathbb{M}\ \mathit{ioco}\ \mathcal{S}$), when in fact $\mathbb{M} \not\mathit{ioco} \mathcal{S}$ and $\mathbb{M}$ should fail for some TC in $\text{gen}(\mathcal{S})$.

If $\mathcal{S}$ is image finite, gen only generates TCs that have finite depth and finite state space (cf. Note 8.36).

There is a strong connection between $\text{gen}(\cdot)$ and $\text{gen}_{exec}(\cdot,\cdot)$: Since most $\text{gen}(\cdot)$ iteratively generate TCs, a generated TC can be executed on-the-fly on a given $\mathbb{S}$ before the next TC is generated, resulting in an algorithm $\text{gen}_{exec}(\cdot,\cdot)$ which inherits soundness and exhaustiveness from $\text{gen}(\cdot)$. Conversely, if $\text{gen}_{exec}(\cdot,\cdot)$ generates TCs independently of the given $\mathbb{S}$ and only uses $\mathbb{S}$ to execute the generated TCs, the TC generation of $\text{gen}_{exec}(\cdot,\cdot)$ can be isolated into an algorithm $\text{gen}(\cdot)$ by reordering TC generation and execution, i.e., firstly generating the full TS $\text{gen}_{exec}(\mathcal{S},\cdot)$ for a given $\mathcal{S}$ and thereafter executing it on a given $\mathbb{S}$. Though this might not be practical due to the size of $\text{gen}_{exec}(\mathcal{S},\cdot)$, it allows us to apply the following lemmas about $\text{gen}(\cdot)$ also to $\text{gen}_{exec}(\cdot,\cdot)$. If the generation of TCs in $\text{gen}_{exec}(\cdot,\cdot)$ does depend on the given $\mathbb{S}$, it can restrict the TCs that are generated. But we can use a special simulated SUT (cf. Lemma 8.69) $\mathbb{S}_{sim\,\mathcal{S}}$ to transform a sound and exhaustive $\text{gen}_{exec}(\cdot,\cdot)$ into a sound and exhaustive $\text{gen}(\cdot)$ by using $\text{gen}_{exec}(\mathcal{S}, \mathbb{S}_{sim\,\mathcal{S}})$ for a given $\mathcal{S}$. Therefore, we focus on $\text{gen}(\cdot)$ in this chapter.

8.8.2. Nondeterministic Test Case Generation genTC

$\text{gen}(\mathcal{S})$ has to deal with nondeterminism when generating TCs; Sec. 8.6 has already described how nondeterminism is resolved for TCs: To cover nondeterminism on output (and the preemption of a test input), all outputs are always included. If the output does not conform to $\mathcal{S}$, the verdict fail is given. Nondeterminism of the LTS is resolved by considering all possible nondeterministic cases using superstates. Controllable nondeterminism is resolved by choosing either one input or quiescence. In genTC from Listing 8.2,

this choice is made nondeterministically, resulting in a nondeterministic algorithm that creates one TC. In genTS from Listing 8.3 in Subsec. 8.8.3, nondeterministic resolutions will be enumerated deterministically, resulting in a TS.

genTC$(\mathcal{S}, \ddot{s})$ nondeterministically generates one TC from $\mathcal{S}$ with the root node $\ddot{s}$. Though described differently, this algorithm corresponds to the one from [Tretmans, 2008]: genTC$(\mathcal{S}, \ddot{s})$ nondeterministically either terminates by returning pass, or assembles a TC that tries to give an input (for $i_\delta \in L_I$), or assembles a TC that only observes the SUT (for $i_\delta = \delta$). The TC is assembled from smaller TCs by **assembleTC** (TreeState, $L_U \dot\cup \{i_\delta\} \to \mathcal{TTS}(L_I, L_U, \delta)$) – again only the description differs from [Tretmans, 2008]: The TC has $\ddot{s}$ in its root and comprises each output and (one input or δ) using a case distinction over all (argument,value) pairs of $l2TC : L_U \dot\cup \{i_\delta\} \to \mathcal{TTS}(L_I, L_U, \delta)$, which appends subtrees that are TCs themselves, via recursion, sas depicted in Fig. 8.3. For $u_{k+1}, \ldots, u_{|L_U|} \notin out_{\mathcal{S}_{\tau^*}}(\ddot{s})$, the recursive calls genTC$(\mathcal{S}, \varnothing)$ yield fail.

```
proc TTS(L_I, L_U, δ) genTC(LTS(L_I, L_U, τ) S, 2^S s̈)                    1
  TreeState s := new TreeState representing s̈;                            2
  𝔹 terminate := nondet({false, true});                                   3
  L_I ∪̇ {δ} i_δ := nondet(in_{S_τ*}(s̈) ∪̇ {δ});                           4
  L_U ∪̇ {i_δ} → TTS(L_I, L_U, δ) l2TC := l ↦ genTC(S, s̈ after_{S_τ*δ} l);  5
                                                                          6
  if (s̈ == ∅) then return new TreeState fail; fi;                         7
  if (terminate) then return new TreeState pass; fi;                      8
  return assembleTC(s, l2TC);                                             9
end;                                                                      10
```

Listing 8.2: Typed nondeterministic genTC$(\mathcal{S}, \ddot{s})$

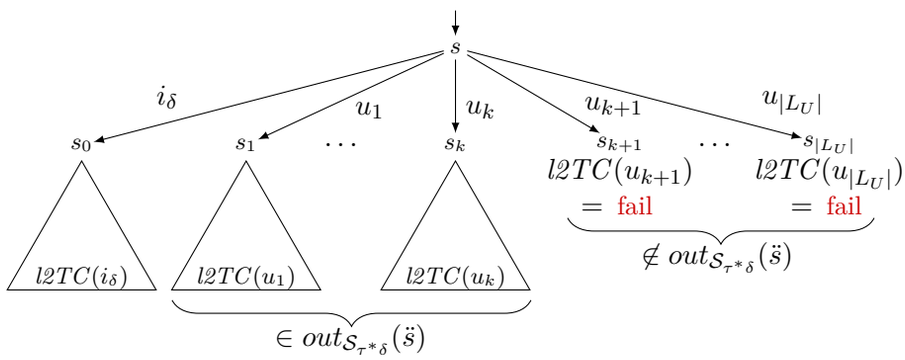


Figure 8.3.: assembleTC$(s, l2TC)$ (with L_U finite)

Worst case complexities for genTC are not sensible since it is a nondeterministic algorithm that returns a single TC, which can become arbitrarily large. The main resource consumption for each genTC call is for

- $\ddot{s}\,after_{\mathcal{S}_{\tau^*\delta}}\,l$, which is covered by the complexities of a determinization step: a worst case time complexity in $O(|S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}})$ and a worst case space complexity in $O(|S_{\to^*}| + branch_{\mathcal{S}_{det}})$ (cf. Subsec. 8.2.5);

- and assembleTC, which has a worst case time and space complexity of $O(|L_U|)$ each.

So for genTC that returns $\mathbb{T} = (S, T, L_\delta) \in \mathcal{TTS}(L_I, L_U, \delta)$, the time complexity is in $O(|S| \cdot |S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}})$, the space complexity in $O(|S| \cdot |S_{\to^*}| + |T| + branch_{\mathcal{S}_{det}})$.

Note 8.44. If $\mathcal{S}$ is infinitely branching, *l2TC* and the TC must be described implicitly, e.g., represented symbolically (cf. Subsec. 5.4.2).

For infinite branchings, the depth of a TC can be infinite (cf. Note 8.36). But since executions are always finite, the generation of finite TCs is sufficient and gen($\mathcal{S}$) terminates.

The Lemmas 8.45, 8.46 and 8.47 show properties of genTC for $\mathcal{S} = (S, \to, L) \in \mathcal{LTS}(L_I, L_U, \tau)$. They use the fact that states in TCs are instances of superstates of S (cf. l.2 of Listing 8.2). Thereafter, Theorem 8.48 proves genTC($\cdot$, $init_{\mathcal{S}}$ *after* τ) sound (using Lemma 8.45, Lemma 8.47) and exhaustive (using Lemma 8.46), with a roughly similar approach as sketched in [Tretmans, 1996].

Lemma 8.45. *Let* $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$, *with* $\mathcal{S}_{det} = (S_{det}, \overrightarrow{d}, L_{\delta_\frown})$, $\ddot{s} \in S_{det}, \mathbb{T} \in genTC(\mathcal{S}, \ddot{s})$, $\pi \in paths_{<max}(\mathbb{T})$. *Then* $\pi \in paths(\mathcal{S}_{det}, \ddot{s})$.

Proof. The proof uses induction over $|\pi|$.

For the base case $n = 0$, $\pi = (\ddot{s}) \in paths(\mathcal{S}_{det}, \ddot{s})$. For the induction step from n to $n+1$, $\pi = (\ddot{s} \xrightarrow{x} \ddot{s}') \cdot \pi'$ with $x \in L_\delta$ and $\pi' \in paths_{<max}(\mathbb{T}')$ for some $\mathbb{T}' \in$ genTC$(\mathcal{S}, \ddot{s}')$. Since π is not maximal, the recursive call genTC$(\mathcal{S}, \ddot{s}$ *after* $x)$ did not yield fail, so $\ddot{s}$ *after* $x \neq \varnothing$. Thus $\ddot{s} \xrightarrow{x} \ddot{s}' \in paths(\mathcal{S}_{det}, \ddot{s})$. By the induction hypothesis for $\ddot{s}'$ and $\mathbb{T}'$, $\pi' \in paths(\mathcal{S}_{det}, \ddot{s}')$, so $\pi \in paths(\mathcal{S}_{det}, \ddot{s})$. □

Lemma 8.46. *Let* $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$, *with* $\mathcal{S}_{det} = (S_{det}, \overrightarrow{d}, L_{\delta_\frown})$, $\ddot{s} \in S_{det}, \pi \in paths^{fin}(\mathcal{S}_{det}, \ddot{s})$. *Then* $\exists \mathbb{T} \in genTC(\mathcal{S}, \ddot{s})$ *such that* $\pi \in paths_{<max}(\mathbb{T})$.

Proof. The proof uses induction over $|\pi|$.

For the base case $n = 0$, $\pi = (\ddot{s})$. genTC$(\mathcal{S}, \ddot{s})$ can immediately choose terminate, thus return $\mathbb{T}$ that only contains pass for $\ddot{s}$, so $paths(\mathbb{T}) = \{(\ddot{s})\}$.

For the induction step from n to $n+1$, we have $\pi = (\ddot{s} \xrightarrow{x} \ddot{s}') \cdot \pi'$ with $\pi' \in paths^{fin}(\mathcal{S}_{det}, \ddot{s}')$. For $x \in in_{\mathcal{S}_{det}}(\ddot{s}) \dot\cup \{\delta\}$, genTC$(\mathcal{S}, \ddot{s})$ can take that nondeterministic choice (cf. l.4 of Listing 8.2). For $x \in out_{\mathcal{S}_{det}}(\ddot{s})$, genTC$(\mathcal{S}, \ddot{s})$ includes that output anyways (cf. l.5). Thus, there is a corresponding transition $\ddot{s} \xrightarrow{x} \ddot{s}'$ in the $\mathbb{T}$ that genTC$(\mathcal{S}, \ddot{s})$ returns (cf. Fig. 8.3). By the induction hypothesis for $\ddot{s}'$ and π', there exists a $\mathbb{T}' \in$ genTC$(\mathcal{S}, \ddot{s}')$ with $\pi' \in paths_{<max}(\mathbb{T}')$. The recursive call genTC$(\mathcal{S}, \ddot{s}')$ of genTC$(\mathcal{S}, \ddot{s})$ can choose that $\mathbb{T}'$, so there exists a $\mathbb{T} \in$ genTC$(\mathcal{S}, \ddot{s})$ with $\pi \in paths_{<max}(\mathbb{T})$. □

Lemma 8.47. *Let* $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau), \mathbb{T} \in genTC(\mathcal{S}, init_{\mathcal{S}}$ *after* $\tau), \pi' \in paths(\mathbb{T}), \pi \in paths_{max}(\mathbb{T})$ *with* $\pi = \pi' \cdot (dest(\pi') \xrightarrow{x} fail)$. *Then* $x \in (L_U \dot\cup \{\delta\}) \setminus out_{\mathcal{S}_{det}}(dest(\pi'))$.

Proof. Let $\ddot{s} := dest(\pi')$.

Assume $x \in L_I$, then $x \in in_{\mathcal{S}_{\tau^*}}(\ddot{s})$ (cf. l.4 of Listing 8.2). Thus $\ddot{s}$ $after_{\mathcal{S}_{\tau^*\delta}}$ $x \neq \varnothing$, contradicting $(\ddot{s} \xrightarrow{x} \text{fail})$. Thus $x \in L_U \dot\cup \{\delta\}$.

Assume $x \in out_{\mathcal{S}_{det}}(\ddot{s})$, then again $\ddot{s}$ $after_{\mathcal{S}_{\tau^*\delta}}$ $x \neq \varnothing$ contradicts $(\ddot{s} \xrightarrow{x} \text{fail})$. Thus $x \notin out_{\mathcal{S}_{det}}(\ddot{s})$. □

Theorem 8.48. *genTC*$(\cdot, \mathit{init}\ \mathit{after}\ \tau)$ *is sound and exhaustive.*

Proof. To prove **soundness**, let $\mathcal{S} = (S, \to, L) \in \mathcal{LTS}(L_I, L_U, \tau), \mathbb{T} \in \mathrm{genTC}(\mathcal{S}, \mathit{init}_{\mathcal{S}}\ \mathit{after}\ \tau), \mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ with $\mathbb{M}$ fails $\mathbb{T}$. Thus $\exists \pi \in \mathit{paths}_{max}(\mathbb{T} || \mathbb{M}_{\tau^*\delta}) : \mathit{dest}(\mathcal{F}_{mod}(\pi)) = \text{fail}$. Therefore, $\mathcal{S}, \mathbb{T}$ and $\mathcal{F}_{MOD}(\pi) = \pi' \cdot (\ddot{s} \xrightarrow{x} \ddot{s}')$ meet the premise of Lemma 8.47, so $x \notin \mathit{out}_{\mathcal{S}_{det}}(\ddot{s})$. Furthermore, $\mathcal{S}, \mathbb{T}$ and π' meet the premise of Lemma 8.45, so $\pi' \in \mathit{paths}(\mathcal{S}_{det}, \mathit{init}_{\mathcal{S}_{det}}\ \mathit{after}\ \tau)$. Thus $\mathit{trace}(\pi') \in \mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}})$ and $\mathit{out}_{\mathbb{M}_{\tau^*\delta}}(\mathit{init}_{\mathbb{M}}\ \mathit{after}\ \mathit{trace}(\pi')) \not\subseteq \mathit{out}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}}\ \mathit{after}\ \mathit{trace}(\pi'))$. Consequently, $\mathbb{M}\ \cancel{\mathit{ioco}}\ \mathcal{S}$.

To prove **exhaustiveness**, let $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau), \mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ with $\mathbb{M}\ \cancel{\mathit{ioco}}\ \mathcal{S}$. Therefore, there exists a suspension trace $\sigma \in \mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}})$ and $x \in L$ with $x \in \mathit{out}_{\mathbb{M}_{\tau^*\delta}}(\mathit{init}_{\mathbb{M}}\ \mathit{after}\ \sigma)$ but $x \notin \mathit{out}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}}\ \mathit{after}\ \sigma)$. Let $\pi \in \mathit{paths}^{fin}(\mathcal{S}_{det}, \mathit{init}_{\mathcal{S}_{det}}\ \mathit{after}\ \tau)$ be the unique path according to suspension trace σ (i.e., with $\sigma = (l_i)_{i \in [1,\dots,|\sigma|]}$, path $\pi = (\ddot{s}_{i-1} \xrightarrow{l_i} \ddot{s}_i)_{i \in [1,\dots,|\sigma|]}$ with $\ddot{s}_0 = \mathit{init}_{\mathcal{S}}\ \mathit{after}\ \tau$). Lemma 8.46 shows that there exists a $\mathbb{T} \in \mathrm{genTC}(\mathcal{S}, \mathit{init}_{\mathcal{S}}\ \mathit{after}\ \tau)$ such that $\pi \in \mathit{paths}_{<max}(\mathbb{T})$. Since π is not maximal, $\mathit{dest}(\pi)$ is not a verdict TreeState. Since $x \notin \mathit{out}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}}\ \mathit{after}\ \sigma)$, $\pi \cdot (\mathit{dest}(\pi) \xrightarrow{x} \text{fail}) \in \mathbb{T}$ (cf. Fig. 8.3). Since $x \in \mathit{out}_{\mathbb{M}_{\tau^*\delta}}(\mathit{init}_{\mathbb{M}}\ \mathit{after}\ \sigma), \mathbb{M}$ fails $\mathbb{T}$. □

Thus $\mathrm{genTC}(\cdot, \mathit{init}_{\mathcal{S}}\ \mathit{after}\ \tau)$ can be used as $\mathrm{gen}(\cdot)$. For brevity, Def. 8.49 extends paths of $\mathcal{S}_{det}$ with verdicts.

Definition 8.49. Let $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$. Then

- $\boldsymbol{\mathit{paths}_{\text{fail}}(\mathcal{S}_{det})} := \{\pi \cdot (\mathit{dest}(\pi) \xrightarrow{x} \text{fail}) \mid \pi \in \mathit{paths}(\mathcal{S}_{\mathrm{det}}), x \in (L_U \dot{\cup} \{\delta\}) \setminus \mathit{out}_{\mathcal{S}_{det}}(\mathit{dest}(\pi))\}$;
- $\boldsymbol{\mathit{paths}_{\mathbb{V}}(\mathcal{S}_{det})} := \begin{cases} \mathit{paths}(\mathrm{genTC}(\mathcal{S}, \mathit{init}_{\mathcal{S}}\ \mathit{after}\ \tau)) & \text{if verdicts in TCs are given explicitly (cf. Def. 8.34);} \\ \mathit{paths}(\mathcal{S}_{det}) \cup \mathit{paths}_{\text{fail}}(\mathcal{S}_{det}) & \text{if verdicts in TCs are given implicitly (cf. Subsec. 13.2.2).} \end{cases}$
- $\mathit{paths}_{\mathbb{V}}(\mathcal{S}_{det})$ can be embedded in $\mathit{paths}(\mathcal{S}_{det})$ by allowing the additional state fail in S_{det} (e.g., represented by $\varnothing$).

Note 8.50. To get more meaningful test cases, gen should avoid building TCs for traces that do not help in detecting failures. Consequently, gen should only construct TCs for the inputs that are enabled in a superstate (cf. Listing 8.2 line 4 and Listing 8.3 line 7) since the other inputs are irrelevant for *ioco* (cf. Sec. 8.5 and Sec. 9.2).

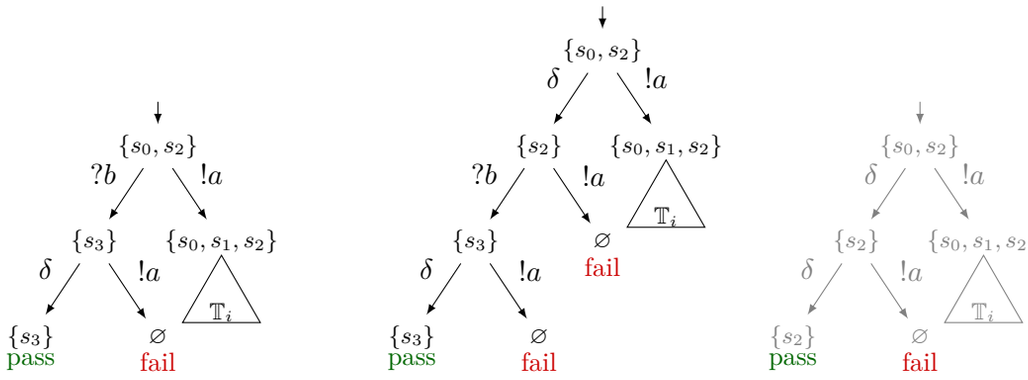
Furthermore, all longest non-maximal paths $\pi \in \mathit{paths}_{<max}(\mathbb{T})$ should have traces in $\mathit{faultable}(\mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}}))$:

- not faultable end states $\ddot{s}$ can be avoided by checking that $\mathit{out}_{\mathcal{S}_{det}}(\ddot{s}) \neq L_U \dot{\cup} \{\delta\}$;
- to handle quiescence, an additional parameter for the test case generation algorithm is required to inform the nested call whether the last transition was δ. Thus paths containing $\delta \cdot \delta$ as well as ending directly after δ can be avoided;
- instead of the additional parameter and the parameter $\ddot{s}$, the current path π can be passed as parameter (cf. Listing 12.1 and Listing 12.2).

Example 8.51. Fig. 8.4 depicts the beginning of three TCs $\mathbb{T}_1$, $\mathbb{T}_2$ and $\mathbb{T}_3$ for $\mathcal{S}$ from Subfig. 8.1a with $L_U = \{!a\}$ and $L_I = \{?b\}$. They are the templates for the set of all

nondeterministically generated TCs genTC($\mathcal{S}$), which can be constructed by combining $\mathbb{T}_1$, $\mathbb{T}_2$, $\mathbb{T}_3$ via concatenation (cf. Def. 8.35), and pruning the TCs to finite ones by nondeterministically replacing sufficiently many non-terminal states with pass.

Template $\mathbb{T}_3$ is unnecessary because its fail never occurs in an SUT due to the testing hypothesis (cf. Note 8.50). Avoiding template $\mathbb{T}_3$ results in TCs that only consist of faultable longest non-maximal paths.



(a) $\mathbb{T}_1$ **(b)** $\mathbb{T}_2$ **(c)** unnecessary $\mathbb{T}_3$

Figure 8.4.: TCs generated by genTC for $\mathcal{S}$ from Subfig. 8.1a

Although all longest non-maximal paths of all TCs are faultable, the next subsection shows that genTC still contains redundant TCs, and refines the test case generation algorithm.

8.8.3. Deterministic Test Suite Generation genTS

As Def. 8.52 and Lemma 8.53 show, genTC still contains redundant TCs (i.e., contradicting the DRY principle, [Hunt and Thomas, 1999]).

Definition 8.52. Let $\mathbb{T} = (S, \rightarrow, L) \in \mathcal{TTS}(L_I, L_U, \delta), \mathbb{T}' = (S', \rightarrow, L) \in \mathcal{TTS}(L_I, L_U, \delta)$. Then:

- $\mathbb{T}'$ **extends** $\mathbb{T} :\Leftrightarrow$ $\mathbb{T}'$ begins like $\mathbb{T}$, but may replace some pass states with some longer TCs. Formally, $\exists n \in \mathbb{N}\, \exists \mathbb{T}_0, \ldots, \mathbb{T}_n :\ \mathbb{T}' = \mathbb{T} \overset{\cdot}{\underset{i\in[1,\ldots,n]}{}} \mathbb{T}_i$.
- the extends relation inherits transitivity, reflexivity and antisymmetry from concatenation.

Lemma 8.53. *Let* $\mathbb{T}, \mathbb{T}' \in \mathcal{TTS}(L_I, L_U, \delta)$ *with* $\mathbb{T}'$ *extends* $\mathbb{T}$, $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$. *Then* $\mathbb{M}$ *fails* $\mathbb{T} \Rightarrow \mathbb{M}$ *fails* $\mathbb{T}'$.

Proof. $\mathbb{M}$ fails $\mathbb{T} \Rightarrow \exists \pi \in paths^{fin}_{max}(\mathbb{T}||\mathbb{M}_{\delta\tau^*}) : dest(\mathcal{F}_{mod}(\pi)) =$ fail. Since an extension only differs from $\mathbb{T}$ at pass states in $\mathbb{T}$, $\pi \in paths^{fin}_{max}(\mathbb{T}'||\mathbb{M}_{\delta\tau^*})$ and $dest(\mathcal{F}_{mod}(i(\pi))) =$ fail, so $\mathbb{M}$ fails $\mathbb{T}'$. □

So larger TCs detect more failures than the TCs they extend, and also make better use of unlikely paths by not terminating with pass when they occur. Thus extensions

should be preferred. But since TCs need to be finite, our next test case generation algorithm **genTS($\mathcal{S}, \ddot{s}, b$)** takes a **bound b** as additional parameter and generates a TS by resolving nondeterminism of genTC($\mathcal{S}, \ddot{s}$) in the following way (cf. Listing 8.3):

- terminate is chosen **true** exactly at depth b of a generated TC, i.e., at recursion depth b (similarly to the bounded considerations of bounded model checking, another bounded exploration for bug finding, cf. Subsec. 5.2.3 and Chapter 7);
- all other nondeterministic choice points are resolved by enumerating all choices via backtracking.

```
proc 2^{TTS(L_I,L_U,δ)} genTS(LTS(L_I,L_U,τ) S, 2^S s̈, ℕ b)                    1
  TreeState s := new TreeState representing s̈;                               2
  2^{TTS(L_I,L_U,δ)} result := ∅;                                            3
                                                                             4
  if (s̈ == ∅) then return {new TreeState fail}; fi;                          5
  if (b == 0) then return {new TreeState pass}; fi;                          6
  for each L_I ∪̇ {δ} i_δ ∈ in_{S_{τ*}}(s̈) ∪̇ {δ} do                           7
    for each L_U ∪̇ {i_δ} → TTS(L_I,L_U,δ) l2TC with                           8
          ∀l ∈ L_U ∪̇ {i_δ}: l2TC(l) ∈ genTS(S, s̈ after_{S_{τ*δ}} l, b−1) do
      result.add(assembleTC(s, l2TC));                                       9
    od;                                                                      10
  od;                                                                        11
  return result;                                                             12
end;                                                                         13
```

Listing 8.3: Typed deterministic genTS($\mathcal{S}$, $\ddot{s}$, b)

Having a bounded, deterministic algorithm, worst case complexities can be computed, but are extremely rough: Since we iterate over each $L_I \dot\cup \{\delta\}$ in each node (i.e., TreeState) and have to consider all L_U for *l2TC*, we need to consider the full computation tree of $\mathcal{S}_{det}$ even if we do not construct each TC or compute each assembleTC from scratch. The main resource consumption for each genTS call is for determinization, so the overall **worst case time complexity of genTS**($\mathcal{S}, init_{\mathcal{S}} after \tau, b$) is in $O(branch^b_{\mathcal{S}_{det}} \cdot |S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}})$, the overall **worst case space complexity** in $O(branch^b_{\mathcal{S}_{det}} \cdot (|S_{\to^*}| + |L|))$, and both are extremely rough.

Lemma 8.54 and Corollary 8.55 show that genTS($\cdot$, *init after* $\tau, \mathbb{N}$) is sound and exhaustive. Thus genTS($\cdot$, $init_{\mathcal{S}}$ *after* $\tau, \mathbb{N}$) can be used as gen($\cdot$), and genTS($\cdot$, $init_{\mathcal{S}}$ *after* τ, b) generates the **non-redundant TS** of TCs with paths pruned at depth b.

Lemma 8.54. *Let $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau), \mathbb{T} \in genTC(\mathcal{S}, init_{\mathcal{S}} after \tau)$ with $depth(\mathbb{T}) = d \in \mathbb{N}$, and $b \in \mathbb{N}$ with $b \geq d$.*

Then $\exists \mathbb{T}' \in genTS(\mathcal{S}, init_{\mathcal{S}}$ after $\tau, b) : \mathbb{T}'$ extends $\mathbb{T}$.

Proof. genTS($\mathcal{S}, init_{\mathcal{S}}$ *after* τ, b) enumerates all of genTC's nondeterministic choices on i_δ for each state in the generated trees up to depth b. Since $b \geq d$, the nondeterministic resolutions made for $\mathbb{T}$ were also made for some $\mathbb{T}' \in$ genTS($\mathcal{S}, init_{\mathcal{S}}$ *after* τ, b). Thus $\mathbb{T}'$ extends $\mathbb{T}$. □

Corollary 8.55. *genTS($\cdot$, init after $\tau, \mathbb{N}$) is sound and exhaustive.*

Proof. Proof of **soundness**: $\forall \mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau) : \text{genTS}(\mathcal{S}, \mathit{init}_{\mathcal{S}} \mathit{\ after\ } \tau, \mathbb{N}) \subseteq \text{genTC}(\mathcal{S}, \mathit{init}_{\mathcal{S}} \mathit{\ after\ } \tau)$, since for each $\mathbb{T} \in \text{genTS}(\mathcal{S}, \mathit{init}_{\mathcal{S}} \mathit{\ after\ } \tau, \mathbb{N})$, $\text{genTC}(\mathcal{S}, \mathit{init}_{\mathcal{S}} \mathit{\ after\ } \tau)$ can pick the same nondeterministic choices. Thus Theorem 8.48 shows soundness.

Proof of **exhaustiveness**: Let $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$, $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ with $\mathbb{M} \not\mathit{ioco} \mathcal{S}$. Theorem 8.48 shows that $\exists \mathbb{TT} \in \text{genTC}(\mathcal{S}, \mathit{init}_{\mathcal{S}} \mathit{\ after\ } \tau)$: $\mathbb{M}$ fails $\mathbb{T}$. Since $\text{genTC}(\mathcal{S}, \mathit{init}_{\mathcal{S}} \mathit{after} \tau)$ terminated to return $\mathbb{T}$, $\mathit{depth}(\mathbb{T}) \in \mathbb{N}$ (cf. Note 8.44). Lemma 8.54 shows that $\exists \mathbb{T}' \in \text{genTS}(\mathcal{S}, \mathit{init}_{\mathcal{S}} \mathit{\ after\ } \tau, \mathit{depth}(\mathbb{T})) : \mathbb{T}'$ extends $\mathbb{T}$. Lemma 8.53 shows that $\mathbb{M}$ fails $\mathbb{T}'$. □

8.8.4. Fairness and Coverage

Having covered test case generation algorithms, we can now investigate precise definitions for our variants of fairness (cf. Subsec. 8.1.2). As mentioned in Subsec. 8.7.1, test case have to be repeatedly executed an unknown but finite number of times, i.e., repeatable reachability is guaranteed. This is enforced by some variant of fairness. All fairness variants demand certain uncontrollable nondeterministic behavior in a superstate, independent of the path leading to that superstate. Depending on the variant, paths and superstates are from $\mathbb{M}$, $\mathbb{T}$ or $\mathcal{S}$, causing different exhaustiveness thresholds in Subsec. 8.8.5 and different heuristics in Chapter 12. The lemmas in this subsection help to understand the effects of our fairness variants. Some lemmas relate exhaustiveness for a fairness variant to (nondeterministic, static) coverage criteria of test cases, which can now be defined formally for the ioco theory, in Def. 8.56.

Definition 8.56. Let $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ with $\mathcal{S}_{det} = (S_{det}, \rightarrow_{det}, L_\delta)$, $S' \subseteq S_{det}$, transition relation $\rightarrow' \subseteq \rightarrow_{det}$, $\mathit{Straces}' \subseteq \mathit{Straces}_{\mathcal{S}_{\tau * \delta}}(\mathit{init}_{\mathcal{S}})$, TS $\ddot{\mathbb{T}} \subseteq \text{genTC}(\mathcal{S}, \mathit{init}_{\mathcal{S}} \mathit{\ after\ } \tau)$, TS $\ddot{\mathbb{T}}' \subseteq \mathcal{TTS}(L_I, L_U, \delta)$, $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ with $\mathbb{M}_{det} = (M_{det}, \rightarrow_{\mathbb{M}_{det}}, L_\delta)$, and $M' \subseteq M_{det}$.

Then we define the following **(nondeterministic, static) coverage criteria**:

- $\ddot{\mathbb{T}}'$ **covers all states of** M' $:\Leftrightarrow$ $\forall \ddot{m} \in M' \; \exists \mathbb{T} \in \ddot{\mathbb{T}}'$ with $\mathbb{T} = (T, \rightarrow_{\mathbb{T}}, L_\delta)$ $\exists t \in T : \; \mathbb{T} || \mathbb{M}_{det}$ contains state $(t, \ddot{m})$
- $\ddot{\mathbb{T}}$ **covers all states of** S' $:\Leftrightarrow$ $\forall \ddot{s} \in S' \; \exists \mathbb{T} \in \ddot{\mathbb{T}}$ with $\mathbb{T} = (T, \rightarrow_{\mathbb{T}}, L_\delta) :$ $\ddot{s} \in T$
- $\ddot{\mathbb{T}}$ **covers all transitions of** $\rightarrow'$ $:\Leftrightarrow$ $\forall (\ddot{s}, l, \ddot{s}') \in \rightarrow' \; \exists \mathbb{T} \in \ddot{\mathbb{T}}$ with $\mathbb{T} =$ $(T, \rightarrow_{\mathbb{T}}, L_\delta) : \; (\ddot{s}, l, \ddot{s}') \in \rightarrow_{\mathbb{T}}$
- $\ddot{\mathbb{T}}$ **covers all suspension traces of** $\mathit{Straces}'$ $:\Leftrightarrow$ $\forall \sigma \in \mathit{Straces}' \; \exists \mathbb{T} \in \ddot{\mathbb{T}} :$ $\sigma \in \mathit{Straces}_{\mathbb{T}}(\mathit{init}_{\mathbb{T}})$.

fairness$_{model}$, our weakest kind of fairness (cf. Lemma 8.64), demands that the SUT behaves fairly in correspondence to the model, as defined in Def. 8.57.

Definition 8.57. Let $\mathbb{S} \in \mathit{SUT}$ and $\mathbb{M} \in \mathcal{IOTS}_{LI}(L_I, L_U, \tau)$ the corresponding model according to the testing hypothesis (cf. Def. 8.4).

Then $\mathbb{S}$ has **fairness**$_{model}$ iff $\exists t \in \mathbb{R}$ with $t > 0 \; \forall \pi \in \mathit{paths}^{fin}(\mathbb{M}_{det})$:

- $\forall u \in \mathit{out}_{\mathbb{M}_{det}}(\mathit{dest}(\pi)) :$ when $\mathit{trace}(\pi)$ is executed infinitely often on $\mathbb{S}$, then $\mathbb{S}$ gives the output u infinitely often directly afterwards within t seconds;

- and $\forall l_i \in L_I$: when $trace(\pi)$ is executed infinitely often on $\mathbb{S}$ and directly afterwards l_i is offered to $\mathbb{S}$ within t seconds, then $\mathbb{S}$ infinitely often accepts l_i, i.e., does not preempt l_i by giving an output beforehand.

Lemma 8.58 connects test execution on the SUT and test execution on its model, so that Lemma 8.59, Lemma 8.61, Lemma 8.63 and their proofs only need to consider models. Finally Lemma 8.64 will show the relationship between our fairness variants.

Lemma 8.58. *Let $\mathbb{S} \in SUT$ have $fairness_{model}$, $\mathbb{M} \in \mathcal{IOTS}_{LI}(L_I, L_U, \tau)$ be a corresponding model according to the testing hypothesis, TC $\mathbb{T} \in \mathcal{TTS}(L_I, L_U, \delta)$, and $\sigma \in traces_{<max}(\mathbb{T}||\mathbb{M}_{\delta\tau^*})$ a non-maximal test run.*

When $\mathbb{T}$ is executed infinitely often on $\mathbb{S}$, then it will exhibit the test run σ infinitely often.

Proof. Since $Straces_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}} \text{ after } \tau) = Straces_{\mathbb{M}_{det}}(init_{\mathbb{M}_{det}})$ (cf. Lemma 8.18), we have $traces_{<max}(\mathbb{T}||\mathbb{M}_{\delta\tau^*}) = traces_{<max}(\mathbb{T}||\mathbb{M}_{det})$. Let $\mathbb{T} = (T, \rightarrow_{\mathbb{T}}, L_\delta)$ and $\pi \in paths_{<max}(\mathbb{T}||\mathbb{M}_{det})$ the unique path with $trace(\pi) = \sigma$, with $\pi = ((t_{i-1}, \ddot{m}_{i-1}) \xrightarrow{l_i} (t_i, \ddot{m}_i))_{i\in[1,\ldots,|\pi|]}$.

The proof uses induction over the length n of π.

For the base case $n = 0$, $\pi_{\leq n} = ((init_{\mathbb{T}}, init_{\mathbb{M}}))$, which every execution of $\mathbb{T}$ on $\mathbb{S}$ exhibits.

For the induction step from n to $n+1$, the induction hypothesis shows that infinitely often executing $\mathbb{T}$ on $\mathbb{S}$ infinitely often exhibits the test run $trace(\pi_{\leq n})$, leading to $\ddot{m}_n$. Since $\pi \in paths_{<max}(\mathbb{T}||\mathbb{M}_{det})$, $l_{n+1} \in out_{\mathbb{M}_{det}}(\ddot{m}_n) \dot{\cup} L_I$, so fairness_{model} guarantees that the next transition after $trace(\pi_{\leq n})$ will infinitely often be l_{n+1} (with an appropriate timeout t of Def. 8.57). □

Lemma 8.59. *Let $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$, $\ddot{\mathbb{T}} \subseteq genTC(\mathcal{S}, init_{\mathcal{S}} \text{ after } \tau)$ and SUT restricted by $fairness_{model}$.*

Then $\ddot{\mathbb{T}}$ is exhaustive iff $\ddot{\mathbb{T}}$ covers all paths in $faultable(Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}))$.

Proof. Let $\mathcal{F} := faultable(Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}))$. Exactly the superstates of $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ that are reachable by some $\sigma \in \mathcal{F}$ can reveal $\mathbb{M} \not\!\!ioco \mathcal{S}$, so no more than the *Straces* in $\mathcal{F}$ need to be covered by $\ddot{\mathbb{T}}$.

The testing hypothesis with fairness_{model} allows a model $\mathbb{M} \in MOD$ to become arbitrarily more complex than $\mathcal{S}$, so any $\sigma \in \mathcal{F}$ might lead to a new superstate of a $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$, which can have non-conforming output (cf. computation tree in proof of Lemma 8.27). Consequently, all *Straces* of $\mathcal{F}$ need to be covered by $\ddot{\mathbb{T}}$. □

Notes. Lemma 8.59 is similar to the combination of Theorem 4.1 and Theorem 4.2 of [Volpato and Tretmans, 2013], which uses *Rtraces* and *uioco* and is directly defined via $\mathcal{F}$ (cf. Sec. 9.1).

We can also translate fairness_{strong} from model checking (cf. Def. 5.6) to the ioco theory: Let $\mathbb{S} \in SUT$ and $\mathbb{M} = (M, \rightarrow, L_\tau) \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ the corresponding model according to the testing hypothesis. Then $\mathbb{S}$ has fairness_{strong} iff each $\ddot{m} \xrightarrow{l} \ddot{m}'$ that occurs during test execution on $\mathbb{S}$ must occur recurrently while test execution progresses (or the test execution eventually fails). Since fairness_{strong} demands that $\ddot{m}$ is successively visited through longer paths, it seems stronger than fairness_{model}, where $\ddot{m}$ may be visited infinitely often through the same finite path by repeatedly executing

some TC. But since $\mathbb{M}$ is not determined in *ioco*, fairness_{model} implies fairness_{strong}: if $\ddot{m}$ needs to be reached by some longer path to exhibit all its nondeterministic behaviors, we simply replace $\mathbb{M}$ by a larger, unwound model $\mathbb{M}'$ that demands the longer path.

fairness_{test}, the strongest kind of fairness we use in this thesis (cf. Lemma 8.64), demands that for each test case $\mathbb{T} \in \text{gen}(\mathcal{S})$, a SUT that is *ioco* $\mathcal{S}$ exhibits all its nondeterministic behaviors in $\mathbb{T}$. Since the SUT hopefully does not exhibit all paths of $\mathbb{T}$ to fail , and since $\mathcal{S}$ might contain underspecification of output (cf. Def. 8.65), not all paths in $\mathbb{T}$ eventually occur during test execution of $\mathbb{T}$ on the system. But all output that occurs after some trace leading to the superstate $\ddot{s}$ of $\mathcal{S}_{det}$ also eventually occurs after any trace in $\mathbb{T}$ that leads to $\ddot{s}$. This condition is given in Def. 8.60.

Definition 8.60. Let $\mathbb{S} \in SUT$ that meets fairness_{model}, and $\mathbb{M} \in \mathcal{IOTS}_{LI}(L_I, L_U, \tau)$ a corresponding model according to the testing hypothesis.

Then $\mathbb{S}$ has **fairness_{test}** iff $\forall \mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ with $\mathbb{M}$ *ioco* $\mathcal{S}$ $\forall \mathbb{T} \in \text{gen}(\mathcal{S})$ $\forall \pi \in paths(\mathbb{T})$ $\forall \pi' \in paths^{fin}(\mathcal{S}_{det})$ with $dest(\pi) = dest(\pi')$:

$$out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}_{\tau^*\delta}} \text{ after } trace(\pi)) = out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}_{\tau^*\delta}} \text{ after } trace(\pi')).$$

Notes. Since $\mathbb{T} \in \text{gen}(\mathcal{S})$, $paths(\mathbb{T}) = paths^{fin}(\mathbb{T})$ (cf. proof of Corollary 8.55).

Def. 8.60 could be defined purely via *SPEC* (cf. Def. 8.62 and Lemma 8.64), without $\mathbb{T}$, but is motivated coming from TCs.

Lemma 8.61. *Let SUT be restricted by* fairness_{test}. *Let* $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ *with* $\mathcal{S}_{det} = (S_{det}, \rightarrow, L_\delta)$, *and* $\ddot{\mathbb{T}} \subseteq genTC(\mathcal{S}, init_{\mathcal{S}} \text{ after } \tau)$.

Then $\ddot{\mathbb{T}}$ *is exhaustive iff* $\ddot{\mathbb{T}}$ *covers all states of* $faultable(S_{det})$.

Proof. Let $\ddot{\mathbb{T}}$ be exhaustive, $\ddot{s} \in faultable(S_{det})$, and $\mathbb{M} := (\mathcal{S}_!)_{det}$ but with $out_{\mathbb{M}}(\ddot{s})$ extended to $L_U \dot\cup \{\delta\}$ by adding transitions to e.g., $init_{\mathbb{M}}$. Thus $\mathbb{M} \not\text{ioco} \mathcal{S}$, but the only fault in $\mathbb{M}$ is in $\ddot{s}$. Since $\ddot{\mathbb{T}}$ is exhaustive, there is a $\mathbb{T} \in \ddot{\mathbb{T}}$ that visits $\ddot{s}$.

Let $\ddot{\mathbb{T}}$ cover all states of $faultable(S_{det})$, and $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ with $\mathbb{M} \not\text{ioco} \mathcal{S}$. Thus $\exists \sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}} \text{ after } \tau) : out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}} \text{ after } \sigma) \not\subseteq out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}} \text{ after } \sigma)$. Let $\ddot{s} := init_{\mathcal{S}} \text{ after } \sigma$, so $\ddot{s} \in faultable(S_{det})$. Then $\exists \mathbb{T} \in \ddot{\mathbb{T}}\, \exists \pi \in paths(\mathbb{T}) : dest(\pi) = \ddot{s}$. Because of fairness_{test}, $out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}_{\tau^*\delta}} \text{ after } trace(\pi)) = out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}_{\tau^*\delta}} \text{ after } \sigma)$, so $\mathbb{M}$ fails $\mathbb{T}$. Thus $\ddot{\mathbb{T}}$ is exhaustive. □

Lemma 8.61 shows that for fairness_{test}, $faultable(S_{det})$ coverage is sufficient for exhaustiveness. So $\mathcal{S}$ constraints the complexity of $\mathbb{M}$, similarly to the FSM-based testing theory [Lee and Yannakakis, 1996]. But fairness_{test} is weaker since the relationship between states of $\mathcal{S}$ and $\mathbb{M}$ is weaker and the bound on states of $\mathbb{M}$ may be exponentially larger compared to FSM-based testing ($2^{|S|} - 1$ vs. $|S|$, cf. Table 8.1 on page 225). But fairness_{test} is still very restrictive on the output behavior of $\mathbb{M}$, i.e., $\mathcal{S}$'s nondeterminism on output is interpreted strictly: $\mathcal{S}$ is a heavyweight specification that must make it explicit if output behavior depends on the path leading to the superstate, i.e., $\mathcal{S}$ must incorporate the condition for some nondeterministic choice of $\mathcal{S}$ to occur in $\mathbb{M}$, e.g., some exception. An exemplary condition is some cycle in $\mathcal{S}$ that needs to be traversed sufficiently often, which $\mathcal{S}$ can incorporate by integrating a counter. Demanding that such a relevant behavioral condition (e.g., the counter) should be made explicit in $\mathcal{S}$

is sensible for desired behavior. Failing behavior, however, is caused by an arbitrarily unusual circumstance not anticipated by the developers, so the fault's condition can become arbitrarily complex, i.e., arbitrarily deep in $\mathbb{M}$. So these details cannot be added to $\mathcal{S}$ because they are infeasible as well as unknown. This leads to fairness_{spec}, which restricts the demands of fairness_{test} to the behavior specified by $\mathcal{S}$. So fairness_{test} is not required for failing behavior, resulting in Def. 8.62. Since specified output is still as predictable as in fairness_{test}, fairness_{spec} is sufficiently helpful for reachability and coverage (cf. Lemmas 8.66 and 8.67).

Definition 8.62. Let $\mathbb{S} \in SUT$ that meets fairness_{model}, and $\mathbb{M} \in \mathcal{IOTS}_{LI}(L_I, L_U, \tau)$ a corresponding model according to the testing hypothesis.

Then $\mathbb{S}$ has **fairness**$_{spec}$ iff $\forall \mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ with $\mathbb{M}\, ioco \mathcal{S}$ $\forall \pi, \pi' \in paths^{fin}(\mathcal{S}_{det})$ with $dest(\pi) = dest(\pi')$:

$$out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}_{\tau^*\delta}} \text{ after } trace(\pi)) \cap out_{\mathcal{S}_{\tau^*\delta}}(dest(\pi)) =$$
$$out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}_{\tau^*\delta}} \text{ after } trace(\pi')) \cap out_{\mathcal{S}_{\tau^*\delta}}(dest(\pi')).$$

Lemma 8.63. *Let SUT be restricted by fairness_{spec}, $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$, and $\ddot{\mathbb{T}} \subseteq genTC(\mathcal{S}, init_{\mathcal{S}} \text{ after } \tau)$.*

Then $\ddot{\mathbb{T}}$ is exhaustive iff $\ddot{\mathbb{T}}$ covers all paths in $faultable(Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}))$.

Proof. For failing behavior of the SUT, fairness_{spec} only demands fairness_{model}. So Lemma 8.59 shows that $\ddot{\mathbb{T}}$ is exhaustiveness iff $\ddot{\mathbb{T}}$ covers all $faultable(Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}))$. □

Note. So for exhaustiveness, fairness_{model} and fairness_{spec} are equally strict. But exhaustiveness is a very theoretical criterion that does not capture all practical aspects. For instance, Lemma 8.66 will show that fairness_{spec} is much more helpful than fairness_{model} for reachability.

Lemma 8.64. *Let $\mathbb{S} \in SUT$. Then fairness_{test} of $\mathbb{S} \Rightarrow \text{fairness}_{spec}$ of $\mathbb{S} \Rightarrow \text{fairness}_{model}$ of $\mathbb{S}$.*

Contrarily, there exist respective $\mathbb{S} \in SUT$ that show fairness_{test} of $\mathbb{S} \not\Leftarrow \text{fairness}_{spec}$ of $\mathbb{S} \not\Leftarrow \text{fairness}_{model}$ of $\mathbb{S}$.

Proof. We show fairness_{test} of $\mathbb{S}$ implies fairness_{spec} of $\mathbb{S}$ by restricting the equation of output in Def. 8.60 to a subset of S: fairness_{test} of $\mathbb{S}$ implies $\forall \mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ $\forall \mathbb{T} \in \text{genTC}(\mathcal{S}, init_{\mathcal{S}} \text{ after } \tau)$ $\forall \pi \in paths(\mathbb{T})$ $\forall \pi' \in paths^{fin}(\mathcal{S}_{det})$ with $dest(\mathcal{F}_{mod}(\pi)) = dest(\pi')$:

$$out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}_{\tau^*\delta}} \text{ after } trace(\pi)) \cap out_{\mathcal{S}_{\tau^*\delta}}(dest(\pi')) =$$
$$out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}_{\tau^*\delta}} \text{ after } trace(\pi')) \cap out_{\mathcal{S}_{\tau^*\delta}}(dest(\pi')).$$

This condition is equivalent to fairness_{spec} of $\mathbb{S}$.

fairness_{spec} of $\mathbb{S}$ implies fairness_{model} of $\mathbb{S}$ since fairness_{model} is a premise in the definition of fairness_{spec}.

Let $\mathbb{S} \in SUT$ have fairness_{model}, $\mathcal{S} = (S, \rightarrow, L) \in \mathcal{LTS}(L_I, L_U, \tau)$ with $L_U = \{u_1, u_2\}$, $out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}} \text{ after } \tau) = \{u_1\}$ and cycle $init_{\mathcal{S}} \xrightarrow{\pi} init_{\mathcal{S}} \in paths(\mathcal{S})$.

- If $\mathbb{S}$ outputs u_1 in $init_{\mathbb{S}}$ only after the Strace $trace(\pi)^{|S|+1}$, but not after the Straces $trace(\pi)^i$ for $i < |S| + 1$, then $\mathbb{S}$ cannot have fairness_{spec} and be $ioco$ $\mathcal{S}$.

- If $\mathbb{S}$ wrongly outputs u_2 in $init_{\mathbb{S}}$ after the trace $trace(\pi)^{|S|+1}$, but behaves *ioco* $\mathcal{S}$ for all other Straces, then $\mathbb{S}$ has fairness_{spec} but cannot have fairness_{test}.

□

The definitions of our fairness variants are sufficiently flexible that we can additionally allow or forbid underspecification of output, which means that more output is given in $\mathcal{S}$ than ever occurs in $\mathbb{M}$, as defined in Def. 8.65. Sec. 9.2 and Sec. 9.3 describe underspecification more generally. Underspecification of output is useful e.g., for refinement and if some anticipated exceptions never occur in the SUT (cf. Sec. 9.3 and Subsec. 14.3.8). But with underspec_U, the specification $\mathcal{S}$ can be unintuitive, and testability becomes difficult as some superstates and traces of $\mathcal{S}$ may never occur (cf. Lemma 8.66).

Definition 8.65. Let $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ and $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ with $\mathbb{M}$ *ioco* $\mathcal{S}$.

Then $\mathcal{S}$ has **underspecification of output** for $\mathbb{M}$ (**underspec**$_U$ for short) $:\Leftrightarrow$ $\exists \ddot{s} \in \mathcal{S}_{det}\ \exists u \in out_{\mathcal{S}_{\tau^*\delta}}(\ddot{s})\ \forall \sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}\ after\ \tau)$ with $\ddot{s} = init_{\mathcal{S}}\ after\ \sigma$: $u \notin out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}}\ after\ \sigma)$.

Lemma 8.66. *Let $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau), \mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ and SUT restricted to fairness_{spec} or fairness_{test}, with forbidden underspec_U.*

Then $\mathbb{M}$ ioco $\mathcal{S} \Rightarrow Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}) \subseteq Straces_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}})$.

Proof. Let $\mathbb{M} ioco \mathcal{S}$ and $\pi = (\ddot{s}_{i-1} \xrightarrow{l_i} \ddot{s}_i)_{i\in[1,\ldots,|\pi|]} \in paths^{fin}(\mathcal{S}_{det})$ with Strace $trace(\pi) = \sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$.

The proof uses induction over the length n of π: For the base case $n = 0$, $\pi_{\leq n} = (init_{\mathcal{S}_{det}})$ and $trace(\pi_{\leq n}) = \epsilon \in Straces_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}})$. For the induction step from n to $n+1$, the induction hypothesis shows that $trace(\pi_{\leq n}) \in Straces_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}})$. Since underspec_U is forbidden, there is an Strace $\sigma' \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$ with $\ddot{s}_n = init_{\mathcal{S}}\, after \sigma'$ and $l_{n+1} \in out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}}\, after \sigma')$. Since we have $\mathbb{M} ioco \mathcal{S}$ and (fairness_{spec} or fairness_{test}), we also have $l_{n+1} \in out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}}\ after\ trace(\pi_{\leq n})$, so $\sigma \in Straces_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}})$. □

Lemma 8.67. *Let $\mathcal{S} \in SPEC$, $\sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$, $b \in \mathbb{N}_{>0}$, $\mathbb{T} \in genTS(\mathcal{S}, init_{\mathcal{S}} after \sigma, b)$, $\mathbb{M} \in MOD$ with $\mathbb{M}$ ioco $\mathcal{S}$ and SUT restricted to fairness_{spec} (or fairness_{test}), with forbidden underspec_U.*

Then recurrent execution of $\mathbb{T}$ on $\mathbb{M}$ in $init_{\mathbb{M}}$ after σ exhibits all $traces_{<max}(\mathbb{T})$.

Proof. Lemma 8.66 shows that if $\mathbb{M} ioco \mathcal{S}$, then $traces_{<max}(\mathbb{T}) \subseteq Straces_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}} after \sigma)$ (so $traces_{<max}(\mathbb{T}) = traces_{<max}(\mathbb{T}||\mathbb{M}_{\tau^*\delta})$). Lemma 8.64 shows that fairness_{model} holds. Therefore, Lemma 8.58 (for $\mathbb{M}'$ equal $\mathbb{M}$ but with $init_{\mathbb{M}'} = init_{\mathbb{M}}$ *after* σ) shows that recurrent execution of $\mathbb{T}$ on $\mathbb{M}$ in $init_{\mathbb{M}}$ *after* σ exhibits all $traces_{<max}(\mathbb{T})$. □

Lemma 8.66 and Def. 8.68 help proof Lemma 8.69, which is used to transform a sound and exhaustive $\text{gen}_{exec}(\cdot,\cdot)$ into a sound and exhaustive $\text{gen}(\cdot)$ via $\mathbb{S}_{sim\,\mathcal{S}}$.

Definition 8.68. Let $\mathcal{S} \in SPEC$.

Then $\mathbb{S}_{\boldsymbol{sim\ \mathcal{S}}}$ is any **simulated SUT** that conforms to $\mathcal{S}$ and meets fairness_{spec} and forbids underspec_U.

Lemma 8.69. *Let $gen_{exec}(\cdot,\cdot)$ be a test case generation algorithm according to Listing 8.1.*

Then $gen_{exec}(\cdot,\cdot)$ is sound (resp. exhaustive) iff $gen : \mathcal{S} \mapsto gen_{exec}(\mathcal{S}, \mathbb{S}_{sim\,\mathcal{S}})$ is sound (resp. exhaustive).

Proof. Let $\mathcal{S}' \in SPEC$, $\mathbb{S} \in SUT$, $\mathbb{M} \in \mathcal{IOTS}_{LI}(L_I, L_U, \tau)$ a corresponding model according to the testing hypothesis, and $\mathbb{M}_{sim\,\mathcal{S}'} \in \mathcal{IOTS}_{LI}(L_I, L_U, \tau)$ a model corresponding to $\mathbb{S}_{sim\,\mathcal{S}'}$ according to the testing hypothesis.

If $\text{gen}_{exec}(\cdot,\cdot)$ is sound and $\mathbb{M}$ fails $\text{gen}_{exec}(\mathcal{S}', \mathbb{S}_{sim\,\mathcal{S}'})$, then $\exists \mathbb{T} \in \text{gen}_{exec}(\mathcal{S}', \mathbb{S}_{sim\,\mathcal{S}'}) \exists \pi \in paths^{fin}_{max}(\mathbb{T}||\mathbb{M}_{\tau^*\delta}) : dest(\mathcal{F}_{mod}(\pi)) = \text{fail}$. Let $\sigma = trace(\pi)$, so $\sigma = \sigma_{\leq|\sigma|-1} \cdot u$ with $u \in L_U$. Since $\text{gen}_{exec}(\cdot,\cdot)$ is sound, $u \notin init_{\mathcal{S}'}\ after\ \sigma_{\leq|\sigma|-1}$, so $\mathbb{M} \not\!ioco \mathcal{S}'$. Thus $\mathcal{S} \mapsto \text{gen}_{exec}(\mathcal{S}, \mathbb{S}_{sim\,\mathcal{S}})$ is sound.

If $\mathcal{S} \mapsto \text{gen}_{exec}(\mathcal{S}, \mathbb{S}_{sim\,\mathcal{S}})$ is sound, then $\text{gen}_{exec}(\mathcal{S}', \mathbb{S}) \subseteq \text{gen}_{exec}(\mathcal{S}', \mathbb{S}_{sim\,\mathcal{S}'})$ is sound.

Let $\text{gen}_{exec}(\cdot,\cdot)$ be exhaustive. Since $\mathbb{M}_{sim\,\mathcal{S}'}\ ioco \mathcal{S}'$ and meets fairness_{spec} with forbidden underspec_U, Lemma 8.66 shows $Straces_{\mathcal{S}'_{\tau^*\delta}}(init_{\mathcal{S}'}) \subseteq Straces_{\mathbb{M}_{sim\,\mathcal{S}'\,\tau^*\delta}}(init_{\mathbb{M}_{sim\,\mathcal{S}'}})$. Since $\text{gen}_{exec}(\cdot,\cdot)$ is exhaustive and $\mathbb{S}_{sim\,\mathcal{S}'}$ is treated as black-box by $\text{gen}_{exec}(\cdot,\cdot)$, $\text{gen}_{exec}(\mathcal{S}', \mathbb{S}_{sim\,\mathcal{S}'})$ covers all $faultable(Straces_{\mathcal{S}'_{\tau^*\delta}}(init_{\mathcal{S}'}))$ and is hence exhaustive (cf. Lemma 8.59).

If $\mathcal{S} \mapsto \text{gen}_{exec}(\mathcal{S}, \mathbb{S}_{sim\,\mathcal{S}})$ is exhaustive, then $\text{gen}_{exec}(\mathcal{S}', \mathbb{S}_{sim\,\mathcal{S}'})$ is exhaustive for $\mathcal{S}'$. If $\mathbb{M} \not\!ioco \mathcal{S}'$, then $\exists \mathbb{T} \in \text{gen}_{exec}(\mathcal{S}', \mathbb{S}_{sim\,\mathcal{S}'})\ \exists \pi \in paths^{fin}_{max}(\mathbb{T}||\mathbb{M}_{\tau^*\delta}) : dest(\mathcal{F}_{mod}(\pi)) = \text{fail}$. Since $\pi \in paths^{fin}(\mathbb{M}_{\tau^*\delta})$, $\exists \mathbb{T}_2 \in \text{gen}_{exec}(\mathcal{S}', \mathbb{S})$ with $\pi \in paths^{fin}_{max}(\mathbb{T}_2||\mathbb{M}_{\tau^*\delta})$, so $\mathbb{M}$ fails $\mathbb{T}_2$. Thus $\text{gen}_{exec}(\cdot,\cdot)$ is exhaustive. □

Notes. The fairness described in the standard literature [Tretmans, 2008] states that "the SUT shows all its possible nondeterministic behaviors with the test case by re-execution of the test case". Depending on how "all its possible nondeterministic behaviors" is interpreted, this can have four different meanings: either fairness_{test} or fairness_{spec}, either with underspec_U allowed or not.

fairness_{model} only restricts SUT, but not MOD, so it is an abstraction from SUT to MOD. Contrarily, fairness_{spec}, fairness_{test} and forbidden underspec_U also restrict MOD, and have even been formulated as relationship between $\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)$ and $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ with the help of fairness_{model}. Since the testing hypothesis restricts MOD by restricting SUT, and all fairness restrictions are related to fairness_{model}, we included them all in the testing hypothesis.

All these fairness criteria allow nondeterminism and underspecification of output and are hence much less restrictive than most FSM-based testing [Lee and Yannakakis, 1996]. fairness_{test} is our only criterion that restricts the number of states the SUT may have (cf. Lemma 8.61 and motivation for fairness_{spec}), like most FSM-based testing [Lee and Yannakakis, 1996]. An example of a stricter fairness criterion that still allows nondeterminism but restraints it, is the constraint $\text{fairness}_{\mathbb{S}\approx\mathcal{S}}$ in Def. 8.70. It is motivated by situations where we know that $\mathcal{S}$ reflects the internal structure of $\mathbb{S}$ (e.g., because $\mathcal{S}$ was used as blueprint when developing $\mathbb{S}$) and that each state $s \in S$ can be reached in isolation in $\mathcal{S}$. Lemma 8.71 shows that $\text{fairness}_{\mathbb{S}\approx\mathcal{S}}$ does not allow nondeterminism of the LTS to cause any additional faults and hence covering the states of $faultable(S)$ in isolation is sufficient for exhaustiveness. Because it is so strict, we will never demand $\text{fairness}_{\mathbb{S}\approx\mathcal{S}}$ in this thesis. But coverage of $faultable(S)$ in isolation can still be used as simple approximation to coverage of $faultable(S_{det})$, which is still unfeasible for large $\mathcal{S}$.

This approximation is good when we expect nondeterminism of the LTS to cause only few faults.

Definition 8.70. Let $\mathbb{S} \in SUT$ that meets fairness_{test}, and $\mathbb{M} \in \mathcal{IOTS}_{LI}(L_I, L_U, \tau)$ a corresponding model according to the testing hypothesis.
The **states of** $faultable(S)$ **in isolation** is defined as the set $\boldsymbol{S_{isol}} := \{\{s\} | s \in faultable(S)\}$.
Then $\mathbb{S}$ has $\textbf{fairness}_{\mathbb{S} \approx \mathcal{S}}$ iff

- $\forall \ddot{s} \in S_{isol}\ \exists \sigma_{\ddot{s}} \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}) : init_{\mathcal{S}}\ after\ \sigma_{\ddot{s}} = \ddot{s}$
  (i.e., $S_{isol} \subseteq S_{det}$);
- and $\forall \sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}) : out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}}\ after\ \sigma) \subseteq \bigcup_{s \in faultable(init_{\mathcal{S}}\ after\ \sigma)} out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}}\ after\ \sigma_{\{s\}})$

Lemma 8.71. *Let SUT be restricted by* $fairness_{\mathbb{S} \approx \mathcal{S}}$, $\mathcal{S} = (S, \rightarrow, L) \in \mathcal{LTS}(L_I, L_U, \tau)$, *and* $\ddot{\mathbb{T}} \subseteq genTC(\mathcal{S}, init_{\mathcal{S}}\ after\ \tau)$.

Then $\ddot{\mathbb{T}}$ *is exhaustive iff* $\ddot{\mathbb{T}}$ *covers all states of* S_{isol}.

Proof. This proof is similar to the one of Lemma 8.61:

Let $\ddot{\mathbb{T}}$ be exhaustive, $\ddot{s} \in S_{isol}$, and $\mathbb{M} := (\mathcal{S}_{\dagger})_{det}$ but with $out_{\mathbb{M}}(\ddot{s})$ extended to $L_U \dot{\cup} \{\delta\}$ by adding transitions to e.g., $init_{\mathbb{M}}$. Due to $\text{fairness}_{\mathbb{S} \approx \mathcal{S}}$, $\mathbb{M}\ \cancel{ioco}\ \mathcal{S}$, but the only fault in $\mathbb{M}$ is in $\ddot{s}$. Since $\ddot{\mathbb{T}}$ is exhaustive, there is a $\mathbb{T} \in \ddot{\mathbb{T}}$ that visits $\ddot{s}$.

Let $\ddot{\mathbb{T}}$ cover all states of S_{isol}, and $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ with $\mathbb{M}\ \cancel{ioco}\ \mathcal{S}$. Thus $\exists \sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}\ after\ \tau)\ \exists u \in out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}}\ after\ \sigma) \setminus out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}\ after\ \sigma)$. Assuming that $\mathbb{M}$ passes $\ddot{\mathbb{T}}$, we have due to $\text{fairness}_{\mathbb{S} \approx \mathcal{S}}$ that $u \in out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}}\ after\ \sigma) \subseteq \bigcup_{s \in faultable(init_{\mathcal{S}}\ after\ \sigma)} out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}} after \sigma_{\{s\}}) \subseteq \bigcup_{s \in faultable(init_{\mathcal{S}}\ after\ \sigma)} out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}} after \sigma_{\{s\}})$ $= out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}\ after\ \sigma)$, a contradiction. Thus $\mathbb{M}$ must fail $\ddot{\mathbb{T}}$, so $\ddot{\mathbb{T}}$ is exhaustive. □

8.8.5. Exhaustiveness Threshold

To investigate how large the bound b should be chosen, we compare genTS to BMC: For BMC, a completeness threshold can be computed for each $\mathcal{S}$ and property F to be checked, such that BMC is complete (cf. Subsec. 5.2.3). To transfer this approach to *ioco*, we define exhaustiveness thresholds in Def. 8.72. Lemma 8.73 and Lemma 8.74 give approximations for exhaustiveness thresholds for our fairness criteria (cf. Table 8.1), and hence show whether $\mathcal{ET} \in \mathbb{N}$ exists for the given situation.

Definition 8.72. Let $\mathcal{S} \in SPEC$ and $\mathbb{S} \in SUT$.

The **exhaustiveness threshold for** $\mathcal{S}$ ($\boldsymbol{\mathcal{ET}}$) is a value $b \in \mathbb{N}$ such that $\text{genTS}(\mathcal{S}, init_{\mathcal{S}}\ after\ \tau, b)$ is exhaustive.

Lemma 8.73. *For* $fairness_{test}$ *and* $\mathcal{S} = (S, \rightarrow, L) \in \mathcal{LTS}(L_I, L_U, \tau)$, *an exhaustiveness threshold* $\mathcal{ET} \leq 2^{|S|} - 1$ *exists.*

Proof. Let $\mathcal{S}_{det} = (S_{det}, \rightarrow, L_\delta)$, and $\ddot{\mathbb{T}} \subseteq \text{genTC}(\mathcal{S}, init_{\mathcal{S}}\ after\ \tau)$.

For fairness_{test}, Lemma 8.61 shows that $\ddot{\mathbb{T}}$ is exhaustive iff it covers all states of $faultable(S_{det})$. A superstate $\ddot{s} \in S_{det}$ is reachable within at most $2^{|S|} - 2$ transitions since $|S_{det}| \lneq 2^{|S|}$. Therefore, a TC $\mathbb{T} \in \text{genTS}(\mathcal{S}, init_{\mathcal{S}}\ after\ \tau, 2^{|S|} - 1)$ also reaches $\ddot{s}$ and can make one final step to reveal if $\mathbb{M}$ is not conform to $\mathcal{S}$ in $\ddot{s}$. □

Lemma 8.74. *For* fairness_{spec} *and* fairness_{model} *and* $\mathcal{S} = (S, \to, L) \in \mathcal{LTS}(L_I, L_U, \tau)$,

- *an exhaustiveness threshold* $\mathcal{ET} \leq 2^{|S|+1} - 4$ *exists if* $\mathcal{S}$ *is finite and its only cycles are* $\delta_{\circlearrowleft}$*;*
- $\mathcal{ET} = \omega$ *otherwise.*

Proof. Let $\mathcal{S}_{det} = (S_{det}, \to, L_\delta)$, and $\ddot{\mathbb{T}} \subseteq \text{genTC}(\mathcal{S}, \mathit{init}_{\mathcal{S}}\ \mathit{after}\ \tau)$.

For fairness_{spec}, respectively fairness_{model}, Lemma 8.63, respectively Lemma 8.59, shows that $\mathbb{T}$ is exhaustive iff it covers all $\mathit{faultable}(\mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}}))$.

- If $\mathcal{S}$ is finite and its only cycles are $\delta_{\circlearrowleft}$, then each *Strace* $\sigma \in \mathit{faultable}(\mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}}))$ has a length of at most $2^{|S|+1} - 4$ since σ traverses each $\delta_{\circlearrowleft}$ at most once, contains no $\delta \cdot \delta$, does not end with δ, and $|S_{det}| \lneq 2^{|S|}$, so it may contain $2^{|S|} - 2$ states twice and its final state once.
- If $\mathcal{S}$ is not finite or contains a cycle other than $\delta_{\circlearrowleft}$, $\mathit{faultable}(\mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}}))$ contains an infinite *Strace*.

□

Note. Usually, $\mathcal{S}$ contains other cycles besides $\delta_{\circlearrowleft}$. Traversing cycles a bounded number of times is not sufficient since $\mathbb{M}$ can become arbitrarily complex (cf. Lemma 8.59), which is the cause for demanding full $\mathit{faultable}(\mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}}))$ coverage. But if we consider only SUTs that have a corresponding $\mathbb{M}$ of of maximal size $k \in \mathbb{N}$, then there exists an exhaustiveness threshold $\mathcal{ET} \leq (2^{|S|} - 1) \cdot (2^k - 1)$: executing $\mathbb{T} \in \text{genTC}(\mathcal{S}, \mathit{init}_{\mathcal{S}}\ \mathit{after}\ \tau)$ corresponds to the synchronous product of $\mathbb{T}$ and $\mathbb{M}$ (cf. Def. 8.38), so there are at most $(2^{|S|} - 1) \cdot (2^k - 1)$ different states $(s, m) \in \mathbb{T}||\mathbb{M}$. This is an upper estimate for an exhaustiveness threshold $\mathcal{ET}$ since no state of $\mathbb{T}||\mathbb{M}$ needs to be revisited on a test execution path. Ignoring certain kind of hardware errors that are usually outside the scope of a software tester (see epistemological frame problem in Subsec. 8.1.1), it might be possible to estimate k by considering the available resources or by static analysis of $\mathbb{S}$'s source code (or intermediate representation or binary code).

$\ddot{\mathbb{T}} = \text{genTS}(\mathcal{S}, \mathit{init}_{\mathcal{S}}\ \mathit{after}\ \tau, \mathcal{ET})$ is a reduced exhaustive test suite: it avoids generating TCs that are larger than an exhaustiveness threshold or contain longest non-maximal paths that are not faultable. But $\ddot{\mathbb{T}}$ might still contain some unnecessary TCs: If multiple TCs are identical within some bound $b < \mathcal{ET}$ and no extension of them cover new elements according to the appropriate coverage criterion beyond b, one of them would be sufficient in $\ddot{\mathbb{T}}$. Considering these TCs during traversal of $\mathcal{S}$ cannot be avoided since all of $\mathcal{S}$ needs to be explored to detect whether uncovered elements are reachable from the current superstate. But the unnecessary TCs can be discarded if they have not already been executed on-the-fly (or at least re-execution can be avoided). Since such TCs become less and less meaningful during test case generation, our guidance heuristics will avoid them in many cases (cf. Chapter 12).

Instead of approximating an exhaustiveness threshold a priori, if one exists, gen can do a **bound check** lazily during test case generation (similarly to a BMC tool with a bound check, cf. Subsec. 5.2.3 and Chapter 7): genTS measures the coverage criterion necessary for exhaustiveness of the given fairness criterion (cf. Lemmas 8.73 and 8.74) on-the-fly and continues to extend the TCs generated so far until the coverage, or a certain level thereof, is achieved. So besides full exhaustiveness, this coverage criterion can also be used as exit criterion, adequacy metric, guidance heuristics, or for test suite

reduction (cf. Sec. 2.5). Reporting the current bound b, or the uncovered coverage tasks, can also be helpful for the test engineer, for instance to inform him about the progress or what is left to achieve.

For these uses, we will measure coverage levels and the bound b on-the-fly during traversal as well as during test execution (cf. Chapter 11). The measurements help adapt the bound b and decide when to stop generating further TCs and when to stop executing TCs (cf. Chapter 12).

8.9. Conclusion

8.9.1. Summary

Fig. 8.5 gives an overview of the ioco theory and the main artifacts involved (used in Fig. 15.1 on page 377 to position this thesis). Cor. 8.75 summarizes why conformance by *ioco* is equivalent to passing the test suite $\text{gen}(\mathcal{S})$. For $\text{gen}(\mathcal{S}) = \text{genTS}(\mathcal{S}, \mathit{init}_{\mathcal{S}}\, \mathit{after}\, \tau, b)$, Table 8.1 gives upper estimates for the exhaustiveness bound $\mathcal{ET}$ depending on the kind of fairness.

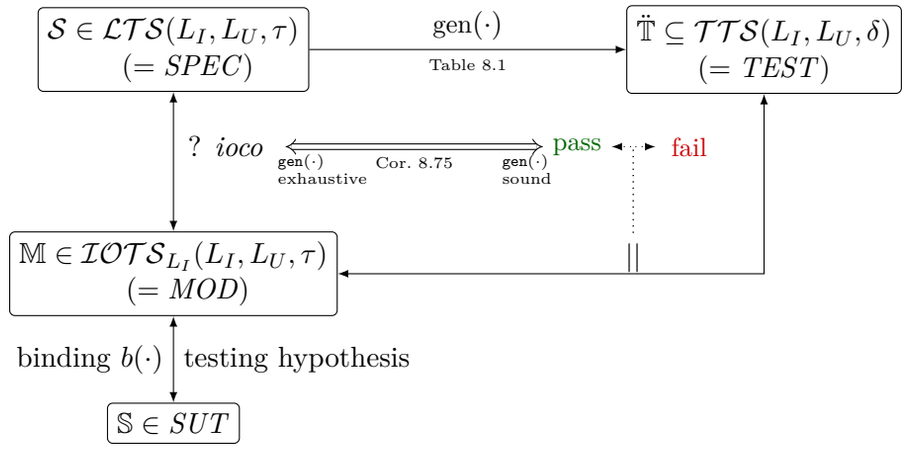


Figure 8.5.: Overview of the ioco theory and its artifacts

Corollary 8.75. *Let $\mathcal{S} \in SPEC$, $\mathbb{S} \in SUT$, and $\mathbb{M} \in MOD$ a corresponding model according to the testing hypothesis. Then:*

$$\begin{array}{ll} & \textit{executing } b(\textit{gen}(\mathcal{S})) \textit{ on } \mathbb{S} \textit{ always gives verdict } \textit{pass} \\ \overset{\textit{testing hypothesis}}{\Longleftrightarrow} & \mathbb{M} \textit{ passes } \textit{gen}(\mathcal{S}) \\ \overset{\textit{gen sound \& complete}}{\Longleftrightarrow} & \mathbb{M} \textit{ ioco } \mathcal{S} \\ \overset{\textit{testing hypothesis}}{\Longleftrightarrow} & \mathbb{S} \textit{ conforms to } \mathcal{S} \end{array}$$

Table 8.1.: Upper estimates for $\mathcal{ET}$ depending on the fairness

kind of fairness	upper estimate for $\mathcal{ET}$
fairness_{test}	$2^{\lvert S\rvert} - 1$
fairness_{spec} or fairness_{model}, with $\lvert S\rvert$ finite and only $\delta_{\circlearrowleft}$ cycles	$2^{\lvert S\rvert+1} - 4$
fairness_{spec} or fairness_{model}, with $\mathbb{M}$ having at most k states	$(2^{\lvert S\rvert} - 1) \cdot (2^k - 1)$
otherwise	no $\mathcal{ET}$, i.e., all $b \in \mathbb{N}$ must be considered

8.9.2. Contributions

There are many publications about the ioco theory; this thesis adopted many concepts, mainly from [Tretmans, 2008; Frantzen, 2016]. Nonetheless, this chapter contributed new concepts:

- many articles about the ioco theory mention the testing hypothesis, but only briefly. This chapter thoroughly investigated the testing hypothesis:
  - it showed implications and motivated decisions in the ioco theory;
  - it covered various abstractions, new fairness constraints and concepts from other fields (e.g., input refusal for input-enabledness and the reliable reset capability);
  - it showed the connection between the formal, abstract ioco theory and practical testing of the SUT, and gave advice how the test adapter should be implemented to meet the testing hypothesis;
- Subsec. 8.2.1 differentiated the semantics of internal transitions τ along two dimensions: whether τ is treated similarly to output, and whether τ transitions require time. The dimensions influence how quiescence and τ-cycles are handled;
- Subsec. 8.2.5 gave a taxonomy of the kinds of nondeterminism involved in the ioco theory, and how to resolve them for TCs, their generation and their execution;
- using our generalized formalisms from Chapter 3 led to more flexible or concise definitions for test cases, their generation and their execution, with interchangeable kinds of LTSs with I/O for *SPEC* and *MOD* without changing the ioco theory;
- besides the classical nondeterministic test case generation algorithm, a new deterministic algorithm, genTS, was given. Both are described in detailed pseudocode. genTS transfers concepts of SBMC to the ioco theory, especially exhaustiveness thresholds (cf. Table 8.1);
- redundancy of test suites was reduced by the bounded deterministic test case generation algorithm: for $\text{genTS}(\mathcal{S}, \mathit{init}_{\mathcal{S}}\mathit{after}\tau, \mathcal{ET})$, no generated TC is larger than the approximated exhaustiveness threshold $\mathcal{ET}$ or contains longest non-maximal paths that are not faultable. The approximations use coverage criteria depending on the given kind of fairness.

8.9.3. Future

Possible future work includes:

- implement the differentiation of $\tau_{\not y}$ vs. τ_t in $\mathcal{S}$, and the combination of test case generation algorithms with livelock detection, preferably $\mathrm{DFS}_{\mathrm{FIFO}}$ (cf. Subsec. 8.2.2 and Chapter 6);
- integrate special kinds of action refinement into our framework (cf. Subsec. 8.7.2), especially with input refinement also covering determined output, and output contraction also covering determined input (cf. Note 8.41);
- investigate possible fault-tolerant approaches and failure recoveries to continue test execution after reaching a failure state in the SUT (cf. Note 8.36);
- improving the approximations for the exhaustiveness thresholds, which might lead to theoretical insights of exhaustiveness and its complexity. But this is less relevant for our application of (lazy) OTF MBT with sets of test objectives (e.g., coverage criteria) as exit criteria (cf. Chapter 11);
- parallelize our algorithms (cf. Listings 8.2 and 8.3): because a TC unwinds $\mathcal{S}$ into a tree, work distribution can easily be implemented by state space partitioning (cf. Subsec. 5.5.3) of the TC, e.g., using one worker thread per outgoing transition as long as free worker threads are available. Besides speedup, parallelization also has the advantage of supporting real-time behavior (cf. Subsec. 11.6.3).

9. Variants of ioco

There are many subsets of $MOD \times SPEC$ besides *ioco* that are useful implementation relations. Many have evolved from *ioco*, others were predecessors of *ioco* or evolved independently [Tretmans, 2008]. This chapter introduces those which are relevant for this thesis.

Roadmap. Sec. 9.1 generalizes the *ioco* relation as basis for variants of *ioco*, and gives historical derivates. Sec. 9.2 investigates underspecification and the variant *uioco*, which handles underspecification of input differently than *ioco*. Sec. 9.3 introduces the relation *refines*, a generalization of *ioco* for refinement in software engineering. Sec. 9.4 describes how *ioco* can be lifted from LTSs to STSs and symbolic *ioco*.

9.1. Generalized *ioco* Relation and Derivates

The *ioco* relation (cf. Def. 8.30) can be generalized by not determining $\mathcal{F}$, the set of *Straces* for which the input and output behavior of the model $\mathcal{M}$ must conform to those of the specification $\mathcal{S}$. This results in Def. 9.1, which is the basis for many variations of the *ioco* relation. Table 9.1 lists several variants of *ioco* that can be described as $ioco_{\mathcal{F}}$ by a specific $\mathcal{F}$.

Definition 9.1. Let model $\mathcal{M} \in MOD$, specification $\mathcal{S} \in SPEC$, and $\mathcal{F} \subseteq L_\delta^*$. Then $\mathbb{M}$ $\boldsymbol{ioco_{\mathcal{F}}}$ $\mathcal{S} :\Leftrightarrow \forall \sigma \in \mathcal{F}:$

$$out_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}} \; after_{\mathbb{M}_{\tau^*\delta}} \; \sigma) \subseteq out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}} \; after_{\mathcal{S}_{\tau^*\delta}} \; \sigma)$$

Table 9.1.: Variants of *ioco* defined via $ioco_{\mathcal{F}}$ and $\mathcal{F}$

variant	$\mathcal{F}$	description
$\leq_{ior}$	L_δ^*	I/O refusal relation (*Straces* preorder)
$\leq_{iot}$	L^*	I/O testing relation ($traces^{fin}$ preorder)
ioconf	$traces^{fin}(\mathcal{S}_{\tau^*}, init_{\mathcal{S}_{\tau^*}})$	I/O *conf* relation
uioco	*Utraces* (see Def. 9.3)	underspecified *ioco*

Def. 9.2 generalizes the definition of soundness and completeness; all implementation relations c listed in Table 9.1 have variants $\text{gen}_c(\cdot)$ of $\text{gen}(\cdot)$ as sound and exhaustive test case generation algorithm [Tretmans, 1996; van der Bijl and Peureux, 2004; Tretmans, 2008; Frantzen, 2016]; $\leq_{ior}$, $\leq_{iot}$ and *ioconf* give insights into implementation relations and their history, *uioco* will be investigated in the next section.

Definition 9.2. Let implementation relation $c \subseteq MOD \times SPEC$.

Then soundness and exhaustiveness for c of a TC, TS, and test case generation algorithm $\mathbf{gen}_c(\cdot)$ are defined as in Def. 8.43, but with *ioco* replaced by c.

9.2. Underspecification and *uioco*

$\mathcal{S} \in SPEC$ is **underspecified** (also called **partially specified**) iff $\mathcal{S}$ is not input-enabled or has underspec_U (cf. Def. 8.65). More precisely, **underspecification of input** is a generalization by omitting inputs in $\mathcal{S}$, i.e., ignoring certain situations (like hazards): the unspecified inputs $i \in L_I \setminus in_{\mathcal{S}_{\tau^*}}(s)$ in state $s \in S$ do not pose any conformance conditions for *ioco*, i.e., the model $\mathbb{M} \in MOD$ may behave arbitrarily. Conversely, **underspecification of output** is a generalization by allowing many nondeterministic choices on output, i.e., giving more output in $\mathcal{S}$ than occurs in the model $\mathbb{M}$ [Fraser et al., 2009]: the additional outputs $u \in L_U$ in state s result in more relaxed conformance conditions for *ioco* in s. Sec. 9.3 will generalize underspecification in $\mathcal{S}$ by relating it to another $\mathcal{S}' \in SPEC$, not only to $\mathbb{M} \in MOD$.

Nondeterminism of the LTS causes real superstates $\ddot{s}$. Consequently, input i of an underspecified $\mathcal{S}$ with nondeterminism of the LTS can be underspecified in some, but not all $s \in \ddot{s}$, i.e., $\exists s_1, s_2 \in \ddot{s} : i \in in_{\mathcal{S}_{\tau^*}}(s_1)$ and $i \notin in_{\mathcal{S}_{\tau^*}}(s_2)$. We say $\ddot{s}$ has **irregular underspecification of input**. Determinization of $\mathcal{S}$ (cf. Def. 8.16) includes any input $i \in in_{\mathcal{S}_{\tau^*}}(\ddot{s})$ as outgoing transition in superstate $\ddot{s}$, causing over-approximation if i is irregular underspecified input. Using under-approximation for irregular underspecified input instead results in a different determinization $\mathcal{S}_{udet}$ and thus a different $ioco_{\mathcal{F}}$ relation named *uioco*, as defined in Def. 9.3, similar to [Frantzen, 2016].

Definition 9.3. Let $\mathcal{S} = (S, \to, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau)$. Then:

$$
\begin{aligned}
&\bullet\ \boldsymbol{in}^{\cap}_{\boldsymbol{\mathcal{S}_{\tau^*}}} && : 2^S \setminus \{\varnothing\} \to 2^{L_I} : \ddot{s} \mapsto \bigcap_{s \in \ddot{s}} in_{\mathcal{S}_{\tau^*}}(s) \\
&\bullet\ \boldsymbol{\mathcal{S}_{udet}} && := (S_{udet}, \xrightarrow[ud]{}, L_{\delta_\frown}) \in \mathcal{LTS}(L_I, L_U, \delta_\frown), \text{ with} \\
&\quad \boldsymbol{S_{udet}} && := 2^S_{\xrightarrow[ud]{}^*}; \\
&\quad \boldsymbol{init_{\mathcal{S}_{udet}}} && := init_{\mathcal{S}} \ after_{\mathcal{S}_{\tau^*\delta}} \ \tau; \\
&\quad \boldsymbol{\xrightarrow[ud]{}} && := \{(\ddot{s}, l, \bigcup_{s \in \ddot{s}} dest_{\mathcal{S}_{\delta\tau^*}}(s, \overset{l}{\Longrightarrow})) \mid \ddot{s} \in S_{udet}, l \in out_{\mathcal{S}_{\tau^*\delta}}(\ddot{s}) \dot\cup in^{\cap}_{\mathcal{S}_{\tau^*}}(\ddot{s})\}; \\
&\bullet\ \boldsymbol{Utraces_{\mathcal{S}_{\tau^*\delta}}} && := traces^{fin}(\mathcal{S}_{udet}, init_{\mathcal{S}_{udet}}); \\
&\bullet\ \boldsymbol{uioco} && := ioco_{\mathcal{F}} \text{ with } \mathcal{F} = Utraces_{\mathcal{S}_{\tau^*\delta}}.
\end{aligned}
$$

We drop the index if it is clear from the context.

Defining *uioco* via $\mathcal{S}_{udet}$ resembles the usual implementations of the on-the-fly test case generation algorithms for *uioco*. Since the definition of $ioco_{\mathcal{F}}$ does not consider the states in $\mathcal{S}_{udet}$, but only the derived *Straces* (cf. Sec. 8.3), $Utraces_{\mathcal{S}_{\tau^*\delta}}$ can be defined without the use of $\mathcal{S}_{udet}$ by defining the under-approximation directly on *Straces*, i.e., excluding irregular underspecified input, resulting in Def. 9.4. Since this is how other literature [Tretmans, 2008; Volpato and Tretmans, 2013; Frantzen, 2016] defines *uioco*, it is also included in this thesis. The definition also shows that $Utraces \subseteq Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$, and hence that *uioco* is weaker than *ioco*.

Definition 9.4. Let $\mathcal{S} = (S, \to, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau)$ and $s \in S$. Then:

$$\begin{aligned}
&\bullet\ \boldsymbol{Utraces_{\mathcal{S}_{\tau^*\delta}}(s)} := \{\sigma \in \mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}(s) \mid \forall \sigma_1, \sigma_2 \in L_\delta^*\ \forall i \in L_I : \\
&\qquad\qquad (\sigma = \sigma_1 \cdot i \cdot \sigma_2 \Rightarrow i \in \mathit{in}^{\cap}_{\mathcal{S}_{\tau^*}}(s\ \mathit{after}_{\mathcal{S}_{\tau^*\delta}}\ \sigma_1))\} \\
&\qquad\qquad \text{(called } \textbf{underspecified suspension traces}\text{)} \\
&\bullet\ \boldsymbol{uioco} \qquad := \mathit{ioco}_{\mathcal{F}} \text{ with } \mathcal{F} = \mathit{Utraces}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}_{\tau^*\delta}}).
\end{aligned}$$

Notes. *Utraces* can also be defined directly for superstates in an on-the-fly manner: $\mathit{Utraces}_{\mathcal{S}_{\tau^*\delta}} : 2^S \setminus \varnothing \to L_\delta^*,\ \ddot{s} \mapsto \{\epsilon\} \dot\cup \{u_\delta \cdot \sigma \in \mathit{Straces}(\ddot{s}) \mid u_\delta \in \mathit{out}(\ddot{s}),\ \sigma \in \mathit{Utraces}_{\mathcal{S}_{\tau^*\delta}}(\ddot{s}\ \mathit{after}\ u_\delta)\} \dot\cup \{i \cdot \sigma \mid i \in \mathit{in}^{\cap}(\ddot{s}),\ \sigma \in \mathit{Utraces}_{\mathcal{S}_{\tau^*\delta}}(\ddot{s}\ \mathit{after}\ i)\}$.

uioco differs from *ioco* by dropping all constraints on *Straces* the moment irregular underspecified input is encountered. This can also be achieved by demonic completion (cf. Subsec. 8.2.6): the originally irregular underspecified input now leads to chaos χ, making the *Strace* and its extensions no longer faultable, so considering them becomes unnecessary. Therefore, *uioco* can be implemented by demonic completion and *ioco*.

Since *ioco* uses over-approximation, a path in $\mathcal{S}_{det}$ that traverses an irregular underspecified input i in $\ddot{s}$ can be considered as filtering out all $s \in \ddot{s}$ with $i \notin \mathit{in}(s)$. This might seem unintuitive since $\mathcal{S}_{det}$ did allow those s to be in $\ddot{s}$. But since the SUT is a black-box, the superstates created during traversal of $\mathcal{S}$ are irrelevant for the SUT and its correctness (cf. Sec. 8.3). So filtering does not cause a problem for the SUT.

If the SUT is not completely black-box and $\mathcal{S}$ reflects the structure of the SUT (e.g., when $\mathcal{S}$ is not too abstract and used as blueprint when developing the SUT), the SUT can be considered as being in a state corresponding to some $s \in \mathit{init}_{\mathcal{S}}\ \mathit{after}\ \sigma$ for the test run σ. In this case, filtering s would require the SUT to backtrack from dead ends, i.e., undo the resolution of nondeterminism of the LTS leading to s. But fortunately, choices of nondeterminism of the LTS that lead to dead ends can be avoided by making these choices lazily, i.e., merging all paths that correspond to one trace, and only diverge when the difference becomes apparent. This corresponds to determinization with either $\mathcal{S}_{det}$ or $\mathcal{S}_{udet}$.

In summary, *ioco* and *uioco* are equally expressive, as the implementation via determinization or demonic completion have shown. But a specification that has irregular underspecified input and under-approximation might be a simpler description of some system and hence more understandable.

9.3. Underspecification and *refines*

The $\mathit{ioco}_{\mathcal{F}}$ relations only state an explicit constraint on outputs, but not on inputs. As implicit constraint, the SUT and the corresponding model must be input-enabled, so that all inputs from TCs can be handled by the SUT. Since only the specified inputs occur in the TCs, input-enabledness can be relaxed, which results in a generalized implementation relation: a relation that gives similar constraints for input and output and can be used for refinement (cf. Sec. 3.7, Subsec. 8.7.2, Sec. 14.2). As before, output on $\mathcal{F}$ that occurs in the model $\mathcal{M} \in \mathit{MOD}$ must also be specified, i.e., must also occur in $\mathcal{S} \in \mathit{SPEC}$. But now, only the input on $\mathcal{F}$ that occurs in $\mathcal{S}$ must also occur in $\mathcal{M}$, leading to the relation $\mathit{refines}_{\mathcal{F}}$ defined in Def. 9.5. It is an endorelation over *SPEC* and thus a candidate for hierarchical refinements.

Definition 9.5. Let specifications $\mathcal{S}', \mathcal{S} \in SPEC$ and $\mathcal{F} \subseteq Straces_{\mathcal{S}'_{\tau^*\delta}} \cap Straces_{\mathcal{S}_{\tau^*\delta}}$. Then

- $\mathcal{S}'$ ***refines***$_{\mathcal{F}}$ $\mathcal{S} :\Leftrightarrow \forall \sigma \in \mathcal{F}$:
  $out_{\mathcal{S}'_{\tau^*\delta}}(init_{\mathcal{S}'}\ after_{\mathcal{S}'_{\tau^*\delta}}\ \sigma) \subseteq out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}\ after_{\mathcal{S}_{\tau^*\delta}}\ \sigma)$ and
  $in_{\mathcal{S}'_{\tau^*}}(init_{\mathcal{S}'}\ after_{\mathcal{S}'_{\tau^*\delta}}\ \sigma) \supseteq in_{\mathcal{S}_{\tau^*}}(init_{\mathcal{S}}\ after_{\mathcal{S}_{\tau^*\delta}}\ \sigma)$;
- $\mathcal{S}'$ ***refines*** $\mathcal{S}$ $:\Leftrightarrow \mathcal{S}' refines_{Straces_{\mathcal{S}'_{\tau^*\delta}} \cap Straces_{\mathcal{S}_{\tau^*\delta}}} \mathcal{S}$.

$\mathcal{S}' refines_{\mathcal{F}} \mathcal{S}$ directly relates to underspecification (cf. Sec. 9.2), but now to another $\mathcal{S}' \in SPECIF$, not only to $\mathbb{M} \in MOD$:

- $out(init_{\mathcal{S}'}\ after\ \sigma) \subsetneq out(init_{\mathcal{S}}\ after\ \sigma)$ occurs at least for underspecification of output in $\mathcal{S}$ in relation to $\mathcal{S}'$;
- $in(init_{\mathcal{S}'}\ after\ \sigma) \supsetneq in(init_{\mathcal{S}}\ after\ \sigma)$ is caused by underspecification of input in $\mathcal{S}$ in relation to $\mathcal{S}'$.

Lemma 9.6 shows that *refines* is a generalization of *ioco* that drops the requirement of input-enabledness.

Lemma 9.6. *refines subsumes ioco:* $\forall \mathbb{M} \in MOD\ \forall \mathcal{S} \in SPEC : \mathbb{M} ioco \mathcal{S} \Leftrightarrow \mathbb{M} refines \mathcal{S}$.

Proof. Since $\mathbb{M} \in MOD$, f.a. $\mathcal{S} \in SPEC$ f.a. $\sigma \in \mathcal{F}$, we have $in_{\mathbb{M}_{\tau^*}}(init_{\mathbb{M}}\ after_{\mathbb{M}_{\tau^*\delta}}\ \sigma) = L_I \supseteq in_{\mathcal{S}_{\tau^*}}(init_{\mathcal{S}}\ after_{\mathcal{S}_{\tau^*\delta}}\ \sigma)$. Without the condition on $in(\cdot)$, the definitions of *refines* and *ioco* are identical. □

Refinement is very helpful for software engineering: Using abstract specifications and refinement makes MBT more lightweight and flexible; the refinement hierarchy enables MBT to lazily introduce details, which is required for iterative software development (cf. Sec. 14.2). Lemma 9.8 and Corollary 9.9 show that *refines* can construct a refinement hierarchy for conformance.

Lemma 9.7. *Let* $\mathcal{S}'', \mathcal{S}', \mathcal{S} \in SPEC$ *with* $\mathcal{S}''$*refines* $\mathcal{S}'$*refines* $\mathcal{S}$.
Then $Straces_{\mathcal{S}_{\tau^*\delta}} \cap Straces_{\mathcal{S}''_{\tau^*\delta}} \subseteq Straces_{\mathcal{S}'_{\tau^*\delta}}$.

Proof. Let $\sigma \in Straces_{\mathcal{S}_{\tau^*\delta}} \cap Straces_{\mathcal{S}''_{\tau^*\delta}}$. Assuming that $\sigma \notin Straces_{\mathcal{S}'_{\tau^*\delta}}$, there is a longest prefix σ_1 of σ that is in $Straces_{\mathcal{S}'_{\tau^*\delta}}$ and thus in $Straces_{\mathcal{S}_{\tau^*\delta}} \cap Straces_{\mathcal{S}''_{\tau^*\delta}} \cap Straces_{\mathcal{S}'_{\tau^*\delta}}$. Since $\mathcal{S}''$ refines $\mathcal{S}'$, $\sigma = \sigma_1 \cdot i \cdot \sigma_2$ for some $i \in L_I$ and σ_2 the remaining suffix of σ. Since $\mathcal{S}'$ refines $\mathcal{S}$, $\sigma = \sigma_1 \cdot u_\delta \cdot \sigma_3$ for some $u_\delta \in L_U \dot\cup \{\delta\}$, leading to a contradiction. □

Lemma 9.8. *refines is reflexive and transitive; on* $SPEC/\approx_{Straces}$, *refines is a partial order.*

Proof. Reflexivity of *refines* on *SPEC* is inherited from the subset relations used in Def. 9.5.

For transitivity of *refines* on *SPEC*, let $\mathcal{S}'', \mathcal{S}', \mathcal{S} \in SPEC$ with $\mathcal{S}''$*refines* $\mathcal{S}'$*refines* $\mathcal{S}$. To show $\mathcal{S}''$*refines*$\mathcal{S}$, let $\sigma \in Straces_{\mathcal{S}''_{\tau^*\delta}} \cap Straces_{\mathcal{S}_{\tau^*\delta}}$. Lemma 9.7 shows that $\sigma \in Straces_{\mathcal{S}''_{\tau^*\delta}} \cap Straces_{\mathcal{S}'_{\tau^*\delta}} \cap Straces_{\mathcal{S}_{\tau^*\delta}}$. Therefore $in(init_{\mathcal{S}''}\ after\ \sigma) \supseteq in(init_{\mathcal{S}'}\ after\ \sigma) \supseteq in(init_{\mathcal{S}}\ after\ \sigma)$ and $out(init_{\mathcal{S}''}\ after\ \sigma) \subseteq out(init_{\mathcal{S}'}\ after\ \sigma) \subseteq out(init_{\mathcal{S}}\ after\ \sigma)$.

For antisymmetry of *refines* on $SPEC/\approx_{Straces}$, let $\mathcal{S}', \mathcal{S} \in SPEC$ with $\mathcal{S}'$*refines*$\mathcal{S}$ *refines*$\mathcal{S}'$*refines*$\mathcal{S}$; applying Lemma 9.7 left-aligned and right-aligned yields $Straces_{\mathcal{S}'_{\tau^*\delta}} \subseteq$

$Straces_{\mathcal{S}_{\tau^*\delta}}$ and $Straces_{\mathcal{S}_{\tau^*\delta}} \subseteq Straces_{\mathcal{S}'_{\tau^*\delta}}$, so $\mathcal{S}' \approx_{Straces} \mathcal{S}$. Since *refines* does not consider states, but only the derived *Straces* of the specification (cf. Sec. 8.3), reflexivity and transitivity of *refines* are invariant under $\approx_{Straces}$ and *refines* is a partial order on $SPEC/\approx_{Straces}$. □

Corollary 9.9. *Let $\mathcal{S}', \mathcal{S} \in SPEC$ and $\mathbb{M} \in MOD$.*
Then $\mathbb{M}$ ioco $\mathcal{S}'$ refines $\mathcal{S}$ implies $\mathbb{M}$ ioco $\mathcal{S}$.

Proof. Lemma 9.6 shows that $\mathbb{M} refines \mathcal{S}'$, Lemma 9.8 that $\mathbb{M} refines \mathcal{S}$, and Lemma 9.6 that $\mathbb{M} ioco \mathcal{S}$. □

Therefore, *refines* enables a refinement hierarchy in model-based testing for *ioco*: With the hierarchy $\mathcal{S}_{max} refines \ldots refines \mathcal{S}_2 refines \mathcal{S}_1$, checking *ioco* for $\mathcal{S}_{max}$ subsumes checking *ioco* for all $\mathcal{S}_i$.

Example 9.10 shows a refinement from the domain of web services. Sec. 14.2 shows an application of refinement in agile software development, where $\mathcal{S}_1$ replaces user stories and is used for communication, to give an overview and as basis for refinements. $\mathcal{S}_{max}$ is sufficiently detailed for deriving abstract test cases that can easily be mapped to concrete test cases.

Example 9.10. Fig. 9.1 shows (an abstract part of) a refinement hierarchy for web services managing licenses, taken from WIBU SYSTEM AG's **License Central** (cf. Subsec. 14.3.1 and [URL:LC]), which is from the domain of service-oriented architecture (SOA). It shows that Fig. (d) *refines* Fig. (c) ~~*refines*~~ Fig. (b) *refines* Fig. (a) and Fig. (c) *refines* Fig. (a).

Just as *refines* generalizes *ioco*, *urefines* generalizes *uioco*, as defined in Def. 9.11.

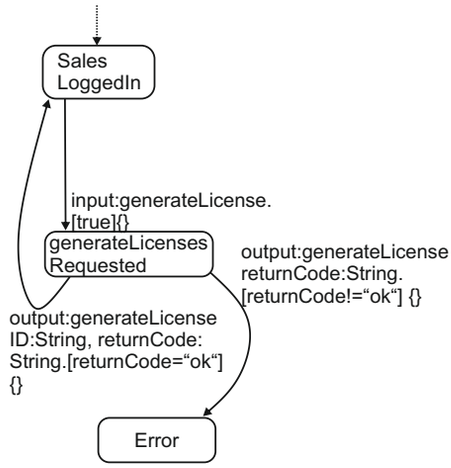
Definition 9.11. Let specifications $\mathcal{S}', \mathcal{S} \in SPEC$ and $\mathcal{F} \subseteq Utraces_{\mathcal{S}'_{\tau^*\delta}} \cap Utraces_{\mathcal{S}_{\tau^*\delta}}$. Then

$$\mathcal{S}' \textbf{\textit{urefines}} \mathcal{S} :\Leftrightarrow \forall \sigma \in \mathcal{F} :$$
$$out_{\mathcal{S}'_{\tau^*\delta}}(init_{\mathcal{S}'} \; after_{\mathcal{S}'_{\tau^*\delta}} \; \sigma) \subseteq out_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}} \; after_{\mathcal{S}_{\tau^*\delta}} \; \sigma) \text{ and}$$
$$in^{\cap}_{\mathcal{S}'_{\tau^*}}(init_{\mathcal{S}'} \; after_{\mathcal{S}'_{\tau^*\delta}} \; \sigma) \supseteq in^{\cap}_{\mathcal{S}_{\tau^*}}(init_{\mathcal{S}} \; after_{\mathcal{S}_{\tau^*\delta}} \; \sigma);$$

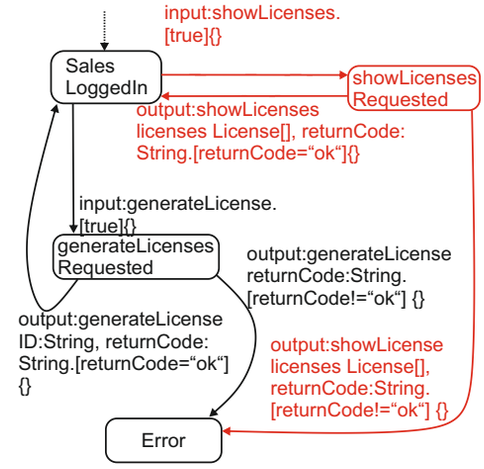
Lemmas 9.6, 9.7 and 9.8 and Corollary 9.9, as well as their proofs, can be transferred from *refines* to *urefines* by replacing every occurrence of *Straces* by *Utraces*, every *ioco* by *uioco*, and every $in(\cdot)$ by $in^{\cap}(\cdot)$.

Notes. Several variants of *urefines* are possible, which treat irregular underspecification of input differently. *urefines*'s original condition on inputs, $in^{\cap}(init_{\mathcal{S}'} \; after \; \sigma) \supseteq in^{\cap}(init_{\mathcal{S}} \; after \; \sigma)$, poses no conditions on irregular underspecified input. Depending on the use of *urefines*, exemplary alternative conditions are:

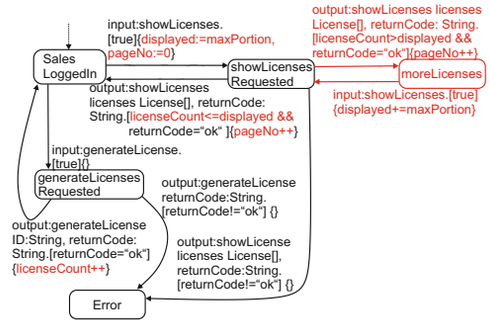
- $in^{\cap}(init_{\mathcal{S}'} \; after \; \sigma) \supseteq in^{\cap}(init_{\mathcal{S}} \; after \; \sigma)$ and $in(init_{\mathcal{S}'} \; after \; \sigma) \supseteq in(init_{\mathcal{S}} \; after \; \sigma)$;
- $faultable(in^{\cap}(init_{\mathcal{S}'} after \sigma)) \supseteq faultable(in^{\cap}(init_{\mathcal{S}} after \sigma))$, where $i \in faultable(in^{\cap}(init_{\mathcal{S}'} after \sigma))$ iff $(i \in in^{\cap}(init_{\mathcal{S}'} after \sigma)$ and $\sigma \cdot i \in faultable(Utraces_{\mathcal{S}'_{\tau^*\delta}})$ (see related work below);
- consider input subsets not for $\ddot{s} \in S_{det}$, but for each $s \in \ddot{s}$ (e.g., by additionally counting the states in which some irregular underspecified input i is enabled);



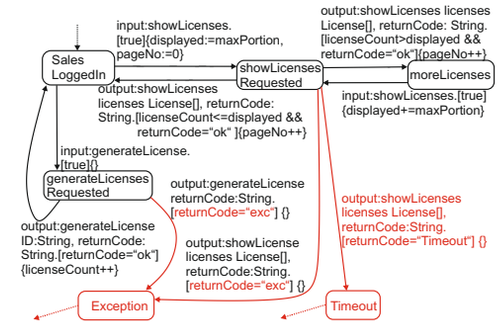
(a) Abstract specification of WS generateLicense



(b) *refines* (a) by adding input for WS showLicenses



(c) Different functionality (with pagination) for showLicenses, thus *refines* (a) but not *refines* (b)



(d) *refines* (c) by removing output for refined exception handling

Figure 9.1.: Exemplary refinement hierarchy for web services managing licenses

- disallow irregular underspecified input.

Which variant of *urefines* is most suitable depends on the application of refinement. For instance, for iterative software development processes, the original definition of *urefines* is most suitable to achieve the highest flexibility (cf. Sec. 14.2). For robustness testing, disallowing irregular underspecified input is most suitable for a final refinement step that adds a minimum robustness specification to formalize robust behavior [Shahrokni and Feldt, 2011] (e.g., fully input-enabled and according to the CRASH scale [Jr. et al., 1997; Kropp et al., 1998]) in each state. Investigating these variants of *urefines* is future work.

The next subsection depicts three variants of how *refines* and *urefines* can be implemented, and presents one such implementation. Subsec. 14.2 will show an application of *refines*.

9.3.1. Implementation

refines (resp. *urefines*) on specifications (cf. Sec. 9.3) can be implemented in several ways:

- programmatically: one possibility would be to extend an existing API for modeling a specification $\mathcal{S}$ [Veanes et al., 2008; Frantzen, 2007; Larysch, 2012]. Since such programmatic $\mathcal{S}$ can become difficult to understand and maintain if they are large, the API should have high usability by following the fail fast principle (i.e., perform early consistency checks and halt on inconcistencies [Gray, 1986; Bloch, 2006]) and concise (e.g., as a domain-specific language (DSL) or at least fluent interface [Fowler, 2010]). Then a simple call of e.g., $\mathcal{S}$.refinedBy() could return $\mathcal{S}$ wrapped, indicating a refinement and disallowing the addition of output and restriction of input. A DSL for STS specifications in the JTorX domain has already been implemented as a small proof of concept [Larsen et al., 2011]; a larger implementation is planned, so its simple extension with refineBy() is future work;
- in a graphical user interface: the larger implementation via the DSL is planned with Xtext [URL:XTEXT; Bettini, 2013], which is capable to generate a graphical editor for the DSL. Once it is available, the editor can be extended with a simple refinement mode which forbids the addition of output and restriction of input. As the editor is not yet available, its extension is future work;
- checking *refines* (resp. *urefines*) between two given specifications: Since this approach does not enforce *refines* (resp. *urefines*) during the creation of the specifications, it is the weakest solution. Since it is the simplest implementation and independent of the planned DSL implementation, this solution is chosen in this thesis.

To implement checking *refines* and *urefines*, the **iocoChecker** [URL:JTorX; Belinfante, 2010; Frantzen, 2016] was extended: The iocoChecker constructs $\mathcal{S}_{\delta\tau^*}||\mathbb{M}_{\delta\tau^*}$ for given $\mathcal{S} \in \mathit{SPEC}$ and $\mathbb{M} \in \mathit{MOD}$ (e.g., because $\mathbb{S}_{\mathit{sim}\,\mathcal{S}}$ is used). Having insight into both $\mathcal{S}$ and $\mathbb{M}$, already visited states can be detected, so loops can be avoided, and a kind of DFS can be performed, resulting in a worst case time complexity of $O(|\mathcal{S}_{\delta\tau^*}||\mathbb{M}_{\delta\tau^*}|)$ and a worst case space complexity linear in the number of states of $\mathcal{S}_{\delta\tau^*}||\mathbb{M}_{\delta\tau^*}$ (cf. Subsec. 5.3.2). So if both $\mathcal{S}$ and $\mathbb{M}$ are finite, then $\mathcal{S}_{\delta\tau^*}||\mathbb{M}_{\delta\tau^*}$ is finite, and the construction terminates. During the construction, the output constraints of *ioco* can be checked (cf. Def. 8.30), so the iocoChecker is a sound and complete algorithm to check *ioco* if both $\mathcal{S}$ and $\mathbb{M}$ are given and finite. Checking *uioco* alternatively (cf. Sec. 9.2) is also possible. The checks could also be integrated into another DFS-based algorithm that traverses $\mathcal{S}$ (e.g., LazyOTF from Chapter 11), leading to an efficient on-the-fly implementation of the iocoChecker with heuristics for large $\mathcal{S}$ and $\mathbb{M}$.

For *refines* and *urefines*, the iocoChecker was adapted by checking the corresponding constraints on inputs and outputs (cf. Def. 9.5 and Def. 9.11) and only traversing the corresponding $\mathit{Straces} \in \mathcal{F}$. Furthermore, the JTorX GUI was extended slightly and failures caused by inputs were allowed (by generalizing the FailureSituation class).

refines cannot, however, be checked as an endorelation over *SPEC* with a sound and exhaustive black-box test case generation algorithm, since invalid inputs would have to be refutable by the refining specification in a black-box manner, leading to some kind of input completion (cf. Sec. 8.1); this results in *ioco* checking (cf. Cor. 9.9).

9.3.2. Related Work

The closest related work is [Volpato and Tretmans, 2013], which introduces the relation *wioco* on *SPEC* as generalization of *uioco*. *wioco* is similar to *refines*: it introduces the set *Rtraces* of reduced, i.e., non-redundant, *Utraces*. So *wioco* corresponds to $urefines_{Rtraces}$, or the condition $faultable(in^{\cap}(init_{\mathcal{S}'} \text{ after } \sigma)) \supseteq faultable(in^{\cap}(init_{\mathcal{S}} \text{ after } \sigma))$ (cf. note above, which it inspired). *wioco* is not inteded for refinement in software engineering, but for transforming the specification to reduce the set of TCs that are generated by a sound and exhaustive test case generation algorithm. Similarly to the ioco theory not including test selection heuristics, the wioco theory only offers the formalism for such transformation, but does not offer heuristics or practical applications. The formalism, however, is quite promising since it lifts test selection to the specification level in a clean way (roughly similar to Subsec. 8.8.4).

Alternating simulation (cf. Subsec. 10.3.4) is similar to *refines* and to *wioco* [Alur et al., 1998; Volpato and Tretmans, 2013], but it does not handle quiescence.

Action refinement (cf. Subsec. 8.7.2 and [van der Bijl et al., 2005]) and *refines* are not comparable: action refinement operates on traces, performs one refinement step, and is a lossless abstraction, *refines* operates on specifications, offers a refinement hierarchy, and is lossy.

9.4. Symbolic Transition Systems and *sioco*

For test case generation, specially in the domain of model-based testing, a specification is often described with an STS (cf. Subsec. 3.4.3 and Fig. 9.1). Similar to LTSs in Chapter 8, input and output is discriminated with $L = L_I \dot{\cup} L_U \dot{\cup} \{\tau\}$, $type(\tau) = \varnothing$, $i \in L_I$ called **input gate**, and $u \in L_U$ called **output gate**. This results in a **symbolic transition system with inputs and outputs** (shortly **STS with I/O**).

Then model-based testing can be applied for STSs by expanding them to LTSs according to their defined semantics (cf. Def. 3.31). The alternative to pushing STSs down to the level of LTSs is to lift all artifacts to the symbolic level:

- a model $\mathbb{M} \in MOD$ to an input-complete initialized STS with I/O $\mathbb{M}_s$;
- a set $\mathcal{F} \subseteq Straces$ to a set of symbolic extended traces $\mathcal{F}_s$;
- TCs to symbolic TCs;
- *ioco* to *sioco*, such that $\mathbb{M}_s sioco_{\mathcal{F}_s} \mathcal{S}$ directly on an STS with I/O $\mathcal{S}$ is equivalent to checking $[[(\mathbb{M}_s)_{\mathcal{V}^0_{\mathbb{M}}}]] ioco_{[[(\mathcal{F}_s)_{\mathcal{V}^0_{\mathcal{S}}}]]} [[\mathcal{S}_{\mathcal{V}^0_{\mathcal{S}}}]]$.

The exact definitions of these symbolic artifacts are given in [Frantzen et al., 2006; Frantzen, 2016], with some variation also in [Rusu et al., 2000], and are not elaborated here since this thesis conducts model-based testing mainly on the level of LTSs.

The advantage of model-based testing on the level of STSs are:

- the STSs comprise more information: variables not only help describe the specification $\mathcal{S}$ concisely (cf. [Rusu et al., 2000]), they also preserve the structure of the data and partition all states (resp. transitions) of $[[\mathcal{S}]]$ into classes of semantically related states (resp. transitions), i.e., all instantiated states (resp. transitions) of an abstract state (resp. transition). This partition can be used for coverage, heuristics (test goals and test objectives, cf. Subsec. 11.2.4) and increased usability

for the feedback to the test engineer. By giving heuristics applied on the level of LTSs access to the artifacts of the original STS, all this information can, however, be reconstructed;
- if those classes become large (e.g., infinite), using abstract states and abstract transitions instead of expanding the artifacts helps cope with the state space explosion and can sometimes handle infinitely branching specifications.

The advantage of model-based testing on the level of LTSs are:
- symbolic artifacts are more complex (cf. [Frantzen, 2016]);
- computations over those artifacts is more complex, e.g., processing nondeterminism (cf. Sec. 13.4), heuristics such as weights (cf. Subsec. 12.3.4), and *sioco* (cf. [Frantzen, 2016]);
- hence STSimulator and JTorX (the model-based testing tools extended in this thesis, cf. Subsec. 10.3.3 and Subsec. 13.2.1) expand STSs to perform model-based testing on the level of LTSs.

Therefore, this thesis performs model-based testing on the level of LTSs. Sec. 13.4 depicts a proof of concept that model-based testing can be lifted to the level of STSs, including LazyOTF (cf. Chapter 11).

STSs can describe the behavior of the SUT and requirements on several levels of abstraction, so they are helpful for refinement (cf. Fig. 9.1). Underspecification of output can now also be performed by strengthening guards on output transitions. Conversely, underspecification of input can also be performed by relaxing guards on input transitions.

9.5. Conclusion

9.5.1. Summary

This chapter generalized the *ioco* relation to the $ioco_{\mathcal{F}}$ relation and briefly introduced derivates of *ioco* that are based on $ioco_{\mathcal{F}}$, especially the relations *uioco* and *sioco*. Furthermore, this chapter covered underspecification and the *refines* relation.

9.5.2. Contributions

The main contribution of this chapter is in Sec. 9.3: devising, investigating and implementing the *refines* relation, a generalization of *ioco* for refinement.

9.5.3. Future

Possible future work for supporting *refines* and *urefines* is creating a fail-fast fluent API or DSL for programmatic modeling of specifications (cf. Sec. 9.3) and adapting a graphical editor accordingly.

Investigate the practical use of variants of *urefines*, especially for robustness tests (cf. Sec. 9.3), is also interesting future work.

10. Introduction to Model-based Testing

10.1. Introduction

10.1.1. Motivation

Model-based testing (MBT) [ETSI, European Telecommunications Standards Institute, 2011; ISTQB, 2013] is a lightweight FM for checking whether an SUT $\mathbb{S}$ conforms to a specification $\mathcal{S}$ (e.g., according to *ioco*) under a specific condition (e.g., a test purpose) using test case generation, which is based on formal methods. Optionally, MBT also covers test case execution.

Compared to more heavyweight FMs, MBT has the advantage of being more feasible because of weaker state space explosion (cf. Chapter 7), and it incorporates functional black-box testing, which executes and inspects the real system. Applying black-box testing and sufficiently general formalisms, MBT is applicable for the hardware and embedded domain, too: The SUT is tested by simulating its complete environment by MBT, resulting in hardware-in-the-loop (HIL) or software-in-the-loop (SIL) simulation, respectively (cf. [Bringmann and Krämer, 2008]). MBT additionally has the advantage of being an extension to classical testing, which industry is familiar with. These advantages also show in the certification of safety-critical software, and their tool qualification (cf. Subsec. 1.1.3): even though formal methods might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification [DO178C Plenary, 2011; Dross et al., 2011], testing will always be part of the certification process and FMs do net get adopted broadly in industry [Hunt, 2011]. Thus MBT is a suitable advent of FMs in safety-critical software development in industry [Peleska, 2013], as well as in their certification and tool qualification [Huang et al., 2013; DO-330 Plenary, 2011].

Compared to classical testing, MBT is more efficient and flexible, less error-prone, and achieves higher coverage (cf. Subsec. 2.5). Furthermore, maintenance is reduced since not the test cases need to be adapted for change, only the specification, which is more concise. This advantage is very important for regression testing because over time, many changes in the SUT that affect many TCs become necessary. In MBT, a small change in the specification is usually sufficient, upon which the new test suite can be generated automatically. Conciseness of the specification also leads to higher understandability compared to the TS [Weißleder, 2009].

Thus MBT is in the sweet spot between testing and FM: a real mix of both, sufficiently formal to completely specify the behavior of the system and automatically generate test cases, but still sufficiently lightweight to execute the real system, stay feasible and understandable for industrial personnel, and applicable for current certification.

10.1.2. Model-based Testing

Def. 10.1 defines model-based testing in a formal but general way.

Definition 10.1. Let $\mathcal{S} \in SPEC$ be a system specification, o some kind of condition or objective selecting what aspects should be tested, and c a fixed implementation relation in $MOD \times SPEC$ determining what conformance between the SUT and $\mathcal{S}$ is checked. If test case execution is performed, the SUT $\mathbb{S} \in SUT$ must also be given.

Then **model-based testing** (**MBT**) for c derives from $\mathcal{S}$ and o a sound TS $\ddot{\mathbb{T}} \in \mathcal{TTS}(L_I, L_U, \delta)$ that meets the aspects described by o and helps to check whether the SUT conforms to $\mathcal{S}$ according to c. Optionally, $\ddot{\mathbb{T}}$ is executed and evaluated on the given SUT $\mathbb{S} \in SUT$, with $\mathbb{M} \in MOD$ being a corresponding model according to the testing hypothesis; then verdicts in $verd_{\mathbb{M}}(\mathbb{T})$ are derived for all $\mathbb{T} \in \ddot{\mathbb{T}}$, .

Within the domain of the ioco theory, Listing 10.1 determines the input, output and contracts for MBT. The names MBT_{exec}, respectively $\text{MBT}_{\cancel{exec}}$, make explicit whether test case execution is performed. For MBT_{exec}, the parameter $\mathbb{S} \in SUT$ is given implicitly if it is not relevant (e.g., in Def. 10.3).

```
// PRE: S is a well-formed system specification in SPEC (or
//      description thereof); o is some kind of condition or objective
//      selecting what aspects should be tested;
// POST: MBT_exec(S, o) gives as result a finite TS 2^TTS(L_I,L_U,δ),
//      according to Def. 10.1;
//      if MBT_exec(S, o) does not terminate, it iteratively creates the
//      TS according to Def. 10.1 by recurrently outputting a TC in
//      TTS(L_I, L_U, δ).
2^TTS(L_I,L_U,δ) MBT_exec(S, o)

// PRE: S is a well-formed system specification in SPEC (or
//      description thereof); o is some kind of condition or objective
//      selecting what aspects should be tested;
//      SUT S ∈ SUT;
// POST: MBT_exec(S, o, S) gives as result a finite TS with verdicts
//      for each TC, 2^(TTS(L_I,L_U,δ)×V),      according to Def. 10.1;
//      if MBT_exec(S, o, S) does not terminate, it iteratively creates the
//      TS according to Def. 10.1 by recurrently outputting a TC in
//      TTS(L_I, L_U, δ) with a corresponding verdict in V.
2^(TTS(L_I,L_U,δ)×V) MBT_exec(S, o, S)
```

Listing 10.1: Contracts for model-based testing for c

So if $c = ioco$, MBT implements *ioco*'s test case generation. For this, MBT can use variations of genTC (cf. Subsec. 8.8.2 and Subsec. 10.3.3) or variations of genTS (cf. Subsec. 8.8.3 and Chapter 11). MBT_{exec} additionally performs the test execution defined in the ioco theory. This thesis focuses on *ioco*, but other implementation relations are also possible.

Notes 10.2. If no specific aspect or condition should be tested, o can be omitted, i.e., be the empty set or the condition **true** to make no restrictions. Then, either all test cases, or a random selection is chosen.

In rare cases, the term "MBT" is also used for deriving TCs manually with the use of models (e.g., via the UML Testing Profile [Lamancha et al., 2009]). We exclude this meaning, since the term "model-oriented testing" comprises this [Roßner et al., 2010].

Def. 10.3 adapts the general Def. 8.43 of sound and exhaustive test case generation algorithms to MBT. Like many other literature, this thesis demands MBT to be computable and sound, but not necessarily exhaustive. In fact, exhaustive MBT is usually not possible due to combinatorial explosion in the possible test steps (leading to $O(2^{(2^{|S|})})$ many TCs for $|S|$ many states in $\mathcal{S}$ if an exhaustiveness threshold exists, cf. Table 8.1), sometimes called **test case explosion** [Mlynarski et al., 2012]. Inexhaustive MBT may, however, miss faults hidden deeply in the system or in corner cases. Thus this thesis will improve MBT's feasibility via the new method LazyOTF (cf. Chapter 11) and guidance heuristics (cf. Chapter 12).

Definition 10.3. Let MBT be an MBT method for the implementation relation c, either $\text{MBT}_{\cancel{exec}}$ or MBT_{exec}. Then we call:

$$\begin{aligned}
&\text{MBT } \textbf{sound} && :\Leftrightarrow \forall \text{ testing conditions } o:\ \text{MBT}(\cdot, o) \text{ is sound;}\\
&\text{MBT}_{\cancel{exec}} \textbf{ exhaustive} && :\Leftrightarrow \text{MBT}_{\cancel{exec}}(\cdot, \textbf{true}) \text{ is exhaustive;}\\
&\text{MBT}_{exec} \textbf{ exhaustive} && :\Leftrightarrow \text{MBT}_{exec}(\cdot, \textbf{true}, \cdot) \text{ is exhaustive.}
\end{aligned}$$

Notes. The SUT $\mathbb{S} \in SUT$ can be given implicitly for the soundness of MBT_{exec} since this thesis only considers MBT methods where $\mathbb{S}$ only influences the verdicts and test selection, but not the rules how test cases are constructed (cf. Chapter 8).

A TS in $2^{\mathcal{TTS}(L_I, L_U, \delta)} \times \mathbb{V}$, i.e., with verdicts for each TC, is exhaustive iff the same TS, but without those verdicts, is exhaustive.

This thesis only considers computable MBT, though $\text{MBT}_{\cancel{exec}}$ that never terminates and recurrently outputs a TC with a verdict can implement a computably enumerable, i.e., semi-decidable, MBT method. For finite $\mathcal{S}$ (and $\mathbb{S}$ following the testing hypothesis), $\text{MBT}_{\cancel{exec}}$ terminates with probability 1.

10.1.3. Roadmap

Sec. 10.2 gives classifications of MBT, especially about the interplay between test generation and test execution. Sec. 10.3 describes the tools TGV, TorX and JTorX, and shortly other tools.

10.2. Classification of MBT

Since MBT is a wide field, many techniques and tools exist that vary in several aspects:

- the kind of properties being tested (cf. Subsec. 10.2.1);
- the kind of test selection o (cf. Subsec. 10.2.2);
- the applied test generation technology (cf. Subsec. 10.2.3);
- the interplay between test generation and test execution (cf. Subsec. 10.2.5);
- the kind of system specifications (cf. Subsec. 10.2.6);
- the kind of test cases (cf. Subsec. 10.2.4);
- the kind of SUTs (cf. Subsec. 8.1.2);
- how detailed the SUT and its environment are specified (cf. Subsec. 10.2.7 and Chapter 2);
- whether $\mathcal{S}$ is also used for the development of the SUT.

This enumeration generalizes the **taxonomy of MBT** given in [Utting et al., 2006] by additionally distinguishing the kind of properties, test cases and SUTs.

10.2.1. The Kind of Properties

Various properties are checked by MBT. Usually, these properties originate from a specification from some level of the V-model (cf. Sec. 2.4), ranging from requirements down to unit design. These properties are mainly still safety properties and checked dynamically; but some are liveness properties (cf. Sec. 4.2.2, Chapter 6, Subsec. 8.2.2, and [Bérard et al., 2001; Fraser and Wotawa, 2006; Yi et al., 2004; Bérard et al., 2013]) and checked within the specification.

For non-functional testing, the properties are usually robustness or performance related. But the focus is mostly functional testing, ranging from general conformance testing to testing specific behavioral properties. For conformance testing (cf. Chapter 8), the implementation relation c determines how the conformance between the system specification and SUT is being checked.

For functional testing, MBT can check only safety properties, no liveness properties (cf. Sec. 4.2): Since a failing TC in dynamic black-box testing corresponds to a counterexample with a prefix causing refutation (cf. Def. 4.9), there exists no TC $\mathbb{T}$ that can check a liveness property P (i.e., such that $\mathbb{T}$ fails iff P is not met). We partition safety properties into two classes depending on their complexity: the simpler class, called **reachability properties**, contains properties where some counterexample must simply reach a state from a given set of states.

The property being checked also influences the kind of test selection, o (cf. next subsection).

10.2.2. The Kind of Test Selection

Various kinds of directives o can be used to select a subset of all possible TCs, which is important due to frequent test case explosion.

The classical test selection directives (cf. Subsec. 12.3.1) are random selection, coverage criteria (cf. Sec. 2.5) and test purposes, which are behavioral descriptions of a particular functionality to be tested [Bertrand et al., 2012; Jard and Jéron, 2005; Belinfante, 2014], usually specified by similar structures as the specification, e.g., LTSs, but often described via regular expressions (cf. Sec. 10.3 and Subsec. 12.3.1). Mixtures of these are also possible. For test selection with better guidance, this thesis introduces test objectives (cf. Subsec. 11.2.4).

Which subsets of all TCs are good test selections depends on the kind of SUT and the bugs it contains, which are unknown in advance for black-box testing. Hence test selection directives are heuristics, which are investigated in Chapter 12.

10.2.3. Test Generation Technology

Def. 10.1 shows that test generation must meet the criteria of c and o, for which various techniques can be applied:

- formal methods based on model checking techniques (cf. Sec. 10.3, [Holzmann, 1992; Engels et al., 1997; Fraser and Wotawa, 2007; Fraser et al., 2009]), especially

graph traversal algorithms similar to explicit state MC (cf. Chapters 5 and Chapters 6). Usually, trap properties are employed, to force the MC to generate the desired paths. To model the criteria of c and o, the specification might have to be extended by variables, called **trap variables** [Hamon et al., 2004, 2005];
- deductive verification (cf. [Brucker and Wolff, 2013; Engel et al., 2008; Beckert et al., 2011]);
- constraint solving, e.g., via SMT (cf. Subsec. 10.3.3, Subsec. 13.4, [Grieskamp et al., 2009; Carlier et al., 2013]);
- search-based software testing (cf. Subsec. 12.3.1, [Ali et al., 2010]),
- the simpler technique of user interaction (cf. Subsec. 10.3.3, [Zander et al., 2011; Utting et al., 2006]);
- the simpler technique of randomness (cf. Subsec. 10.3.2, [Belinfante, 2010; Ciupa et al., 2011; Oriol and Ullah, 2010; Belinfante, 2014]), which yields randomized algorithms [Motwani and Raghavan, 1995].

More details about the various test generation technologies can be found in [Broy et al., 2005; Utting et al., 2006; Utting and Legeard, 2007; Weißleder, 2009]. Nowadays, most tools employ multiple techniques: For instance JTorX for STSs uses constraint solving and randomness, LazyOTF in Chapter 11 will use constraint solving for STSs, MC-like algorithms, heuristics from search-based software testing, and some randomization.

10.2.4. The Kind of Test Cases

The kind of test cases, i.e., their structure and formalisms for representation, depend strongly on the other aspects of the MBT taxonomy, especially the interplay between test generation and test execution, the kind of system specification and the kind of test selection. Since test cases play a major role in MBT and their kind is not fully determined by the other aspects, this thesis adds the kind of test cases as another aspect to the MBT taxonomy.

TCs determine what kind of verdicts can occur (cf. Sec. 2.4). TCs have a trade-off between expressiveness and memory requirement. Therefore, this thesis defines the test case complexity in Def. 10.4. Analogously to most of this thesis's complexities in Landau notation, test case complexity is only a rough estimate, especially due to heuristics (cf. Subsec. 10.2.5).

Definition 10.4. Let $\boldsymbol{t_{curr}} \in \mathbb{N}_{>0}$ be the number of test steps that have been executed on the SUT so far (alternatively the number of executable test steps, i.e., the TCs must be able to execute at least t_{curr} test steps).

Then the **worst case test case complexity** is the worst case size of the TCs required for executing t_{curr} test steps.

The main three kinds of TCs are:
- paths, which correspond to linear TCs: for systems with uncontrollable nondeterminism, they only represent one resolution and can thus lead to the verdict inconclusive during classical test execution if the SUT's resolution of uncontrollable nondeterminism diverges from the TC (sometimes called path divergence [Anand et al., 2013]). In such a case, the expressiveness of the TCs is very low. But the TCs only have a worst case test case complexity of $O(t_{curr})$;

- graphs, e.g., $G = (S, \rightarrow, L) \in \mathcal{LTS}(L_I, L_U)$ in the ioco theory: the probability of inconclusive verdicts depends on how much of the uncontrollable nondeterminism is covered by G. If G does not perform any unwinding of $\mathcal{S}$, i.e., does not construct parts of its computation tree, the worst case test case complexity is independent of t_{curr}: $O(|\mathcal{S}_{det}|)$, which is in $O(2^{|S|})$. If G allowed multiple inputs for a state, they would need to be resolved during testing, leading to on-the-fly MBT (see below). Allowing only one input per state leads to an inexhaustive TS or a TS that contains exponentially many graphs. Furthermore, G cannot make a distinction during test execution between different paths leading to the same state in G. Unwinding paths instead leads to the next item;
- trees, e.g., $\mathbb{T} \in \mathcal{TTS}(L_I, L_U, \delta)$ in the ioco theory: Since each state of $\mathbb{T}$ represents a unique path through the tree, their expressiveness is highest, but their worst case test case complexity is $O(branch_{\mathbb{T}}^{t_{curr}})$, but much lower if $\mathbb{T}$ is degenerated. Chapter 11 will therefore use degenerated trees (which are lists modulo trees of a given depth).

Notes. If the TC does not allow the verdict inconclusive, the test case complexity is a measure for how efficiently the TC deals with uncontrollable nondeterminism.

If cycles are present in $\mathcal{S}$, using the computation tree would never terminate. Thus a bound $b > t_{curr}$ needs to be introduced (cf. genTS in Subsec. 8.8.3).

The test case complexity illustrates that MBT for SUTs with uncontrollable nondeterminism is more complex [Arcaini et al., 2013]: More computations have to be performed, more elaborate structures used for specifications and TCs, and hence the complete MBT and software development process adapted.

The test case complexity does not reflect, however, the overall cost for TCs of the corresponding MBT approach, since other aspects like guidance influence it as well (cf. Subsec. 10.2.5).

10.2.5. Interplay between Test Generation And Test Execution

The scheduling of tasks (cf. Def. 3.33) that current MBT algorithms use are either **offline** techniques or **strict on-the-fly** techniques that perform transition tasks and check tasks in lockstep (exceptions will be explored in Chapter 11). We call these approaches **offline MBT** and **on-the-fly MBT**, respectively. This subsection depicts the interplay between test generation and test execution for both approaches.

The complexities of on-the-fly and offline MBT for the number t_{curr} of executable test steps are also investigated. Due to heuristics, complexities in Landau notation offer only rough comparability of MBT approaches: they relate the complexities of the underlying algorithms, but not the overall complexities of the MBT approaches, which are more influenced by other aspects like guidance (similarly to optimizations for MC, cf. Subsec. 6.3.2). Thus complexities in Landau notation are less important for MBT approaches and often not given in literature; instead, experiments are performed.

Offline MBT

The oldest MBT approach is **offline MBT** (also called **off-the-fly MBT**): it only generates the complete test suite $\ddot{\mathbb{T}}$, but does not execute $\ddot{\mathbb{T}}$ during the generation. Afterwards, a classical testing framework like JUnit [URL:JUNIT] can be used to execute $\ddot{\mathbb{T}}$. Thus, offline MBT applies $\text{MBT}_{\not{exec}}$ (or the trivial case of MBT_{exec} that firstly performs $\text{MBT}_{\not{exec}}$ and thereafter classical execution of the returned test suite). The used test directive o can be test purposes, coverage criteria or combinations of these. An early and prominent, representative tool for offline MBT is TGV (cf. Subsec. 10.3.1).

Offline MBT can use any kind of test generation technology since test execution, which can cause restrictions, is not present. But the lack of test execution during test generation causes many deficits because there is no dynamic information from test execution yet available:

- for state space traversal, offline MBT must decide a priori, i.e., before test execution, which parts of the state space $\mathcal{S}_{det}$ (with the states $\S_{det}$) to explore (and which values to explore if the specification contains variables). This is a difficult decision: A restrictive selection will lead to inexhaustive and inconclusive testing. A broad selection can quickly cause state space explosion. Furthermore, static coverage levels can deviate dramatically from the dynamic coverage levels if uncontrollable nondeterminism is present (cf. Sec. 2.5, Subsec. 12.3.1). Often MC algorithms (cf. Chapter 5) are used for state space traversal (their severe state space explosion is demonstrated in Sec. 5.1, Subsec. 6.6.1 and Subsec. 14.3.6). State space traversal can use on-the-fly MC (cf. Subsec. 5.2.2), e.g., LTL MC (cf. Subsec. 5.2.5), and on-the-fly determinization (cf. Subsec. 8.2.5) to avoid traversing the full state space (restricting both controllable and uncontrollable nondeterminism) by aborting traversal as soon as a goal (i.e., a sensible counterexample as TC) is found. The property description F that MC checks can implement the criteria of c and o, e.g., with trap properties (cf. Subsec. 10.2.3). Often, large parts of the state space might still have to be traversed until a goal is found (Table 6.9 on page 151 shows examples for large traversals even though efficient on-the-fly algorithms like NDFS are used and goals are located close to *init*). Consequently, the worst case complexities for an efficient state space traversal (and construction of the transition system) for offline MBT are those of a full DFS: the worst case time complexity is in $O(|\mathcal{S}_{det}|)$, the worst case space complexity in $O(|S_{det}|)$ (cf. Sec. 5.3.2). If the original specification $\mathcal{S}$ contains nondeterminism of the LTS, traversal including on-the-fly determinization has an overall worst case time complexity of $O(2^{|S_{\to^*}|} \cdot |S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}})$ and an overall worst case space complexity of $O(2^{|S_{\to^*}|} \cdot (branch_{\mathcal{S}_{det}} + |S_{\to^*}|))$ (cf. Subsec. 8.2.5). If the offline MBT algorithm offers a bound b on the depth of exploration and test case generation (cf. Subsec. 8.8.3 and Subsec. 10.3.1), the overall worst case time complexity for $b = t_{curr}$ executable test steps is in $O(branch^{t_{curr}}_{\mathcal{S}_{det}} \cdot |S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}})$, for weaker guidance that does not consider all inputs in each node in $O((branchout_{\mathcal{S}_{det}} + 1)^{t_{curr}} \cdot |S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}})$. The overall worst case space complexity is in $O(branch^{t_{curr}}_{\mathcal{S}_{det}} \cdot (branch_{\mathcal{S}_{det}} + |S_{\to^*}|))$ if the full computation tree of depth t_{curr} is stored, or $O((branchout_{\mathcal{S}_{det}} + 1)^{t_{curr}} \cdot (branchout_{\mathcal{S}_{det}} + |S_{\to^*}|))$ if only one input transition per node is stored. If state space traversal applies a MC algorithm for property description F, the worst case

complexities are increased by that algorithm: Consequently, the worst case time complexities additionally depends on F, e.g., for on-the-fly LTL MC, the worst case complexities additionally contain the factor $2^{|F|}$ (cf. Subsec. 5.3.2). Furthermore, if more than one test case is generated, parts of the state space might have to be traversed multiple times, resulting in much higher complexities, similar to Subsec. 8.8.3. If test cases are generated independently and have a depth of maximal b, complexities have an additional factor in $O(t_{curr}/b)$. All these complexities are again very rough, especially since the complexities for determinization are very rough (cf. Subsec. 8.2.5);

- for test case generation, offline MBT either covers all possible nondeterministic resolutions or performs test selection: To anticipate a priori all possibilities that might occur during test execution, the test suite $\ddot{\mathbb{T}}$ must include all nondeterministic resolutions. For one test execution of $\ddot{\mathbb{T}}$, only one resolution will occur. If $\ddot{\mathbb{T}}$ is executed recurrently, a resolution r might still not occur (unless SUT is restricted to fairness_{spec} or fairness_{test}, with forbidden underspec_U, cf. Lemma 8.66). Since r might lead to a large part of the TC, a lot of resources might be spent unnecessarily. Including all resolutions yields (besides a lot of unnecessary runtime, see first item) exponentially many TCs, i.e., test case explosion, or an exponentially large TC for t_{curr} executable test steps. So we have a worst case test case complexity in $O((branchout_{\mathcal{S}_{det}} + 1)^{t_{curr}})$ (before a single test step is executed). Conversely, elaborate test selection on controllable nondeterminism (e.g., via LTL MC) possibly restricts the test case explosion, but adds runtime to the state space traversal (see first item) and does not reduce uncontrollable nondeterminism. Performing heuristics to select only few resolutions of uncontrollable nondeterminism costs high runtime and risks inconclusive and inexhaustive testing. Consequently, if the specification allows uncontrollable nondeterminism, "the only sensible approach is on-the-fly MBT" [Utting and Legeard, 2007], described below. Therefore, some tools offer both offline and on-the-fly MBT, and can cope with uncontrollable nondeterminism (or just nondeterminism on output) only in their on-the-fly mode (cf. Subsec. 10.3.4);
- the resulting TS usually achieves its directives o very inefficiently – or possibly not at all if we have a weak fairness constraint or underspec_U. Consequently, the overall number of test steps t_{curr}^{TO} (cf. Subsec. 11.3.3) that are required to achieve o is usually exponentially higher for offline MBT than the minimum number of required test steps to achieve o. Hence complexities in Landau notation can be misleading. Furthermore, estimating a bound b for the number of required test steps (similar to Subsec. 5.2.3 and Subsec. 8.8.5) is hard, especially due to uncontrollable nondeterminism, so exploration (and test case generation due to uncontrollable nondeterminism) cannot be bounded efficiently.

In summary, for $t_{curr} \in \mathbb{N}$ executable test steps, the overall **worst case time complexity of offline MBT** is in $O(2^{|S_{\to^*}|} \cdot |S_{\to^*}| \cdot branch_{\vec{\mathcal{S}}_{\to^*}})$ for a test case, the overall **worst case space complexity of offline MBT** in $O(2^{|\vec{S}_{\to^*}|} \cdot (branch_{\mathcal{S}_{det}} + |S_{\to^*}|)$; for both, the corresponding worst case complexities for the applied MC algorithm must be added, and for execution of the TCs also the requirements of the SUT. The overall **worst case test case complexity of offline MBT** is in $O((branchout_{\mathcal{S}_{det}} + 1)^{t_{curr}})$, which can also be considered as additional space requirement since test cases need to be stored.

On-the-fly MBT

Because of the deficits of offline MBT, many industrial tools (like Spec Explorer and Conformiq Designer, cf. Subsec. 10.3.4) had switched to on-the-fly MBT after gaining experience with offline MBT: **on-the-fly MBT** (also called **OTF** or **online MBT**) applies MBT_{exec} by strictly synchronously performing test case generation and test case execution. For this, on-the-fly MBT derives the possible next transitions from $\mathcal{S}$ in lockstep with executing one of them on the SUT. Therefore, uncontrollable nondeterminism is immediately resolved by the SUT, controllable nondeterminism immediately by the tool picking one transition randomly. Besides randomness, the choice can be made via objective o using test purposes. Other test directives (cf. Subsec. 12.3.1), such as coverage criteria, cannot be implemented efficiently since on-the-fly MBT only considers the immediately following transition, i.e., cannot guide any better than what lies directly ahead of the current state, i.e., within one transition. But since dynamic information is available, dynamic coverage values can be measured during test execution as information to the test engineer and as exit criterion. Measuring new coverage criteria for uncontrollable nondeterminism would also be possible (cf. Subsec. 12.3.1), but this is future work. An early and prominent, representative tool for on-the-fly MBT is TorX (cf. Subsec. 10.3.2).

On-the-fly MBT can avoid the deficits of offline MBT since now dynamic information from test execution is available, which determines the resolution of uncontrollable nondeterminism and the corresponding part of $\mathcal{S}$ to traverse. So for OTF, the only transitions that are considered are those executed plus all direct outgoing input transitions from visited states. Thus the **worst case test case complexity of OTF** for t_{curr} test steps is in $O(t_{curr})$. Because of on-the-fly determinization, the overall **worst case time complexity of OTF** is in $O(t_{curr} \cdot |S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}})$ plus the worst case time complexity of the SUT; the overall **worst case space complexity of OTF** is in $O(t_{curr} \cdot |S_{\to^*}|)$ plus the worst case space complexity of the SUT. Furthermore, once a fault is reached or o is achieved, OTF can be stopped, i.e., it has strong on-the-flyness (cf. Subsec. 3.6.2, Subsec. 6.8.6 and Subsec. 11.2.3).

Notes. Often $branch_{\mathcal{S}_{\to^*}}$ is small (e.g., due to manual or symbolic abstractions or reduction heuristics, cf. 12.3.1), so the factor can be ignored in the worst case time complexity.

If only the current superstate of $\mathcal{S}$ is stored, the factor t_{curr} can be ignored in the worst case space complexity.

Since only the immediately following transitions are considered in each state, the guidance of on-the-fly MBT is weak, picking input transitions at random, thus often missing a transition that leads to a much more meaningful TC (cf. Subsec. 14.3.4). Consequently, in practice "the random input selection strategy does not give us the tests we are interested in" [de Vries et al., 2002], and the overall number of test steps required to achieve o, t^{TO}_{curr}, is usually exponentially higher for OTF than the minimum number of required test steps to achieve o (cf. Subsec. 14.3.4). But in case a specific test purpose needs to be tested (e.g., a single path) and uncontrollable nondeterminism does not impede this test purpose, on-the-fly MBT is very efficient.

The histories of many industrial MBT tools (like Spec Explorer and Conformiq Designer, cf. Sec. 10.3) reflect these problems of on-the-fly and offline MBT: They had

started with offline MBT, and then tried to resolve its deficits for the lack of dynamic information by switching to OTF. Due to OTF's weak guidance, it is only better in cases where random exploration of behaviors detects many faults [Jard and Jéron, 2005], so the tools switched back again to mainly support offline MBT.

In summary, on-the-fly and offline MBT resolve nondeterminism differently: Offline MBT resolves uncontrollable nondeterminism by considering all cases, whereas on-the-fly MBT considers the result of test execution. Offline MBT resolves controllable nondeterminism by considering all choices (or a subset thereof via test selection) using backtracking (similar to e.g., Listing 5.3). On-the-fly MBT cannot backtrack since executed test steps cannot be undone in the SUT; so it needs to pick a transition immediately. Lacking information for a more knowledgeable decision, the choice is usually performed randomly. Consequently, offline MBT often has infeasible test case generation, test case complexity and coverage; on-the-fly MBT has weak guidance and thus also problems achieving (efficiently or at all) directive o, e.g., some coverage criteria and some feature (cf. Example 11.1). Further disadvantages in the context of the whole software development process are: long TCs that are hard to understand, weak reproducibility (cf. Subsec. 12.4.3), and weak traceability (cf. Subsec. 11.1.2 and Subsec. 13.3.5). To resolve these deficits, Chapter 11 will introduce LazyOTF, a novel method that synergetically integrates on-the-fly and offline MBT.

10.2.6. The Kind of System Specification

Many kinds of system specifications exist; among the most popular formalisms are FSMs and LTSs (cf. Subsec. 3.4). Many other formalisms can be reduced to them. The main differences are: whether the specifications are finite, what kind of nondeterminism they comprise, and their description language (cf. Subsec. 3.4.3).

Subsec. 8.2.5 describes a taxonomy of the kinds of nondeterminism that specifications can comprise, and how to resolve them. The use, resolution and consequences of nondeterminism are investigated throughout this section, and more generally throughout this thesis since they are a main aspect of formal methods.

FSMs only allow finite system specifications; contrarily, an LTS $\mathcal{S} \in SPEC$ may also be countably infinite, which is much more practical (cf. Subsec. 8.1.2 and [Huima, 2007]). Extending FSMs by adding variables, to so-called extended FSMs (EFSMs), usually remains finite since usually the variables' domain is finite [Anand et al., 2013]. Further deficits of most FSM-based testing are a strict testing hypothesis (cf. Subsec. 8.1.2, [Jard and Jéron, 2005]), inefficiencies in coping with uncontrollable nondeterminism, and more severe state space explosion [Arcaini et al., 2013]. Therefore, this thesis uses LTSs and the ioco theory.

Note. Because of physical restrictions, the systems of the real world that we investigate are finite. They are, however, unbounded. For instance, a system can have

- input that can be arbitrarily large, e.g., a BigInteger;
- an unbounded number of concurrent processes;
- arbitrarily large data structures, e.g., lists or (clock) counters;

Therefore, it is often more suitable to model these real world systems with infinite specifications by allowing arbitrarily many states or variable values (cf. Subsec. 8.1.2 or [To, 2010]).

10.2.7. Level of Detail of The SUT's and Environment's Specification

MBT can be performed on any level of the V-model (cf. Sec. 2.4). The level determines how detailed the specification must be (and which details it must contain, e.g., implementation details for unit testing and requirements for acceptance testing). Details can be avoided by abstraction, e.g., via nondeterminism (cf. Sec. 3.7), where multiple levels of abstraction between connected specifications are possible (e.g., in a refinement hierarchy, cf. formalization in Sec. 9.3 and application in Sec. 14.2). The level of lossless abstraction determines how much refinement and abstraction is performed by the test adapter (cf. Subsec. 8.7.2). The level of lossy abstraction determines what faults can be detected by MBT.

The level of lossy abstraction of the SUT's specification determines the degree of uncontrollable nondeterminism, the level of lossy abstraction of the environment determines the degree of controllable nondeterminism.

10.3. Tools

Since MBT varies in several aspects (cf. previous section), many tools [URL:MBTtoolsHP] exist, which are compared in several papers [Goga, 2001; Hartman, 2004; Belinfante et al., 2005; Utting and Legeard, 2007; Shafique and Labiche, 2010; Binder, 2011; Lackner and Schlingloff, 2012; Shafique and Labiche, 2013]. One of the strongest aspects for the test generation algorithm is whether on-the-fly MBT or offline MBT is applied; hence we pick a prominent, representative tool for both. The tools listed below all have the deficits described in Subsec. 10.2.5, especially the weak guidance for on-the-fly MBT. These deficits are also mentioned in the cited tool papers, and in the papers that compare MBT tools.

Roadmap. This section chronologically lists some MBT tools that are based on transition systems and traversal algorithms: first TGV, a main representative of offline MBT, in Subsec. 10.3.1; then TorX, resp. JTorX, main representatives of on-the-fly MBT, in Subsec. 10.3.2, resp. Subsec. 10.3.3. Since other MBT tools use similar techniques, this thesis only mentions some of them shorty in Subsec. 10.3.4.

10.3.1. TGV

The offline MBT tool **Test Generation V** (**TGV**) [Jard and Jéron, 2005] is part of the proprietary **Construction and Analysis of Distributed Processes** toolbox (**CADP**) [URL:CADP], which focuses on the design of communication protocols and distributed systems. Besides maintenance and a port to 64-bit architectures, advancements on TGV have stopped in 2005.

SPEC are the labeled transition systems with inputs and outputs, finitely many states, and a countable set of internal transitions, where cycles of internal transitions, i.e., livelocks, are allowed. As system specification description language, TGV uses the CADP formats SEQ and BCG graphs [URL:CADP]; simulation APIs make various other formats possible: LOTOS [URL:LOTOS; SC 7, JTC 1, 1989] via the CAESAR compiler [URL:CADP], the FSP language [Magee and Kramer, 1999], IF [Bozga et al.,

2002], SDL [Kerbrat et al., 1999], and UML [Ho et al., 1999; Bozga et al., 2002]. TGV uses Tarjan's DFS to detect livelocks in the specification (cf. Subsec. 8.2.2) and transforms them into quiescence. As implementation relation c, TGV uses *ioco*, but slightly modified due to quiescence from livelocks in the specification.

As test generation technology, TGV employs on-the-fly MC algorithms based on Tarjan's DFS (cf. Subsec. 5.2.2), as well as synchronous products (cf. Subsec. 3.4.3).

TGV offers as test selection directives randomness, coverage criteria, test purposes and mixtures of these. In TGV, test purposes are FSMs that may contain internal transitions. They are described with the help of regular expressions for the labels and use the format BCG graphs, IF or the old Aldebaran format Aut [URL:CADP]. A generalization of test selection directives can also handle variables, in test purposes as well as in coverage criteria. To express coverage, an extension of trap variables can be used: all reachable values of a given expression e are allotted to be covered (without the need to explicitly introduce a new propositional variable for e). e might unexpectedly cause the construction of the whole state graph, which demands the mixture of coverage criteria and test purposes as countermeasure. Furthermore, the depth of the traversal (after τ-closure) can be bounded.

TGV's testing hypothesis demands SUTs to be input-enabled. TGV does not perform test execution. Instead, TGV generates abstract TCs that are graphs, in the format BCG, Aut, or TTCN (cf. Subsec. 11.5.1, [Schieferdecker and Vassiliou-Gioles, 2003]); the translation into concrete test cases needs to be performed by other tools.

TGV has a modular architecture, similar to LTSmin (cf. Subsec. 5.5.2) and TorX (see next subsection).

10.3.2. TorX

TorX [URL:TorX; Tretmans and Brinksma, 2003; Belinfante, 2014] is an open source, on-the-fly MBT tool that is now also part of CADP [URL:CADP].

TorX uses $SPEC = \mathcal{LTS}(L_I, L_U, \tau_\not\delta)$ (cf. Subsec. 8.2.2 and Sec. 8.3). As system specification description language, TorX can use PROMELA (cf. Subsec. 3.4.3), the FSP language [Magee and Kramer, 1999], CADP's old Aldebaran format Aut [URL:CADP], LOTOS [URL:LOTOS; SC 7, JTC 1, 1989] via the CAESAR compiler [URL:CADP], and all languages that offer an implementation of the graph exploration interface OPEN/-CAESAR [Garavel, 1998]. Since the specification is accessed only on demand, i.e., lazily, this exploration interface enables the use of infinite specifications, as long as they are finitely branching (cf. [Jéron et al., 2013] and reductions in Subsec. 12.3.1).

As implementation relation c, TorX uses a deprecated version of the ioco theory where TCs need not be output-enabled.

As test generation technology, TorX employs random input selection and thus performs a random walk over the specification. Additionally, TorX contains technology to derive synchronous products (cf. Subsec. 3.4.3).

TorX offers as test selection directives (cf. Subsec. 12.3.1) randomness and test purposes. In TorX, test purposes are also called observation objectives and are represented by LTSs [de Vries and Tretmans, 2001]. They can be described by the same languages as specifications, and additionally by regular expressions in the special purpose language Jararaca. Furthermore, [Goga, 2003] introduces a user-supplied reduction heuristic and

a cycling heuristic (cf. Subsec. 12.3.1), implemented as proof of concept in an offline approach for TorX, which has not been integrated in TorX.

TorX also offers an offline MBT mode, called **batch mode**, by simply recording the executed linear test cases. The recorded linear test cases can be executed again later on (possibly leading to inconclusive verdicts).

TorX's testing hypothesis demands SUTs to be input-enabled.

TorX can easily be extended and adapted because it has a flexible component-based architecture. The components are depicted and described in the next subsection for JTorX, which is TorX's successor and main MBT tool for this thesis.

10.3.3. JTorX

The open source, on-the-fly MBT tool **JTorX** [URL:JTorX; Belinfante, 2010, 2014] is the successor of TorX, implemented in Java. The techniques introduced in the following chapters will be embedded in JTorX.

JTorX inherited most aspect from TorX: its test generation technology of random input selection and synchronous products; offline test execution; and its testing hypothesis, demanding SUTs to be input-enabled.

$SPEC = \mathcal{LTS}(L_I, L_U, \tau)$, described by any format offered by TorX, except PROMELA, and additionally GraphML [URL:GRAPHML], GraphViz [URL:GRAPHVIZ], Jararaca also for system specifications, jtorx log files, networks of timed automata in the UPPAAL input format, and TorX Explorer programs, which now additionally enable mCRL2 [Cranen et al., 2013; Groote and Mousavi, 2014; Gregorio-Rodriguez et al., 2015] and LTSmin (cf. Subsec. 5.5.2). Finally, STSs are supported and read in the XML format sax. STSs contain constraints, i.e., expressions over variables to formulate guards and updates (cf. Subsec. 3.4.3). Consequently, a constraint solver is required to solve these constraints and find satisfying variable instantiations. Currently, JTorX employs STSimulator, which uses treeSolver, which is based on a Prolog constraint solver (cf. Subsec. 13.2.1), but integration of a more powerful solver is possible (cf. Subsec. 3.3.3 and Sec. 13.4).

As implementation relation c, JTorX uses the *ioco* relation (cf. Sec. 8.5) and alternatively *uioco* (cf. Sec. 9.2).

JTorX offers as test selection directives randomness and test purposes. All random choices can be replaced by user interaction. In the standard setting, the user sets the number of steps, which are then executed in one run fully automatically by JTorX. Coverage criteria are not used for test selection, but can be measured for user feedback and exit criteria (cf. Subsec. 13.2.1 and [Sijtema et al., 2014]). In JTorX, test purposes are also called guides. They can be described as in TorX, and by jtorx log files.

JTorX inherits TorX's flexible component-based architecture, which allows easy extending, replacing, and adapting of its components (cf. Subsec. 13.2.1). From the outside in, JTorX consists of the following components, also depicted in Fig. 10.1, taken from [Belinfante, 2010]:

An **adapter** component maps between abstract and concrete test cases and covers the technicalities to fulfill the testing hypothesis (cf. Subsec. 8.7.2). Therefore, adapters depend on the SUT, the testing hypothesis and on the specification (more precisely, on L, cf. Def. 8.5). Since the adapter connects JTorX to the SUT for test execution, it is sometimes called test execution engine [Belinfante, 2014]. JTorX offers default adapters

to automatically connect to SUTs where abstract and concrete test cases are identical, i.e., SUTs that use L as interface (via TCP or the operating system's stdin/stdout pipes).

An **explorer** component is the specification language front-end that enabled to uniformly employ all the system specification description languages mentioned above, similarly to other tools that understand multiple languages, e.g.,LTSmin (cf. Subsec. 5.5.2). The specification is explored lazily. Explorers depend on the specification language, but not on the specification.

A **primer** component lazily performs determinization (cf. Subsec. 8.2.5) and aids in generating test cases, using explorer components to explore the given specifications. Therefore, primers depend on explorer components and the test case generation algorithm only. The explorer and primer together are called **derivation engine**.

A **manager** component (also called **driver**), which is the central component of TorX that controls and orchestrates the other components for on-the-fly test case generation and test case execution. It offers an interactive and an automatic mode. Therefore, the driver only depends on the other components, but not on the SUT, specification, specification language or core test case generation algorithm, which are all encapsulated by the respective component.

An optional **combinator** component derives synchronous products (cf. Subsec. 3.4.3) and is only required for test purposes. Like the driver, a combinator only depends on other components.

An optional **partitioner** component determines probability distributions over L_I and is only required to depart from equidistribution. Therefore, partitioners depend on the specification (more precisely, on L_I, cf. Def. 8.5).

An optional **IOchooser** component determines the Bernoulli distribution between stimulating and observing, and is only required to depart from equidistribution. Therefore, IOchoosers only depend on other components (or additionally on the specification if the Bernoulli distribution should not be constant during test execution).

An optional **instantiator** component instantiates variables and is only required if the specification contains variables. Therefore, instantiators depend on the specification language and on the specification.

An optional **GUI** component offers setup, interaction during test execution, and visualization of the results: The JTorX GUI increases usability by allowing all relevant parameters for on-the-fly testing to be set. It visualizes test execution by dynamically updating the plots of the specification, message sequence charts and further test artifacts [Jéron et al., 2013; Sijtema et al., 2014; Belinfante, 2014]. Without the GUI, JTorX operates in its **text mode**.

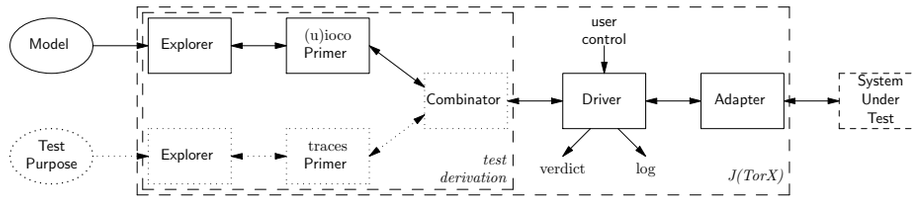


Figure 10.1.: Architecture of JTorX

JTorX offers additional functionality:

- the iocoChecker [Frantzen, 2016] for verifying (as opposed to testing) whether two specifications are *ioco* or *uioco*; additionally, it can detect underspecified traces in a model;
- simulated test execution of specifications (cf. Sec. 8.8 and [Belinfante, 2010; Jéron et al., 2013; Belinfante, 2014]).

10.3.4. Other Prominent Tools

TGV, resp. JTorX, are main representatives of offline, resp. on-the-fly MBT. Many other tools with varying focus exist [URL:MBTtoolsHP], but they all use similar techniques and are mostly proprietary. Thus this thesis focuses on TGV and JTorX.

The **symbolic test generation tool** (**STG**) extends TGV's test generation algorithms to handle symbolic data and strongly focuses on test purposes. STG operates on variants of STSs [Rusu et al., 2000; Clarke et al., 2002; Ployette et al., 2007]: It automatically derives symbolic test cases and uses the Omega constraint solver to instantiate them for test execution. However, to be useful in practice, future work needs to simplify test cases (e.g., using constraint solving, automated static analysis, or proof strategies) and investigate abstractions [Rusu et al., 2000]. The tool is no longer developed or maintained since 2007.

A younger offline MBT tool is Spec Explorer [URL:Spec Explorer; Veanes et al., 2008], which is used industrially at Microsoft and was employed in the largest case study on MBT [Grieskamp et al., 2011]. Spec Explorer is now available as free (but not open source) plug-in for Visual Studio 2010 and Visual Studio 2012. *SPEC* are FSMs (cf. Def. 3.26) or abstract state machines [Gurevich, 1994] or interface automata, which are similar to our TCs for the ioco theory (cf. Def. 8.34). Specifications are described mainly in C# (called **Model Programs**), or in Spec# or in the Abstract State Machine Language (AsmL). As implementation relation c, Spec Explorer uses alternating simulation [Alur et al., 1998], whose effect is similar to *ioco* [Anand et al., 2013], but its origin is from two-player games where in each state, the SUT must accept every input from $\mathcal{S}$, and $\mathcal{S}$ must accept every output from the SUT (cf. [Alur et al., 1998] and Sec. 9.3). So alternating simulation treats the SUT and specification symmetrically and does not demand that the SUT is input-enabled. Unfortunately, alternating simulation cannot handle nondeterminism of the LTS. As test generation technology, Spec Explorer has an integrated explicit state MC (cf. Subsec. 5.2.1). Spec Explorer offers as test selection directives randomness, a few coverage criteria, test purposes and mixtures of these. In Spec Explorer, test purposes are FSMs and called **slicing scenarios** (**scenarios** for short). They are described with the help of regular expressions and the scripting language Cord. Spec Explorer still has some on-the-fly capabilities, which were more in focus in the past. Since they have weak guidance, Spec Explorer now focuses on offline MBT.

Conformiq Designer [Huima, 2007; Utting and Legeard, 2007; Grabowski et al., 2013] is the commercial MBT tool of Conformiq Inc [URL:CONFORMIQ]. It started similarly to Spec Explorers, but has evolved since. Due to weak guidance of on-the-fly MBT, Conformiq Designer now focuses on offline MBT. While the on-the-fly mode can cope with nondeterminism on output, its offline mode cannot [Anand et al., 2013].

SPEC are finite or infinite state machines that can contain concurrency and timing constraints. Internally, they are represented in the language **CG**λ, a variant of multi-threaded Scheme [Huima, 2007; Abelson et al., 1998]. As system specification description language, the **Conformiq Modeling Language** (**QML**) was developed, which is from the domain of the Eclipse Modeling Framework (EMF). It extends UML statecharts by including variables (also with infinite data types), static and dynamic polymorphism, timing constraints and concurrency; actions are written in extended Java (but may not use Java's libraries). As test generation technology, Conformiq Designer has an integrated explicit state MC with symbolic execution capabilities (cf. Subsec. 5.2.1) and hence constraint solving. Conformiq Designer offers as test selection directives randomness, several coverage criteria implemented similarly to trap variables (e.g., requirements coverage, transition coverage, branch coverage, atomic condition coverage), boundary value analysis [Utting and Legeard, 2007; Weißleder, 2009; Naik and Tripathy, 2011], test purposes and mixtures of these. In Conformiq Designer, test purposes are FSMs and are called **abstract use cases**. They are described with the help of partial or high-level UML use cases or sequence diagrams [Fowler, 2003]. Conformiq Designer has a flexible plug-in architecture, including distribution (cf. Subsec. 11.5.1) and multiplexing components, enabling testing distributed systems. Furthermore, Conformiq Grid offers distributed test case generation (cf. Subsec. 11.5.1). In summary, Conformiq Designer is one of the most powerful and modern MBT tools, but not open source and not freely available.

Web services (cf. Subsec. 3.5.2) are a popular architecture for distributing application over the Internet. So machines use web services in their computations and thus depend on their correctness. Since web services are black-box systems that have formal specifications, MBT is a suitable approach for checking them. Hence there are MBT tools that have specialized on checking WSs. Since STSs are an alternative to BPEL and WSDL-S, some MBT tools are based on STSimulator (cf. Subsec. 10.3.3,[URL:STSimulator; Frantzen, 2016]), for instance the PLASTIC Framework [Bertolino et al., 2008], Jambition [Frantzen et al., 2009], and Audition [Bertolino et al., 2004]. But incorporating all technical aspects of WSs directly in the MBT tool can become elaborate [Pascanu, 2010]. An alternative is using a regular MBT tool and moving all web service technicalities to the level of the test adapter, while abstract test cases are not aware of the web service technology (cf. Subsec. 8.7.2 and Subsec. 14.3.2).

Finally, since parallelization is "a key to MBT success" [Nupponen, 2014], some MBT tools have a focus on parallelization. These, and more generally parallel test automation tools, are covered in Subsec. 11.5.1.

10.4. Summary

This chapter gave various definitions related to MBT in a general way, based on previous chapters, to ensure flexibility and comparability. The established taxonomy of MBT was introduced and extended (Fig. 15.1 on page 377 uses this for positioning). Finally, various MBT tools were introduced and positioned according to the taxonomy. Considering their deficits motivated an alternative approach (cf. Chapter 11).

11. Lazy On-the-fly MBT

11.1. Introduction

11.1.1. Motivation

Chapter 10 has investigated the bad trade-off between on-the-fly and offline MBT: offline MBT cannot handle large specifications and uncontrollable nondeterminism, whereas on-the-fly MBT has insufficient guidance (cf. Example 11.1). Handling uncontrollable nondeterminism is relevant in practice (cf. examples in Sec. 3.7, Subsec. 8.2.5, and [Huima, 2007; Fraser et al., 2009]). Guidance is just as relevant in practice: Current tools returned from on-the-fly MBT to offline MBT for better guidance since on-the-fly MBT does not efficiently target precise features and potential faults (cf. Subsec. 10.3 and [Jard and Jéron, 2005]). Because of this bad trade-off, there is no significant correlation between the size of the test suite $\ddot{\mathbb{T}}$ generated by current tools and the fault detection capability of $\ddot{\mathbb{T}}$; the situation could be improved by "semantically integrating requirements into MBT and into the generation of the test suite" $\ddot{\mathbb{T}}$ [Fraser and Wotawa, 2006; Bifﬂ et al., 2006; Rajan et al., 2008b; Lackner and Schlingloff, 2012]. This chapter introduces lazy on-the-fly MBT to avoid this bad trade-off, and test objectives (TOs) to semantically integrate requirements, specifications, and their coverage into MBT and test case generation.

11.1.2. Lazy On-the-fly MBT

To avoid the respective disadvantages of offline and on-the-fly MBT, i.e., the bad trade-off, this section introduces a novel method that synthesizes these two approaches, baptized **lazy on-the-fly MBT** (**LazyOTF** for short): it executes parts of TCs lazily on-the-fly on the SUT, i.e., only when there is a reason to, e.g., when a TC reaches a test goal (cf. Subsec. 11.2.4), a certain depth, or some choice point of uncontrollable nondeterminism. Therefore, we do interleave graph traversal of $\mathcal{S}$ and execution of the SUT, but not strictly synchronously for each test step, but only loosely after several steps. Thus LazyOTF's scheduling of tasks (cf. Def. 3.33) does insert check tasks between transition tasks and is hence on-the-fly, but does not intertwine strictly in lockstep, but lazily.

Since LazyOTF integrates test case execution, it has the contract of MBT_{exec} of Listing 10.1. Listing 11.1 shows the abstract LazyOTF algorithm, which cycles through **phases**, each consisting of firstly a **traversal sub-phase** and secondly an **execution sub-phase**. The abstract algorithm contains the following three polymorphic methods:

- **exitCriterion(dynamicInfo)**, which determines via dynamic information given in the argument dynamicInfo whether the LazyOTF algorithm should cycle another iteration p through both sub-phases or terminate (e.g., when all TOs or some coverage level is achieved, or fail occurs and LazyOTF should not continue thereafter);

- **traversalSubphase**($\mathcal{S}$, $\ddot{o}$, **dynamicInfo**), which traverses some part of $\mathcal{S}_{det}$ via on-the-fly determinization (cf. Subsec. 8.2.5), generates (e.g., via gen, cf. Sec. 8.8) and chooses a heuristically best TC $\mathbb{T}_p \in \mathcal{TTS}(L_I, L_U, \delta)$ within that sub-graph: Based on the given TOs $\ddot{o}$ and dynamic information dynamicInfo, guidance heuristics determine which TCs are best (cf. Subsec. 11.2.4). Phase heuristics determine the sub-graph that is traversed: The initial superstate $\ddot{s}$ of the sub-graph is determined by dynamicInfo, the sub-graph's border via $\ddot{o}$, by so-called **inducing states** (investigated in Subsec. 11.2.3); furthermore, depth bounds for the TCs can be set, depending on dynamicInfo;
- **testExecutionSubphase**($\mathbb{T}_p$, $\mathbb{S}$, **dynamicInfo**), which executes $\mathbb{T}_p$ on $\mathbb{S}$: The state that $\mathbb{S}$ is in at the beginning of this testExecutionSubphase is determined by the previous phases (and is hence contained in dynamicInfo). Test execution yields further dynamic information, which is fed back to exitCriterion and the next traversalSubphase.

dynamicInfo comprises all **dynamic information** that the algorithm collects during test case execution that cannot be collected without executing the SUT:

- verdicts, and their location;
- resolution of uncontrollable nondeterminism;
- consequential information, e.g., the reached behaviors (requirements, other artifacts, or safety properties), which are modeled by test goals and test objectives and get discharged once they are reached. This is encoded in dynamicInfo by updating $\ddot{o}$ (cf. Subsec. 11.2.4).

```
proc 2^(TTS(L_I,L_U,δ)×𝕍) LazyOTF(LTS(L_I, L_U, τ) 𝒮, TOs ö, SUT 𝕊)
  p := 0; dynamicInfo := ∅;

  while (!exitCriterion(dynamicInfo)) do
    p++;
    TTS(L_I, L_U, δ) 𝕋_p :=
                traversalSubphase(𝒮, ö, dynamicInfo);
    dynamicInfo :=
            testExecutionSubphase(𝕋_p, 𝕊, dynamicInfo);
  od;
  return all 𝕋_p and their verdicts;
end;
```

Listing 11.1: Abstract LazyOTF($\mathcal{S}$, $\ddot{o}$, $\mathbb{S}$) algorithm

Example 11.1. Fig. 11.1 depicts one exemplary phase of LazyOTF:

- the traversal sub-phase generates a heuristically meaningful TC $\mathbb{T}_1$ by traversing a sub-graph in the specification, bounded by a depth bound of 5 and by inducing states (q_4 and the test goal q_3). Output transitions to fail are omitted in $\mathbb{T}_1$ (cf. Subsec. 13.2.2);
- then $\mathbb{T}_1$ is executed on the SUT, and all dynamic information is fed back for the next phase.

In this concrete example, one phase is sufficient to reach the test goal q_3 in the generated TC, $\mathbb{T}_1$. Due to inducing states and bounds, the traversed sub-graph and TC

$\mathbb{T}_1$ are kept small. If the SUT does not exhibit the resolution of uncontrollable nondeterminism leading to q_3, or if the number $k = 1$ in the guard $[i > k]$ of transition $q_1 \to q_3$ were larger, multiple phases are required to reach the test goal q_3. For this case, Chapter 12 will exploit dynamic information to reach test goals more efficiently via guidance heuristics.

In contrast, on-the-fly MBT randomly chooses an input in q_0, so the transition $q_0 \to q_2$, which resets i, is taken with probability 0.5. So if we increase the number k, the expected number of steps to reach q_3 in a random walk increases exponentially in k (cf. Subsec. 14.3.4 and Subsec. 14.3.10).

Conversely, offline MBT has to deal with state space explosion, even for on-the-fly MC if the state space below q_2 or q_4 is large and traversed before q_3, or k and the domain of i are large. If offline MBT uses a bound, q_3 may not be reachable within that bound. With state space explosion comes test case explosion (from controllable nondeterminism) and high test case complexity (also from uncontrollable nondeterminism): For every visited state q_0, two extension of the TCs are possible due to $?a$ and $?b$; for every visited state q_1, TCs must contain at least the two outputs $!x$ and $!z$ to avoid inconclusiveness.

Note. High test case complexity might be reducible if TCs are symbolically represented, but only if all concrete states can be represented by much fewer symbolic states.

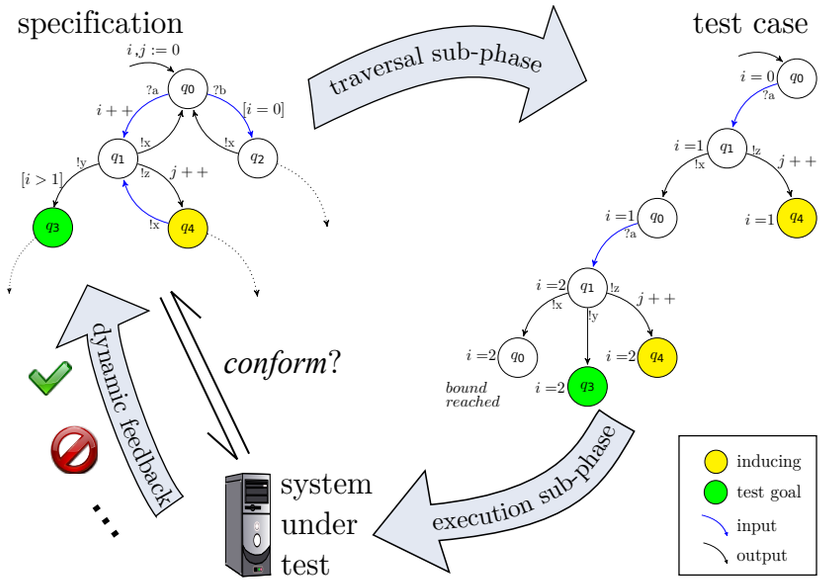


Figure 11.1.: Exemplary phase of LazyOTF

LazyOTF's integration of on-the-fly and offline MBT is **synergetic** since LazyOTF has the unique feature of combining dynamic information with backtracking:

- all the dynamic information, as in on-the-fly MBT, is available after the delay of the current traversal sub-phase. So the current traversal sub-phase can make use of all dynamic information from previous test execution sub-phases, such that test case generation may depend on the test execution of TCs from previous phases;

- backtracking, as in offline MBT, is available within each sub-graph, to search for a meaningful TC;

So unlike on-the-fly or offline MBT, LazyOTF can use dynamic information during **guidance**, i.e., during the search for meaningful TCs. Thus LazyOTF

- need not consider all nondeterministic cases, as opposed to exhaustive offline MBT (cf. Subsec. 10.2.5) to avoid inconclusive verdicts;
- can efficiently guide SUTs with uncontrollable nondeterminism to reach yet untested behaviors (e.g., requirements, test objectives, or coverage tasks). For instance, if a requirement is unexpectedly not, or hardly, reachable from *init* due to uncontrollable nondeterminism, but easily reachable from the current state, LazyOTF adapts to this. Guidance can use novel heuristics, such as coverage criteria for uncontrollable nondeterminism (cf. Subsec. 12.3.1, [Faragó, 2011]) or taking previous failures into account to avoid the same failure again, but to find related ones (cf. Note 8.36).

Considering extreme cases of the depth bound show that LazyOTF subsumes on-the-fly and offline MBT:

- if all depth bounds are 1 (or all states are inducing), LazyOTF performs the traversal of $\mathcal{S}$ and test execution in strict lockstep, like on-the-fly MBT. If no test objectives are used, LazyOTF behaves identical to on-the-fly MBT (cf. $P_{vanishing}$ in Def. 12.6);
- if no states are inducing and the depth bound is sufficiently large, LazyOTF performs similarly to offline MBT in the beginning: it generates a full TC $\mathbb{T}$ without any dynamic information. But then, LazyOTF executes $\mathbb{T}$ before generating the next TCs, for which dynamic information could be used. Both LazyOTF and offline MBT via on-the-fly MC algorithms (cf. Subsec. 10.3.1) prune the state space that has to be traversed for generating TCs. To determine the order in which states are traversed and TCs are generated, Chapter 12 will show that LazyOTF employs heuristics tailored towards MBT, whereas offline MBT usually employs standard on-the-fly MC algorithms (cf. Subsec. 5.2.2, Subsec. 10.3.1).

In summary, LazyOTF's guidance is more directed than the randomness of on-the-fly MBT. But randomness is usually not completely replaced by LazyOTF's guidance since the applied heuristics can rank multiple choices as most meaningful, from which one is usually chosen randomly (cf. Subsec. 12.3.4, Sec. 13.5, [Nieminen et al., 2011]). But like JTorX (cf. Subsec. 10.3.3), LazyOTF offers to replace full automation implemented with random choices by user interaction. LazyOTF's guidance is also more directed than on-the-fly MC algorithms for offline MBT, and better-informed than the guidance of offline MBT because of the available dynamic information. Therefore, LazyOTF can efficiently handle large specifications and uncontrollable nondeterminism to automatically generate TCs that are shorter and more revealing (cf. Sec. 14.3). Shorter TCs are simpler to execute and to understand [Anand et al., 2013; Gay et al., 2015], solving the failure analysis problem of understanding the cause of a failing TC [Arcaini et al., 2013]. Furthermore, the dynamic information of reached behaviors includes which TO has been discharged where. So for TOs that describe requirements (or other specification or design artifacts), LazyOTF yields automatic traceability between TCs and those artifacts for free (cf. Chapter 2, Subsec. 13.3.5, and Subsec. 14.2.2), which is important in practice [Peleska, 2013]. Finally, LazyOTF is more reproducible (cf. Subsec. 12.4.3).

11.1.3. Roadmap

Sec. 11.2 reiterates the taxonomy of MBT for LazyOTF to give insights to all aspects (see also Fig. 15.1 on page 377), especially the new possibilities for test selection. Sec. 11.3 formalizes the test selection, interplay between test generation and test execution and then analyzes LazyOTF. Sec. 11.4 introduces the two related works. Sec. 11.5 firstly introduces distribution for MBT and the corresponding related work, and then relates them to distributed LazyOTF, which is introduced. Sec. 11.6 summarizes the chapter, our contributions and possible future work.

11.2. Classifications

To show the details of LazyOTF, this section investigates the aspects according to the taxonomy of MBT given in Sec. 10.2.

11.2.1. Test Generation Technology

Test cases are generated iteratively in traversalSubphase($\mathcal{S}$, $\ddot{o}$, dynamicInfo). So within each sub-graph, any test generation technology can be applied. To make strong use of the available dynamic information and backtracking capabilities, the test generation technology must be adapted and the simpler test generation technologies of user interaction and randomness should be kept to a minimum.

This thesis will implement test generation in traversalSubphase($\mathcal{S}$, $\ddot{o}$, dynamicInfo) mainly using graph traversal and MC-like algorithms, based on genTS (cf. Subsec. 8.8.3), and employ heuristics similarly to search-based software testing (cf. Subsec. 11.2.4). Since the test generation algorithm and the heuristics are intertwined, they are all investigated in detail in Chapter 12. Furthermore, constraint solving is applied for STSs (cf. Subsec. 10.3.3), and randomness is used to choose amongst equally meaningful TCs.

Alternatives are possible, for instance applying SMT solvers as main test generation technology (cf. Subsec. 3.3.3 and Sec. 13.4).

11.2.2. Properties

This thesis focuses on functional properties. We check conformance based on the given implementation relation c, for which we use the *ioco* relation. The reduction from full $\mathcal{S}$ to a sub-graph, which traversalSubphase($\mathcal{S}$, $\ddot{o}$, dynamicInfo) performs, is invisible to the $\text{MBT}_{\not\text{exec}}$ algorithm that traversalSubphase applies. Therefore, LazyOTF can check any implementation relation by using a corresponding $\text{MBT}_{\not\text{exec}}$ algorithm. If the applied $\text{MBT}_{\not\text{exec}}$ algorithm can integrate heuristics that respect $\ddot{o}$ and dynamicInfo, LazyOTF can unfold its synergetic power (cf. Subsec. 11.1.2). For instance, *uioco* can be performed on unwound LTSs by simply restricting inputs (cf. Sec. 9.2).

11.2.3. Interplay Between Test Generation and Test Execution

On an abstract level, the interplay between test generation and test execution is given in Listing 11.1. Heuristics determine where and when the sub-phases are swapped, e.g., in states with nondeterminism on output. Details on heuristics are given in Chapter 12.

In this subsection, only the overall approach and the information that the heuristics depend upon are introduced.

Traversal Sub-phase

The part of the graph that traversalSubphase traverses is determined by heuristics we call **phase heuristics**, based on the current dynamicInfo and on user-supplied meta-information about $\mathcal{S}$: Depending on dynamicInfo, each state but traversalSubphase's first can indicate that traversal beyond the state should not continue immediately, but be postponed to the next traversal sub-phase if the SUT makes the corresponding nondeterministic resolutions. Since such states induce switching between a traversal sub-phase and an execution sub-phase, they are baptized **inducing states**. Strategies can be used to annotate states programmatically as inducing, instead of manually (cf. Subsec. 13.2.3). Since nondeterminism of the LTS is uncontrollable, we define a superstate $\ddot{s}$ to be inducing iff $\exists s \in \ddot{s} : s$ is inducing.

The test engineer can set inducing states for many reasons, for instance at points where:

- the SUT has a high degree of uncontrollable nondeterminism;
- more generally, where traversalSubphase requires dynamic feedback;
- where traversalSubphase has costly specification traversal that is likely not required during test execution;
- the SUT is busy anyways, or where the SUT is able to wait (i.e., is quiescent);
- the test engineer wants to interact with LazyOTF.

Notes 11.2. Each traversalSubphase's root is not inducing since at least one transition should be chosen for testing, i.e., the returned TC $\mathbb{T}_p$ should have a depth of at least one.

Instead of deciding inducingness based on states from $\mathcal{S}$, it could be decided more generally based on superstates or paths. The implementation can be extended for this without difficulty (cf. Subsec. 14.3.3). But using states is often sufficient (cf. Sec. 14.3), can be implemented more efficiently, is less complex and more natural and usable: The user only needs to determine inducingness for states in $\mathcal{S}$, which is more intuitive and exponentially less work than for superstates or paths.

To restrict the runtime and memory requirement of a traversalSubphase (cf. Subsec. 14.3.6, Subsec. 10.3.1), a bound is set on the depth of the sub-graph's computation tree (cf. Def. 4.4) and consequently on the depth of the generated TC (cf. Def. 11.8). To make use of dynamicInfo, the bound is set individually for each phase p by heuristics (cf. Sec. 12.2), resulting in a sequence of bounds $b_p \in [b_{min}, \ldots, b_{max}]$, with the **minimal bound** $\boldsymbol{b_{min}} \in \mathbb{N}_{>0}$ and **maximal bound** $\boldsymbol{b_{max}} \in \mathbb{N}_{>0}$ determined by the user-supplied heuristic configuration.

Test Execution Sub-phase

traversalSubphase always provides one TC $\mathbb{T}$, which testExecutionSubphase thereafter fully executes. Since $\mathbb{T}$ incorporates all resolutions of uncontrollable nondeterminism within the corresponding sub-graph, test execution never leads to inconclusive verdicts.

For the next traversalSubphase and exitCriterion, testExecutionSubphase can return the accrued dynamicInfo at the end of the sub-phase. If dynamicInfo should be processed immediately (e.g., for parallelization or visualization, cf. Subsec. 13.3.4), it should be made available immediately or processed accordingly by testExecutionSubphase.

To analyze the degree of laziness for MBT, i.e., its on-the-flyness, we define **on-the-flyness for MBT** as the number of check tasks and transition tasks required to feed information from one sub-phase back to the other sub-phase (cf. Subsec. 3.6.2). So for LazyOTF, if traversalSubphase detects that a TO can potentially be discharged, how much more time is spent in traversalSubphase until the next testExecutionSubphase is executed to discharge a TO? And if testExecutionSubphase yields some dynamicInfo such as a nondeterministic resolution or a discharge, how much more time is spent in testExecutionSubphase until the next traversalSubphase is executed with the new dynamicInfo? So for LazyOTF, the size of the sub-graphs, especially the bounds, strongly influences its on-the-flyness. For on-the-fly MBT, on-the-flyness is as high as possible. But it only considers the direct outgoing transitions to find a potential TO. For offline MBT, traversal is performed a priori, so its degree of on-the-flyness depends on the applied on-the-fly MC algorithm to avoid unnecessary traversal once a counterexample is found (cf. Subsec. 6.8.6), if one test case is generated. If many test cases are generated, the generation of the full test suite hinders on-the-flyness. Furthermore, dynamicInfo is never fed back to the traversal.

Strong on-the-flyness is very important in practice, since it means little additional resources for traversal once a potential TO is found, and little additional test execution steps until traversal can make use of the new dynamic information. For the final (or a single) test case, this means that traversal and test execution can stop as soon as the desired TOs have been achieved. But the strongest performance improvement in test execution comes from guidance with the help of dynamic information (see next subsection), which yields very meaningful test cases, i.e., few needed test steps to achieve *ö*. For slow SUTs, test case execution becomes the bottleneck and thus significant for the overall speed of fault detection and hence success of the approach [Fraser et al., 2009; Nieminen et al., 2011; Gay et al., 2015].

11.2.4. Test Selection

Test selection is the guidance heuristics that traversalSubphase employs within each sub-graph to select a potentially most meaningful TC. Test selection is a hard problem in theory and practice (cf. Chapter 12 and [Lackner and Schlingloff, 2012; Nupponen, 2014; Gay et al., 2015]), which many domains like MBT [Utting and Legeard, 2007] and search-based software testing (cf. Subsec. 10.2.3, [Ali et al., 2010]) deal with. This thesis contributes an own chapter on heuristics, Chapter 12, which investigates properties, conditions and solutions for guidance heuristics. In this subsection, only the overall approach and information required for LazyOTF's test selection via TOs is introduced.

Since test selection is only the responsibility of traversalSubphase, it can independently choose the kind of test selection without any restriction from the rest of the LazyOTF algorithm. But test selection must be performed individually for each sub-graph, and classical test selection (i.e., coverage criteria, test purposes or randomness, cf. Subsec. 10.2.2) is not sufficient to unfold LazyOTF's synergetic power. Instead, more suitable heuristics make use of dynamic information and semantically integrate requirements and specifica-

tions, yielding better guidance for LazyOTF. One possible solution is via test objectives (TOs). A test objective is used to generate the necessary tests for a new aspect, artifact or coverage task, e.g., a new requirement or specification element for a newly implemented feature. Therefore, TOs are a natural fit for the software development process (especially agile approaches that focus on few features each sprint, cf. Sec. 14.2). Test objectives are defined on an abstract level in Def. 11.3.

Definition 11.3. Let E be a feature, requirement, specification artifact or any other element to be tested.

Then a **test objective** (**TO** for short) $\boldsymbol{o}$ for E comprises heuristical settings to efficiently generate the tests for E.

The moment E has been tested sufficiently, o is **discharged** by a **discharge function**, i.e., o becomes **inactive**; beforehand, o was **active**. A test objective must be dischargeable.

$\ddot{\boldsymbol{o}}$ denotes the finite set of currently active test objectives.

Discharging depends on the SUT's resolution of uncontrollable nondeterminism, i.e., on dynamic information. Hence it is best performed in testExecutionSubphase. We say **LazyOTF is active** at the moment iff the set $\ddot{o}$ of currently active test objectives is not empty and LazyOTF currently uses our heuristics for test selections (i.e., not OTF's random choice, cf. $\mathrm{P}_{vanishing}$, Def. 12.6).

By composing the heuristics of all $o \in \ddot{o}$ (cf. Subsec. 12.3.7), the overall guidance heuristic becomes a **synergetic TO** in $\ddot{\boldsymbol{o}}$, i.e., discharging all TOs in $\ddot{o}$ requires less test steps than $\sum_{o \in \ddot{o}}$(test steps to discharge o) on average. The order in which the TOs $o \in \ddot{o}$ are discharged is determined automatically on-the-fly by the composed guidance heuristics and depends on the strength of their respective guidance heuristics, how easily they can be reached, and of course on the resolution of uncontrollable nondeterminism.

Using this TO technique is more flexible and expressive than classical test selection techniques, which it subsume: coverage criteria by defining corresponding coverage tasks as TOs, randomness by $\ddot{o} = \varnothing$ (cf. $\mathrm{P}_{vanishing}$, Def. 12.6), a test purpose with the help of a manually defined TO that always picks a path that is in the test purpose, as long as there is one. Since there is no single superior test selection technique (cf. Subsec. 12.3.1), this flexibility is important [Gay et al., 2015].

Often, tests for a new requirement or feature correspond to reachability properties in the specification $\mathcal{S} = (S, \rightarrow, L)$ [Arcaini et al., 2013]. Usually, not a single state $s \in S$ needs to be reached, but any state from a set of states $\ddot{s} \subseteq S$. Such a set of states is called a **test goal** (**TG**). Since the only insight into the SUT's behavior that black-box testing allows are suspension traces, a TG $\ddot{s}$ corresponds to the behavioral property $\boldsymbol{P_{\ddot{s}}} := \{\sigma \in \mathit{Straces}_{\mathcal{S}_{\tau^*\delta}} | \ddot{s} \cap \mathit{init}_{\mathcal{S}} \mathit{after}_{\mathcal{S}_{\tau^*\delta}} \sigma \neq \varnothing\}$. So during test execution, $\ddot{s}$ is **reached** (or **achieved**) when there is a prefix σ of the test run trace with $\sigma \in P_{\ddot{s}}$. Hence reaching any superstate in $\mathcal{S}_{det}$ containing s is considered as reaching $\ddot{s}$, and TGs are lifted from states to superstates analogously to inducing states. Similarly to inducing states, this simplifies the definition, implementation and use of TGs, since otherwise, a set of superstates $\ddot{\ddot{s}}$ would be needed to specify a TG. Usually, using states is sufficient. In other cases, the implementation can be extended without difficulty (demonstrated by S_{isol}, cf. Subsec. 14.3.3).

A TO o describing a reachability property can contain a TG to make the definition and configuration of o's guidance heuristic and discharge function simpler and hence more usable. An **active test goal** is a test goal of some active TO. In this thesis, inactive TGs are usually discarded, so a TG is an active TG if not described otherwise.

For a TO describing a more complex safety properties (cf. Subsec. 10.2.1) than reachability, the heuristics settings and discharge function become more complex and cannot be based on TGs only. These TOs can be implemented manually or using multiple auxiliary TOs (see Subsec. 13.3.2 and o_{dec} in Subsec. 14.3.3).

Notes 11.4. Since our TO technique subsumes classical test selection techniques, they can easily be combined, which leads to more meaningful test suites, as described in Subsec. 12.3.1 and [Abdurazik et al., 2000; Dupuy and Leveson, 2000; Fraser and Wotawa, 2006; Rajan et al., 2008b; Gay et al., 2015]). Due to these studies, [Fraser and Wotawa, 2006; Gay et al., 2015] suspect that future test generation tools will need to flexibly augmenting coverage criteria with additional, domain-specific objectives to guide test generation. Our TO technique fulfills this need. Since TOs subsume a test directive o (cf. Subsec. 10.2.2), we also denote TOs by o.

TGs are not sufficient to simulate test purposes since test purposes can specify properties that are more complex than reachability.

Since mutation testing can make tests more meaningful (cf. Sec. 2.5), it is interesting future work to investigate TOs for mutation testing: Similar to mutant selection [Grün et al., 2009], we could add a TO_m for each mutant m to guide test generation to cover the mutated statement. Since LazyOTF integrates test execution, m can be executed in parallel; if m is killed, TO_m is discharged.

Another approach to test selection is model-based mutation testing: It mutates the specification $\mathcal{S}$, not the source code like regular mutation testing, and is hence also called specification mutation testing. A model-based mutant m is killed if the generated TS fails on SUTs that are conform to m [Aichernig et al., 2011, 2014] (or dually if the tests generated from m fail on SUTs that are conform to $\mathcal{S}$ [Hollmann, 2011], which requires the generation of a TS from each mutant m). Either way, m is killed if tests lead to a location where m's and $\mathcal{S}$'s observations differ. Therefore, using a conformance checker (e.g., the iocoChecker in Subsec. 9.3.1) for test generation is an efficient guidance to the observational difference between $\mathcal{S}$ and m [Aichernig et al., 2007, 2011, 2014]. With this approach, m can be considered a test purpose [Aichernig and Tappler, 2015]. Due to the high number of mutants and their conformance checking, especially for equivalent model-based mutants, mutation based test case generation strategies are not efficient [Aichernig et al., 2014; Aichernig and Tappler, 2015]; hence integrating model-based mutation testing as test selection heuristics in LazyOTF is future work. To improve efficiency, an on-the-fly iocoChecker and on-the-fly mutations could be integrated in LazyOTF. Further future work can compare the meaningfulness of simple model-based mutation test selection strategies, particularly those resulting in linear test cases [Ammann et al., 1998; Aichernig et al., 2014], and TOs for mutation testing and classical mutants of the implementation, since the latter can be implemented efficiently. Or maybe an even more efficient implementation by LazyOTF's default guidance heuristics (cf. Sec. 12.3), e.g., reaching specific transitions with specific conditions, is just as meaningful?

Fully automatic on-the-fly MBT often generates one huge test case (cf. Subsec. 10.3.3). To be exhaustive, on-the-fly MBT must, however, recurrently restart exploration in $init_{\mathcal{S}_{det}}$ as well as restart execution in $init_{\mathbb{M}_{det}}$ (cf. Subsec. 8.8). For this, the SUT must have a reliable reset capability, which we demand by the testing hypothesis (cf. Subsec. 8.1.2). To incorporate recurrent restarts, traversalSubphase employs some **restart heuristic** to decide at the beginning of each traversal sub-phase whether to continue traversal from the current superstate or to restart.

Both $\mathcal{S} = (S, \rightarrow, L) \in \mathcal{LTS}(L_I, L_U, \tau)$ and $\mathbb{M} \in \mathcal{IOTS}_{L_I}(L_I, L_U, \tau)$ are SCCs iff they can always restart on their own, i.e., reach the initial state. This is usually the case for reactive systems. If both $\mathcal{S}$ and $\mathbb{M}$ are SCCs, on-the-fly MBT does not need to explicitly restart by resetting exploration and execution to their respective *init* states since fairness_{spec} or fairness_{test}, with underspec_U forbidden, guarantees a restart in $\mathbb{M}$ if $\mathbb{M}$ *ioco* $\mathcal{S}$ and we visit $init_{\mathcal{S}}$ sufficiently often through any cycle (cf. Lemma 8.66); the weaker fairness_{model} guarantees that for each state m in $\mathbb{M}$, there exists a cycle π in $\mathcal{S}$ such that $\mathbb{M}$ restarts from m if we cycle sufficiently often through π. If not both $\mathcal{S}$ and $\mathbb{M}$ are SCCs, or these guarantees are not strong enough in some situation, we can reify the implicit reset capability in $\mathcal{S}$: We add a new label $\boldsymbol{?r}$ to L_I, which corresponds to resetting $\mathbb{M}$. Therefore, f.a. $s \in S, s \xrightarrow{?r} init_{\mathcal{S}}$ is added to $\rightarrow$. Hence $\mathcal{S}$ and $\mathbb{M}$ are transformed to SCCs with guaranteed, synchronous resets via $?r$. Operating on SCCs, guidance heuristics subsume the restart heuristic; they can prefer fewer but longer TCs if those are more meaningful, which is often the case [Fraser et al., 2009].

11.2.5. Other Classifications

The kind of the test cases is determined by other aspects: Since all resolutions of uncontrollable nondeterminism is considered, TCs are trees. They are determined by the interplay between test generation and test execution and the phase heuristics. Due to the phases and dynamic information, the TCs are degenerated: Only one path in each phase's TC is continued, so TCs are lists modulo a given depth (cf. Fig. 11.2). Since the path follows the resolution of nondeterminism, inconclusive verdicts are avoided.

All other aspects of the MBT taxonomy given in Sec. 10.2 are independent of the LazyOTF algorithm, i.e., are not restricted:

- LazyOTF can handle any kind of system specification description language that can be used for $\mathcal{S}$ in the domain of the ioco theory, by employing for traversalSubphase an $\text{MBT}_{\not\text{exec}}$ algorithm suitable for the given specification. But if the specification contains timing constraints, they restrict the runtime resources that guidance heuristics may consume (cf. Subsec. 12.5.3) and consequently the phase heuristics. Since each traversalSubphase only traverses one sub-graph, whose depth is limited by a bound and breadth can be limited by inducing states or other heuristics (cf. Subsec. 12.3.1), LazyOTF can handle any kind and size of $\mathcal{S}$;
- LazyOTF can handle any kind of SUT that meets the testing hypothesis by using for traversalSubphase a corresponding $\text{MBT}_{\not\text{exec}}$ algorithm and a suitable configuration, including the timeout value (cf. Subsec. 8.1.2) and the employed heuristics (cf. Chapter 12);
- the level of detail in the specification of the SUT and its environment are irrelevant for traversalSubphase. But testExecutionSubphase must be able to automatically

execute the generated TCs on the SUT, which is only a restriction if MBT is used to generate TCs that are later executed manually, which we can exclude (cf. Note 10.2) and is inefficient most of the time anyways [Utting and Legeard, 2007];
- the LazyOTF algorithm works independently of whether $\mathcal{S}$ is also used for the development of the SUT, and needs not know this fact. If development does not use $\mathcal{S}$, MBT can, however, detect more failures [Pretschner and Philipps, 2005; Faragó et al., 2013], so LazyOTF's on-the-flyness (cf. Subsec. 11.2.3) becomes even more relevant.

11.3. Formalization

Having covered all aspects of MBT for LazyOTF in the last section, we can now formalize them.

11.3.1. Test Selection

Since we introduced test objectives and test goals for efficient test selection, we formalize them in this subsection.

Usually, the states of a test goal $\ddot{s}$ are also interpreted as inducing states, to feed back the dynamic information of whether $\ddot{s}$ is reached during the next testExecutionSubphase, before continuing to search for further test goals during the next traversalSubphase. This is the most sensible approach since the prime goal of finding a test goal in $\mathcal{S}$ has been achieved by the current traversal sub-phase, and the most relevant question now is whether the test execution sub-phase reaches that test goal, too. If it does, the next traversal sub-phase knows with certainty that the SUT starts in that test goal and the corresponding TOs have been discharged. We favor this greedy approach since uncontrollable nondeterministic causes high uncertainty whether a TG is really reached by the SUT, so many early and short tries are usually better than taking higher risks by performing testExecutionSubphase less frequently with larger TCs (cf. Note 12.14).

Furthermore, the dynamicInfo that phase heuristics use can usually be reduced to which TOs have already been discharged. So we also use TOs for phase heuristics, and phase heuristics and guidance heuristics are intertwined. Therefore, we also describe inducing stated with the help of TOs by defining a function $I^{lazy}(\cdot)$ that determines both TGs and inducing states, cf. Def. 11.5, resulting in Def. 11.6 for TOs. Chapter 12 describes phase heuristics and guidance heuristics, and how they use our TOs.

Definition 11.5. Let $\mathcal{S} = (S, \rightarrow, L_\tau) \in \mathit{SPEC}$ and $\mathbb{M} \in \mathit{MOD}$.

Then
- $\boldsymbol{\Sigma_{lazy}} := \{\textbf{TESTGOAL}, \textbf{INDUCING}, \textbf{ORDINARY}\}$ is a totally ordered set, with TESTGOAL $\geq$ INDUCING $\geq$ ORDINARY;
- $\boldsymbol{I^{lazy}} : S \rightarrow \Sigma_{lazy}$, i.e., $I^{lazy}(s)$ =TESTGOAL if s is in some TG, otherwise $I^{lazy}(s)$ =INDUCING if s is inducing, and otherwise ORDINARY; furthermore, $\ddot{s} = (I^{lazy})^{-1}$(TESTGOAL) and ($\ddot{s} = \varnothing$ or $P_{\ddot{s}} \cap \mathit{Straces}_{\mathbb{M}_{\tau^*\delta}}(\mathit{init}_{\mathbb{M}}) \neq \varnothing$, i.e., reaching TESTGOAL must be feasible for test execution on $\mathbb{M}$;
- to deal with nondeterminism of the LTS, $I^{lazy}(\cdot)$ is lifted to superstates: $I^{lazy} : S_{det} \rightarrow \Sigma_{lazy}, \ddot{s} \mapsto \underset{s \in \ddot{s}}{max}(I(s))$.

Definition 11.6. Let $\mathcal{S} = (S, \to, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau)$. **Test objective *o*** consists of

- $\boldsymbol{I^{lazy}} : S \to \Sigma_{lazy}$, according to Def. 11.5;
- a **guidance heuristic**, determining the meaningfulness of each path $\pi \in paths(\mathcal{S}_{det})$for o;
- ***active*** : a variable of type $\mathbb{B}$ that is **true** iff o has not yet been discharged;
- ***discharge*** : dynamicInfo $\twoheadrightarrow \mathbb{B}$, is the **discharge function for** o.
  Discharging o must be feasible; thus *discharge* must be surjective. The default is *discharge*(dynamicInfo) = **true** :$\Leftrightarrow$ I^{lazy}(current superstate) = TESTGOAL, but more complex functions are possible, e.g., for other than reachability properties.

Note. So we will never reactivate a discharged TO, even though it would technically be possible by non-monotonic discharge functions, i.e., functions that return **false** for some dynamicInfo, **true** for an extension, and again **false** for a further extension. This is interesting future work, for instance if a fault has occurred and some, but not all, discharged TOs are re-activated, so that further TCs help in finding the root cause of the fault. If dynamicInfo does not reflect the full history (cf. Subsec. 11.3.3), *discharge* might not be non-monotonic. Hence we use the variable *active*, which remains **false** once it is set to **false**, so *active* =no *discharge* until now.

Example 11.7. Exemplary TOs are:

- o_1 for testing a new coverage task, e.g., a newly added reachable state s in S: In this case, we can choose as TG $\{s\}$, i.e., $(I^{lazy})^{-1}$(TESTGOAL)= $\{s\}$, a guidance heuristic based on the TG and the default *discharge*;
- o_2 for testing a functional feature, e.g., pagination (cf. Subsec. 14.3.2): the TG contains all states that exhibit pagination, all other settings are as in o_1;
- o_3 for testing a more complex behavior that is not a reachability property, e.g., that some operation can be performed k times in succession (e.g., o_{dec} in Subsec. 14.3.3): If the states after the k operations can also be reached differently, this cannot be expressed as TG; so we set TG = $\varnothing$ and have to define an individual discharge function and guidance heuristic.

In summary, a TO o for an element (e.g., feature or requirement or coverage task) determines how to set induce states and how to guide test generation and test execution to quickly test o, and when to discharge o. Once o is discharged, it no longer has an effect, i.e., o can be discarded. Test goals simplify the specification of TOs for reachability properties since they help describe guidance heuristics, phase heuristics and discharge functions. TOs that are not based purely on TGs can flexibly describe more complex properties (using strategies and observes, cf. Chapter 13).

11.3.2. Interplay Between Test Generation And Test Execution

Due to LazyOTF's interplay between test generation and test execution, we need to formalize test generation and test execution over phases. We can formalize the whole run of LazyOTF MBT with recurrent restarts, which therefore contains all dynamic information. For this, we use a sequence of TCs and a sequence of finite paths, as defined in Def. 11.8 and depicted in Fig. 11.2.

Definition 11.8. Let $\mathcal{S} \in \mathit{SPEC}$. Then we define for the complete current run of LazyOTF MBT:

- $\boldsymbol{p_{curr}} \in \mathbb{N}_{>0} \dot\cup \{\omega\}$ as the **current number of phases**;
- $\boldsymbol{r_{curr}} \in \mathbb{N}_{>0} \dot\cup \{\omega\}$ as the **current number of restarts**, with $r_{curr} = \omega$ if $p_{curr} = \omega$ and $r_{curr} \leq p_{curr}$ otherwise;
- the strictly monotonically increasing sequence $\boldsymbol{(r_i)_{i \in [1,\dots,1+r_{curr})}}$ as the **restart sequence**, indicating which phases where directly preceded by a restart; so $r_1 = 1$ and $\forall i \in [1, \dots, 1 + r_{curr}) : r_i \leq p_{curr}$;
- the sequence $\boldsymbol{(b_i)_{i \in [1,\dots,1+p_{curr})}}$ as the **bound sequence**, describing what **dynamic bound** the corresponding traversal sub-phase used; so $\forall p \in [1, \dots, 1 + p_{curr}) : b_p \in [b_{min}, \dots, b_{max}]$;
- the sequence $\boldsymbol{(\mathbb{T}_i)_{i \in [1,\dots,1+p_{curr})}}$ as the **test case sequence** (**TC seq** for short), describing the TC the corresponding traversal sub-phase has selected. Therefore, $\forall p \in [1, \dots, 1 + p_{curr}) :$
  $\mathbb{T}_p \in \begin{cases} \text{genTS}(\mathcal{S}, \mathit{init}_{\mathcal{S}}\ \mathit{after}\ \tau, b_p) & \text{if } p \in (r_i)_i, \\ \text{genTS}(\mathcal{S}, \mathit{dest}(\pi_{p-1}), b_p) & \text{if } p \notin (r_i)_i; \end{cases}$
- the sequence $\boldsymbol{(\pi_i)_{i \in [1,\dots,1+p_{curr})}}$ as the **path sequence**, describing each path π_i the corresponding test execution sub-phase took through $\mathbb{T}_i$; so $\forall p \in [1, \dots, 1+p_{curr}) : \pi_p \in \mathit{paths}_{max}(\mathbb{T}_p)$ with $\mathit{dest}(\pi_p) \neq$ fail if ($p < p_{curr}$ and $p + 1 \notin (r_i)_i$). During the traversal sub-phase, when $\mathbb{T}_{p_{curr}}$ has not yet been constructed, the DFS stack is used as $\pi_{p_{curr}}$, i.e., the path currently considered in the current sub-graph (cf. Fig. 11.2);
- the **trace sequence** $\boldsymbol{(\sigma_i)_{i \in [1,\dots,1+p_{curr})}} := (\mathit{trace}(\pi_i))_i$;
- the sequence $\boldsymbol{(\mathbb{T}_i^{full})_{i \in [1,\dots,1+r_{curr})}}$ as the **full test case sequence** (**full TC seq** for short), which contains the same trees as the TC seq, but indicates the full test cases starting from *init* that the corresponding traversal sub-phases since the last restart up to the next restart have selected: $\forall i \in [1, \dots, r_{curr}) :$
  $\mathbb{T}_i^{full} := \prod_{p \in [r_i, \dots, r_{i+1})} \mathbb{T}_p$, and $\mathbb{T}_{r_{curr}}^{full} := \prod_{p \in [r_{r_{curr}}, \dots, p_{curr}]} \mathbb{T}_p$ if $p_{curr} < \omega$;
- the sequence $\boldsymbol{(\pi_i^{full})_{i \in [1,\dots,1+r_{curr})}}$ as the **full path sequence**, indicating each full path π_i^{full} the corresponding test execution sub-phases took through the corresponding $\mathbb{T}_i^{full}$: $\forall r \in [1, \dots, 1 + r_{curr}) : \pi_r^{full} \in \mathcal{F}_{mod}(\mathit{paths}_{max}(\mathbb{T}_r^{full} || \mathbb{M}_{\delta\tau^*}))$. During the traversal sub-phase, when $\mathbb{T}_{p_{curr}}$ has not yet been constructed, $\pi_{r_{curr}}^{full}$ is extended by the DFS stack $\pi_{p_{curr}}$, i.e., by the path currently considered in the current sub-graph (cf. Fig. 11.2);
- the set $\boldsymbol{\mathit{paths}_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}}}$ of **all full path sequences of** $\mathcal{S}$;
- the **full trace sequence** $\boldsymbol{(\sigma_i^{full})_{i \in [1,\dots,1+r_{curr})}} := (\mathit{trace}(\pi_i^{full}))_i$;
- the **number of test steps** $\boldsymbol{t_{curr}} := |(\pi_i)_{i \in [1,\dots,1+p_{curr})}| = \Sigma_{p=1}^{p_{curr}} |\pi_p|$, updated after testExecutionSubphase (i.e., without adding the DFS stack during traversal sub-phases).

Notes. Def. 11.8 uses recursion, but the structures are well defined as can be seen by induction on their complexity.

We also allow ω for the amount of test steps and phases, in spite of testing always performing only a finite number of steps. Hence we are able to use these notations also

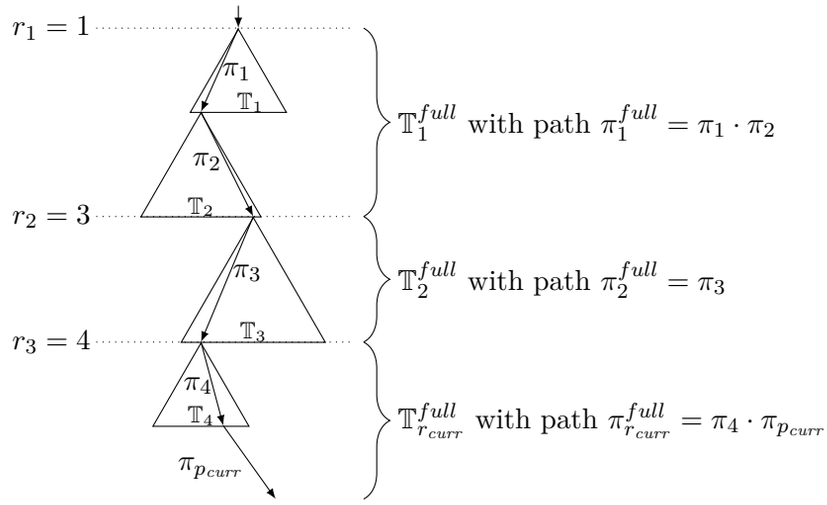


Figure 11.2.: Sequences describing the whole run of LazyOTF

for theoretically infinite runs (for exhaustiveness and $\mathrm{P}_{discharge}$, cf. Sec. 12.3).

For $\mathrm{fairness}_{test}$ or $\mathrm{fairness}_{spec}$, with forbidden $\mathrm{underspec}_U$, surjectivity of $discharge_o$ implies its feasibility (cf. Lemma 8.66). For other cases, we must additionally demand that there exists $\pi^{full} \in paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}}$ with $discharge_o(\pi^{full}) = \mathbf{true}$ and $trace(\pi^{full}) \in (Straces_{\mathbb{M}_{\tau^*\delta}}(init_{\mathbb{M}}))^{r_{curr}}$ to guarantee feasibility (cf. Def. 11.5).

For correct use of $dest(\pi_p)$ in Def. 11.8 and easier TC concatenation, we use pass states implicitly in TCs, i.e., all non- fail states that are inducing states or are reaching the bound limit have implicit outgoing transitions to pass.

Def. 11.8 determines that after a test execution fail, testing continues with a restart. This failure recovery of the tester is important in practice, e.g., for automatic nightly tests. More elaborate strategies when fail occurs would be possible (cf. Note 8.36), which is future work. Def. 11.8 still allows the full TC seq to consist of only one full TC, i.e., without restarts, as special case. But a finite or infinite single full TC can be considered as TC sequence by recurrently splitting it up nondeterministically at $init_{Spec_{det}}$ (cf. SCCs in Subsec. 11.2.4).

In summary, f.a. $i \in [1, \ldots, 1 + r_{curr})$, test execution of $\mathbb{T}_{r_i}$ **restarts** $\mathbb{M}$. The full history of all exploration sub-phases is contained in $(\mathbb{T}_i)_i$, the full history of all execution sub-phases is contained in $(\pi_i)_i$.

Def. 11.9 reduces $paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}}$ to $paths(\mathcal{S}_{det})$, for practice if the premise is met, and for theory and brevity.

Definition 11.9. Let the reliable reset capability be reified in $\mathcal{S}$ and fail (e.g., represented by $\varnothing$) be included in S_{det}. Then

- the **concatenated full path** can be defined as $\boldsymbol{\pi^{full}} := \dot{\bigcup}_{i \in [1,\ldots,1+r_{curr})} \pi_i^{full}$;
- the **concatenated full trace** can be defined as $\boldsymbol{\sigma^{full}} := \dot{\bigcup}_{i \in [1,\ldots,1+r_{curr})} \sigma_i^{full}$.

By using the concatenated full path (resp. trace) and the full path (resp. trace) sequence interchangeably, all operations defined on $paths(\mathcal{S}_{det})$ (resp. $traces(\mathcal{S}_{det})$) are lifted to $paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}}$, e.g., $|\pi^{full}| = \sum_{\pi \in (\pi_i^{full})_i} |\pi|$.

11.3.3. LazyOTF Algorithm

With the formalizations from the previous subsections, we can describe and analyze the LazyOTF algorithm in detail:

dynamicInfo can be defined to contain the full history, encoded as full test case sequence; the full path sequence of previous test execution sub-phases can be reconstructed from the full test case sequence. But often, not the full history is required by LazyOTF, only parts of the history of the execution sub-phases, so dynamicInfo can be reduced to only comprise the relevant parts of the full path sequence, e.g., $\pi^{full}_{r_{curr}}$, or even just $\pi_{p_{curr}}$.

LazyOTF can return a subset $U \subseteq \mathcal{TTS}(L_I, L_U, \delta) \times \mathbb{V}$, as defined in Listing 11.1, or dynamicInfo, which subsumes U if it is not reduced too strongly.

For traversalSubphase, extensions of genTC or genTS (cf. Sec. 8.8) that integrate phase heuristics and guidance heuristics can be used (e.g., Listing 12.2).

For testExecutionSubphase, a test adapter connects the TC with the SUT (cf. Subsec. 8.7.2) and additionally updates dynamicInfo (and checks exitCriterion), either on-the-fly or at the end of the current phase.

Theorem 11.10. *If traversalSubphase extends a sound test case generation algorithm by guidance heuristics and phase heuristics, then LazyOTF is sound.*

Proof. If traversalSubphase extends a sound (for *ioco* or some variant, cf. Chapter 9) test case generation algorithm by guidance and phase heuristics, traversalSubphase is itself sound since guidance heuristics only influences guidance, but not the construction of TCs. So the heuristics only perform test selection (cf. Chapter 12); the result is a subset of the sound genTC (cf. Theorem 8.48) and therefore itself sound.

If traversalSubphase is sound, then LazyOTF is sound, since it correctly assembles the TCs from each phase into TCs $(\mathbb{T}^{full}_i)_i$ (cf. Def. 11.8). □

Exhaustiveness, on the other hand, can be influenced by the guidance heuristics. Chapter 12 investigates conditions for the guidance heuristics to guarantee exhaustiveness and other desired properties. But exhaustiveness is not very relevant for MBT in practice: For all but trivial scenarios, an infinite test suite would be necessary, but test execution is always finite. Furthermore, having a single test case already leads to semi-decidability only: we do not know how often we have to execute it to cover all of the SUT's resolution of uncontrollable nondeterminism (cf. Subsec. 8.8.4, Subsec. 10.1.2). Therefore, guidance heuristics that prioritize meaningful test cases are very important in practice, whereas exhaustiveness is of secondary relevance. Chapter 12 covers both.

Complexity analyses for the time and space requirements of LazyOTF are difficult and vague since they depend on many aspects, especially on the applied test case generation algorithm, phase heuristics and guidance heuristics. Therefore, this paragraph only depicts which aspects, factors and dependencies need to be considered, and roughly compares LazyOTF to on-the-fly MBT and offline MBT using Landau notation. But typical complexity measures in Landau notation are not very meaningful for MBT approaches anyway (cf. Subsec. 10.2.5): the number of required overall test steps, $\boldsymbol{t^{TO}_{curr}}$, needed to achieve the desired TOs is difficult to estimate since they are strongly dependent on the efficiency of heuristics, which are hard to predict (see e.g., Subsec. 6.8.6). Sec. 14.3 will perform concrete measures on our case study for our implementation, and for on-the-fly MBT to compare their performances.

The runtime and memory of LazyOTF for t_{curr} test steps depends on the overall runtime and memory of traversalSubphase, testExecutionSubphase and exitCriterion (cf. Listing 11.1):

For traversalSubphase, time and space complexities depend on the employed test case generation algorithm and the employed heuristics. The basic test case generation algorithm is variation of the bounded offline MBT algorithm (cf. genWTS in Subsec. 12.3.4), with the worst case space complexity per traversalSubphase in $O(branch_{\mathcal{S}_{det}}^{b_{max}} \cdot (branch_{\mathcal{S}_{det}} + |S_{\to^*}|))$; the worst case time complexity per traversalSubphase is in $O(branch_{\mathcal{S}_{det}}^{b_{max}} \cdot |S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}})$, but additionally, each traversal step requires

- $|L_U|$ time for assembling the TC (cf. Subsec. 8.8.2). In our implementation, we omit output transitions to fail (cf. Subsec. 13.2.2 and Fig. 11.1), leading to $branchout_{\mathcal{S}_{det}}$ time for assembling the TC, but this is negligible in Landau notation since $branchout_{\mathcal{S}_{det}} \in O(branch_{\mathcal{S}_{\to^*}} \cdot |S_{\to^*}|)$;
- $|S_{\to^*}| + t_{curr}$ time per active TO for computing heuristics: inducingness based on $\ddot{s}$ requires $O(|S_{\to^*}|)$ time, and guidance heuristics based on π^{full} $O(t_{curr})$ time since usually only one pass over the history of the execution sub-phases is necessary, without touching individual states in superstates. (If weights are computed, each traversal step requires $branchout_{\mathcal{S}_{det}}$ time for aggregation, cf. Subsec. 12.3.4. As for assembling, this is negligible in Landau notation).

In sum, the worst case time complexity per traversalSubphase is in $O(branch_{\mathcal{S}_{det}}^{b_{max}} \cdot (|S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}} + (t_{curr} + |S_{\to^*}|) \cdot |\ddot{o}|))$.

Often smaller parts than π^{full} are sufficient, e.g., $\pi_{p_{curr}}$, or caching can be used, leading to **fast heuristics** that do not require t_{curr} time per TO, but are often in $O(b_{max} \cdot |S_{\to^*}|)$ (cf. Sec. 14.3). In this case, the worst case time complexity per traversalSubphase is in $O(branch_{\mathcal{S}_{det}}^{b_{max}} \cdot |S_{\to^*}| \cdot (branch_{\mathcal{S}_{\to^*}} + |\ddot{o}|))$. We often use efficient heuristics computations based on composition and TGs, in which case $|\ddot{o}|$ can be neglected, too.

Furthermore, the factor $|S_{\to^*}|$ is required since superstates are used, but this is a very rough estimate since large superstates are seldom and can often be restricted by inducing states (cf. Subsec. 14.3.4). In this case, the worst case time complexity per traversalSubphase is in $O(branch_{\mathcal{S}_{det}}^{b_{max}} \cdot (branch_{\mathcal{S}_{\to^*}} + |\ddot{o}| + branchout_{\mathcal{S}_{det}}))$, the worst case space complexity per traversalSubphase in $O(branch_{\mathcal{S}_{det}}^{b_{max}+1})$.

For the overall number of traversalSubphase calls, we usually require t_{curr}/b_{max} many calls or less, but in the worst case, the TCs contain one short path which is chosen during test execution, which yields $O(t_{curr})$ many calls. Consequently, the **overall worst case time complexity of all traversalSubphase** for t_{curr} test steps is in $O(branch_{\mathcal{S}_{det}}^{b_{max}} \cdot t_{curr} \cdot (|S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}} + (t_{curr} + |S_{\to^*}|) \cdot |\ddot{o}|))$, for fast heuristics in $O(branch_{\mathcal{S}_{det}}^{b_{max}} \cdot |S_{\to^*}| \cdot t_{curr} \cdot branch_{\mathcal{S}_{\to^*}})$. The **overall worst case space complexity of all traversalSubphase** for t_{curr} test steps is in $O(t_{curr} \cdot branch_{\mathcal{S}_{det}}^{b_{max}} \cdot (branch_{\mathcal{S}_{det}} + |S_{\to^*}|))$ if the whole test case sequence $(\mathbb{T}_i)_i$ is stored as dynamic feedback. For π^{full} and $\mathbb{T}_{p_{curr}}$, this reduces to $O((t_{curr} + branch_{\mathcal{S}_{det}}^{b_{max}}) \cdot (branch_{\mathcal{S}_{det}} + |S_{\to^*}|))$

Of course an increase in b_{max} can exponentially decrease t_{curr}^{TO} (cf. Subsec. 14.3.6), which is not captured by our formulas in Landau notation.

For testExecutionSubphase, a simple test adapter requires constant overall memory and constant runtime per test step to connect the TC and the SUT. Of course, the resource requirements of the SUT must be added to testExecutionSubphase's complexity. The more abstract the specification and hence the abstract test cases, the more the SUT has to perform per abstract test step.

Most dynamicInfo can also be updated in constant runtime per test step, e.g., nondeterministic resolutions. To check whether active TOs are discharged requires the resources of the *discharge* functions of the $|\ddot{o}|$ active TOs, which requires time in $O(t_{curr})$ for one pass per TO, as for the guidance heuristic. Since dynamicInfo is called $O(t_{curr})$ times, its overall worst case time complexity is in $O(t_{curr}^2 \cdot |\ddot{o}|)$. Since the history is already stored, e.g., in π^{full}, the overall worst case space complexity is lower, usually in $O(1)$. Again, for efficient implementations, e.g., based on TGs, the *discharge* functions have negligible time requirements. Then the overall worst case complexities of testExecutionSubphase are dominated by the SUT's resource requirements.

For exitCriterion that investigate the history of the execution sub-phases, the time requirement is also in $O(t_{curr})$ for one pass. Since exitCriterion is called after each phase, its overall worst case time complexity is also in $O(t_{curr}^2)$. Again the worst case space complexity is lower, e.g., $O(1)$, since the history is already stored. But usually, exitCriterion simply checks $\ddot{o} = \varnothing$, which has time and space requirements in $O(1)$ since the *discharge* functions are already computed in testExecutionSubphase.

In sum, the overall **worst case time complexity of LazyOTF** for t_{curr} test steps is in $O(t_{curr} \cdot branch_{\mathcal{S}_{det}}^{b_{max}} \cdot (|S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}} + (t_{curr} + |S_{\to^*}|) \cdot |\ddot{o}|)) + O(t_{curr}^2 \cdot |\ddot{o}|) + O(t_{curr}^2) = O(t_{curr} \cdot (t_{curr} + branch_{\mathcal{S}_{det}}^{b_{max}} \cdot (|S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}} + (t_{curr} + |S_{\to^*}|) \cdot |\ddot{o}|)))$, for typical settings in $O(t_{curr} \cdot branch_{\mathcal{S}_{det}}^{b_{max}} \cdot |S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}})$. The overall **worst case space complexity of LazyOTF** for t_{curr} test steps is in $O(t_{curr} \cdot branch_{\mathcal{S}_{det}}^{b_{max}} \cdot (branch_{\mathcal{S}_{det}} + |S_{\to^*}|))$, for typical settings in $O((t_{curr} + branch_{\mathcal{S}_{det}}^{b_{max}}) \cdot (branch_{\mathcal{S}_{det}} + |S_{\to^*}|))$. If the overall complexities should also include the SUT's resource requirements, these need to be added, which is impossible a priori since the testing hypothesis does not make sufficient performance restrictions.

The test case complexity is $|(\mathbb{T}_i)_{i \in [1,\ldots,1+p_{curr})}| \leq \Sigma_{p=1}^{p_{curr}}((|L_U|+1)^{b_p}) \leq (|L_U|+1)^{b_{max}} \cdot p_{curr} \leq (|L_U|+1)^{b_{max}} \cdot t_{curr}$, taken after testExecutionSubphase.

In our implementation, we omit output transitions to fail (cf. Subsec. 13.2.2 and Fig. 11.1), in which case we replace $(|L_U| + 1)$ with the factor $(branchout_{\mathcal{S}_{det}} + 1)$. So the **worst case test case complexity of LazyOTF** for t_{curr} test steps is in $O((branchout_{\mathcal{S}_{det}} + 1)^{b_{max}} \cdot t_{curr})$.

The efficiency of the guidance heuristics is hard to guess (cf. e.g., Subsec. 6.8.6): in the worst case, guidance heuristics attain no reduction of the overall number of required test steps t_{curr}^{TO} to achieve the desired TOs compared to offline and on-the-fly MBT (or requires even more steps). In this case, we can compare the complexity formulas for on-the-fly and offline MBT and LazyOTF:

- the overall worst case runtime complexity of LazyOTF is larger than of on-the-fly MBT, for typical setting by a factor in $O(branch_{\mathcal{S}_{det}}^{b_{max}})$, but usually smaller than of offline MBT since the factor $branch_{\mathcal{S}_{det}}^{b_{max}} \cdot t_{curr}$ is usually smaller than the factor $2^{|S_{\to^*}|}$;

- the overall worst case space complexity of LazyOTF is larger than of on-the-fly MBT, by a factor in $O(branch_{\mathcal{S}_{det}}^{b_{max}+1})$, but usually smaller than of offline MBT since the factor $(branch_{\mathcal{S}_{det}}^{b_{max}} + t_{curr})$ is usually smaller than the factor $(2^{|S_{\rightarrow^*}|})$ (and the additional space of $O(branchout_{\mathcal{S}_{det}}^{t_{curr}})$ for storing test cases);
- overall worst case test case complexity of LazyOTF is larger than of on-the-fly MBT, by a factor in $O((branchout_{\mathcal{S}_{det}} + 1)^{b_{max}})$, but usually smaller than of offline MBT since the factor $(branchout_{\mathcal{S}_{det}} + 1)^{b_{max}} \cdot t_{curr}$ is usually smaller than the factor $(branchout_{\mathcal{S}_{det}} + 1)^{t_{curr}})$.

Fortunately, the guidance heuristics often yield t_{curr}^{TO} that is linear in the minimum number of required test steps to achieve the desired TOs, i.e., the required t_{curr}^{TO} is reduced exponentially compared to offline MBT and on-the-fly MBT (see also Sec. 14.3, Subsec. 12.3.1, [Anand et al., 2013; Gay et al., 2015]).

Comparing the number of required test steps t_{curr}^{TO} to achieve the desired TOs and the worst case complexities show that usually LazyOTF has exponentially better time, space and test case complexity compared to offline MBT and on-the-fly MBT.

Due to hard predictions of the guidance heuristics and rough estimates for worst case complexities of traversalSubphase, we also compare actual runtime and memory requirements in Sec. 14.3.

11.4. Related Work

A large body of work exists about MBT in general, of which the most relevant was cited in Chapter 10. Here, we cover related work that intertwines offline MBT and on-the-fly MBT, which was conducted by two groups – both motivated by the problems that systems with uncontrollable nondeterminism cause.

[Fraser and Wotawa, 2007] is the earliest work that tries to increase the feasibility for MBT of systems with nondeterminism on output (but not nondeterminism of the LTS) by enhancing the interplay between test generation and test execution, i.e., mixing on-the-fly and offline MBT: To reduce runtime and memory overhead (cf. Chapter 10), it does not consider all resolutions of uncontrollable nondeterminism a priori. Instead, one resolution is chosen eagerly and randomly using MC's counterexample generation (cf. Sec. 5.1) via a trap property F (cf. Subsec. 10.2.3). If the SUT chooses a different resolution of nondeterminism on output, the approach does not return fail as false positive, but detects the verdict inconclusive. As alternative to returning inconclusive, a new path can be derived via a new MC phase that follows the resolution of nondeterminism on output that the SUT has just chosen. For this, variables model the resolutions of nondeterminism on output. These variables also enable measuring the coverage of nondeterminism on output during test execution (cf. Subsec. 12.3.1). An advantage of the approach is that the test generation technology is based purely on MC, so that an off-the-shelf MC tool can be applied. This adds modularity and enables the advantages of those tools, especially optimizations (cf. Sec. 5.4). Hence off-the-shelf MC tools are often used for MBT [Hamon et al., 2005; Fraser and Wotawa, 2006; Fraser, 2007; Enoiu et al., 2014]. But this also entails multiple disadvantages: off-the-shelf MC tools usually require workarounds such as model duplication or extension to generate TCs, have low usability [Fraser et al., 2009], generate an impractically large amount of redundant TCs,

even with subsequent reduction (cf. Sec. 8.8, [Fraser et al., 2009]), only create linear TCs (cf. Subsec. 10.2.4), and are usually not able to tailor heuristics towards MBT [Fraser and Wotawa, 2006, 2007; Rajan, 2009]; hence they often have test case generation with no synergetic test objectives or directives, low meaningfulness, high runtime, and state space explosion, especially in presence of nondeterminism [Rajan, 2009; Anand et al., 2013]. The variables for trap properties and for the resolutions of nondeterminism on output have to be added to the system specification description and cause a stronger explosion of the state space $\mathcal{S} = (S, T, \Sigma, I)$ during MC. The re-computations of counterexample paths via MC are performed iteratively whenever inconclusive verdicts would occur. Since long counterexample paths might have to be discarded frequently due to re-computation, the test case complexity is in $O(t_{curr}^2)$, the number of MC runs in $O(t_{curr})$. For an LTL MC run that searches the entire state space, the overall worst case time complexity of this approach is $O(t_{curr} \cdot |\mathcal{S}_{\rightarrow^*}| \cdot 2^{|F|})$ (and would be even higher with nondeterminism of the LTS, cf. Subsec. 10.2.5). The runtime could be improved by using on-the-fly MC, but without guidance heuristics, probably a large part of the state space must still be traversed until a counterexample is found (cf. Subsec. 6.8.6). Furthermore, counterexample generation via MC and trap properties cannot differentiate degrees of meaningfulness of counterexamples (cf. Subsec. 12.3.9) since MC just returns the first or all counterexamples. Finally, the check for inconclusiveness and handling nondeterminism on output via re-computations are based only on the last transition between observed states (cf. [Fraser and Wotawa, 2007, Sec. 3.1]), which does not reflect the full run so far and can cause inexhaustive MBT in case fairness_{test} is not met (cf. Def. 8.60). LazyOTF neither has the advantage nor the disadvantages.

In [Arcaini et al., 2013], the feasibility of MBT for systems with nondeterminism on output (but not nondeterminism of the LTS) is increased by combining a test driver from the field of MC with an oracle from the field of runtime conformance monitoring, i.e., monitoring (cf. Sec. 2.2) that checks conformance. For the test driver, MC via trap properties is used, leading to the deficits mentioned above, especially TCs generated independently of the SUT's resolution of uncontrollable nondeterminism. When the generated linear test case π, i.e., the counterexample path, differs from the SUT's resolution of nondeterminism on output, no re-computation occurs during the execution of π (as in [Fraser and Wotawa, 2007]). Instead, the test driver simply continues to execute the inputs of π, which is made possible by demanding the SUT be input-enabled (cf. Sec. 8.1), resulting in an execution trace σ. The monitor uses techniques from white-box testing, but could easily be transformed to purely black-box conformance testing. The monitor measures requirements coverage and is able to avoid inconclusive verdicts no matter which resolutions of nondeterminism on output the SUT chooses. But for *ioco*, σ might not be in $\mathit{faultable}(\mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}}))$ (cf. Sec. 8.5) and might not increase the coverage level. So if much nondeterminism on output is present, many executed linear test cases do not aid in conformance testing (but can still serve as stress tests). If most σ are in $\mathit{faultable}(\mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}(\mathit{init}_{\mathcal{S}}))$, they might still not aid in increasing the desired coverage. For instance, if a test target is reachable via σ_1 and σ_2, MC likely always chooses the same linear TC π; wlog π corresponds to σ_1. If σ_1 never occurs in the SUT due to underspec_U or only fairness_{model} being met (cf. Subsec. 8.8.4), recurrent execution of π never reached the test target. Besides being wasteful, this example shows that the approach is inexhaustive for SUTs with underspec_U or only fairness_{model}. Nonethe-

less, the approach does try to achieve some coverage (mainly requirements coverage) by successively and randomly picking active coverage tasks to discharge. Due to this consecutiveness and randomness, no synergy in the set of coverage tasks is achieved (cf. Subsec. 11.2.4). To focus on the more relevant coverage tasks, the authors have planned to weight coverage tasks for prioritization, as future work. LazyOTF does not have the mentioned inefficiencies. However, an interesting and important concept introduced by [Arcaini et al., 2013] is the addition of a monitor that is decoupled from the test driver: This offers a flexible oracle decoupled from TCs. For approaches that have strong guidance but not as flexible oracles as LazyOTF, monitoring can add flexibility. The monitor and input-enabledness lead to an advantage of [Arcaini et al., 2013] over LazyOTF: its real-time behavior. During the execution of a TC, there is no need to stop for any computation if the monitor runs sufficiently fast. But real-time behavior is not the focus of this thesis, so it is solely mentioned as future work (cf. Subsec. 11.6.3).

In summary, both related work [Fraser and Wotawa, 2007; Arcaini et al., 2013] perform TC generation and TC execution in isolation, which are intertwined only after each fully generated linear test case, i.e., MBT_{exec} corresponds to iteratively executing (cf. gen_{exec} in Subsec. 8.8.1):

1. $\text{MBT}_{\cancel{exec}}$ of one linear TC π;
2. test execution of (a prefix of) π.

LazyOTF does not apply off-the-shelf MC tools or technologies in isolation, but adapts, re-implements and tightly integrates the required MC algorithms (as concluded in [Fraser et al., 2009]) to efficiently generate meaningful TCs.

11.5. Parallelization

11.5.1. Introduction

The runtime of performing tests is crucial in practice: more and more testing is needed (cf. Subsec. 1.1.2), but short runtime is wanted for low cost and practicality, e.g., for nightly tests or before a product release date. Since the free lunch is over (cf. Sec. 3.5), the processing speed of tasks is now increased mainly by parallelization, which also applies for testing [Starkloff, 2000; Nupponen, 2014]. Thus concurrent testing for speedup is very relevant in practice; so it is surprising that there has not been much research on that subject [Geronimo et al., 2012; Oriol and Ullah, 2010]. Most work applies testing on the cloud, due to the rise of cloud computing and Testing as a Service (TaaS) [Bai et al., 2011; Geronimo et al., 2012; Incki et al., 2012; Vilkomir, 2012; Priyanka et al., 2012; Tilley and Parveen, 2013].

Whereas multi-threading can make use of current hardware developments on one computer (cf. the model checking example $\text{PDFS}_{\text{FIFO}}$ in Sec. 6.7) parallelization of tasks by distributed computing is more suitable for testing: each computer can run its own SUT, so that test execution is efficiently parallelized, especially for slow SUTs. Additionally, the SUT instances are independent of each other, so multiple platforms and environments can be considered and the concurrent test execution is more stable and fault-tolerant, e.g., if one system crashes due to a fault in the SUT. Therefore, most parallel testing approaches distribute test execution. The generation of TCs, however, is rarely parallelized. We firstly cover parallel test execution, then also parallel test generation.

Parallel Test Execution

Not covering test generation, the TS is created either manually, which is time-consuming, error-prone and achieves lower coverage (cf. Subsec. 1.1.2), or using offline MBT, which cannot efficiently handle large, nondeterministic specifications (cf. Chapter 10).

These parallel test execution tools usually have one master (also called centralized test controller or server machine), which is distributing TCs to slaves (also called client machines), i.e., centralized computing in a client/server network is applied. There are several tools of this kind:

Joshua [Kapfhammer, 2001] is one of the first work on distributed testing and shows that distribution can complement optimizations by test selection. Joshua is an extension of JUnit [URL:JUNIT] that uses the tuple space model [Carreiro and Gelernter, 1989] for communication with low coupling and transaction primitives, and a server to handle scheduling of TCs.

For [Lastovetsky, 2005], a new centralized test controller, called **test manager** was implemented to distribute test cases, using their own parallel language (called mpC). The experiments achieve almost linear speedup, but distribution was only performed between two machines.

Some classical testing tools have technical extensions to distribute test execution of their test cases: **Selenium (Grid)** [URL:SELENIUM], where a Selenium Grid Server distributes Selenium test cases; **HadoopUnit** [Parveen et al., 2009; Tilley and Parveen, 2012, 2013], which extends JUnit via Hadoop [URL:HADOOP; White, 2012], where each test case is a task. Experiments with a 150-node cluster show an improvement by the factor 30. As MapReduce framework, HadoopUnit cannot give real-time feedback on the testing session, but only after the reduce function (cf. Subsec. 3.5.2). As a master/slave network, the complexity [Starkloff, 2000] of TC distribution is high, especially if TCs should be scheduled efficiently and distributed via atomic transactions to not break exhaustiveness. Furthermore, have a master results in a single point of failure and fast contention. Due to these disadvantages, the following work made first steps towards decentralization for distributed test execution:

GridUnit [Duarte et al., 2005, 2006] is a JUnit extension for grid computing that does use a server, employing a Grid broker scheduler. The authors conducted experiments with 288 TCs, each one taking exactly 5 minutes, which achieved parallel efficiency between 0.64 and 0.09. They state that for shorter TCs, overhead can become large and should be avoided. Their step towards decentralization is to partition the cluster into multiple smaller clusters. But that causes further complexity of distribution and possibly unbalanced load.

[de Almeida et al., 2010] not only builds sub-groups on one level, but a **distributed tree** of instances where messages are exchanged only between parents and children. This reduces contention, but adds complexity since a good tree order needs to be found, a tree needs to be arranged and each inner tree node is a centralized test controller for its children. Additionally, this tree is less fault-tolerant and causes more contention (especially at the root node) compared to fully decentralized peer-to-peer networks.

Parallel MBT

There is even less published work on **parallel** or **distributed model-based testing** [URL:PARALLELblogpost], even though MBT is particularly well suited for parallelization:

- MBT is a kind of **high volume automated testing** [McGee and Kaner, 2004], which are techniques (e.g., random [McGee and Kaner, 2003] or genetic [Berndt and Watkins, 2005]) for automated execution and evaluation of a large number of TCs, to expose functional faults that are otherwise hard to find. These techniques are resource intensive but can effectively be distributed, for instance onto the cloud [Parveen and Tilley, 2010];
- like test execution above, test generation can also be distributed, which is important since "real world test generation problems are computationally very complex and take long to process on a single PC" [Nupponen, 2014], causing painful delays in the software development process [BusinessWire, 2009]. Since MBT is usually employed iteratively, the delays are multiplied. Hence "distributed test design processing is a key to MBT success" [Nupponen, 2014];
- with both test generation and test execution distributed, it is also possible to get rid of a master and thus avoid the problems of a master/server network (cf. Subsec. 3.5.2).

There are only four known distributed MBT tools [URL:PARALLELblogpost], which all use a master/slave network and thus are not decentralized:

Conformiq Grid [BusinessWire, 2009; Nupponen, 2014] is an extension to the offline MBT variant of Conformiq (cf. Subsec. 10.3.4), so it parallelizes test generation, but does not offer distributed test execution, which can, however, be delegated to a parallel test execution tool. Conformiq Grid offers multi-core, multiprocessing, and distributed computing, and achieves savings of up to 90% of test generation time by scaling from one to sixteen processor cores [BusinessWire, 2009; Nupponen, 2014]. It has a master/slave architecture (based on CORBA) and thus a single point of failure and contention. But the master distributes traversal tasks to the slaves, offering high performance load balancing and recovery. This allows the helpful feature of varying the number of processors while test generation is running. One focus of Conformiq Grid is deterministic test case generation, regardless of the number of processor cores, their speed, and load. This is a difficult feature, but achieves reproducibility of test case generation; in case one SUT is fixed and has no uncontrollable nondeterminism, also test coverage and bug detection are fully reproducible, which is very helpful.

Parallel QuickCheck [Kusakabe et al., 2010; Kusakabe, 2011; Wada and Kusakabe, 2012] uses property-based random testing (cf. [Fink and Bishop, 1997] and Subsec. 1.1.3) via the tool QuickCheck [Claessen and Hughes, 2000; Kusakabe et al., 2010; Kusakabe, 2011], where properties are described as Haskell functions [Marlow et al., 2010; O'Sullivan et al., 2009]. By additional libraries, QuickCheck can be extended, e.g., supporting quantifiers, conditionals and test data monitors for the properties, other languages and generators for the input. For Parallel QuickCheck, distribution is performed by Hadoop and properties are specified either in Haskell or in VDM-SL [Plat and Larsen, 1992] (both used as executable specification languages, and one specification can also test the other [Wada and Kusakabe, 2012]). TCs are generated randomly according to the

specification with the help of property-based testing. Experiments in [Kusakabe et al., 2010; Kusakabe, 2011] have shown that their distributed random TC generation produces highly redundant TCs, causing low coverage and inefficiency. Thus Parallel QuickCheck runs in two phases: The first distributedly generates TCs, which are then selected on the server. In the second phase, the server schedules TCs to the nodes, where they are executed and evaluated distributedly. Speedup is relatively good, but not linear: for Haskell specifications efficiency is 1 for 8 cores, but drops to 0.6 on 32 cores, i.e., to a speedup of less than 20.

YETI on the cloud [Oriol and Ullah, 2010] uses the automated random testing tool Yeti, which is an on-the-fly MBT tool that tests for runtime exceptions and failures or uses contracts as test oracles. Therefore, both test case generation and test execution are distributed, using Hadoop. Yeti is distributed easily using varying seeds for Yeti's pseudo-random number generator. Limited experiments were conducted and showed good scalability, but the need for exit criteria and real-time feedback [Oriol and Ullah, 2010] (which are both offered by distributed LazyOTF below). Furthermore, random testing requires many more steps than LazyOTF to achieve the same desired TOs (cf. Subsec. 14.3.4) and the long failure traces that random testing produces are impracticable and need to be reduced, which is also a difficult task. Since Yeti on the cloud only uses Hadoop to distribute the Yeti instances at the start and collect their results at the end, the deficits of the master/slave architecture are not severe.

In **PGA** [Geronimo et al., 2012; Tilley and Parveen, 2013], a parallel genetic algorithm (cf. Subsec. 12.3.1) is used to parallelize the individual fitness evaluation in each iteration using Hadoop. After sufficiently many iterations, a JUnit test suite with high branch coverage is derived. Therefore, PGA does not perform distributed test execution. The preliminary evaluation show a speedup in test case generation of 1.57 on an Intel Core i3-2100 processor, which has 2 cores and 4 threads via hyper-threading. Using a master/slave network might be avoidable if not just the fitness evaluation, but all genetic operations were parallelized.

Notes. [Schieferdecker and Vassiliou-Gioles, 2003] states that the prominent Testing and Test Control Notation version 3 (TTCN-3) [URL:TTCN] does not provide an implementation for distribution itself, but the **TTCN-3 Control Interface** (TCI) declares entities and operations that help to distribute a test system, and to test a distributed system.

The well-known Cloud9 [Ciortea et al., 2009] does not test but uses symbolic execution and execution-based model-checking. So it is not closely related to distributed MBT, but an interesting parallelization of a tool similar to those described in Subsec. 7.3.2.

11.5.2. Distributed LazyOTF

Algorithm. For LazyOTF, both the graph traversal and the test execution can be parallelized very easily using a distributed algorithm that is asynchronous and decentralized [Lynch, 1996; Raynal, 2013], baptized **distributed LazyOTF**: P nodes run in parallel, each one executing an instance of LazyOTF that tests an own instance of the given SUT. We call these LazyOTF instances $\text{LazyOTF}_1, \ldots, \text{LazyOTF}_P$. Each LazyOTF_i starts with the same set $\ddot{o}$ of TOs. Thus each $o \in \ddot{o}$ needs a common handle throughout the distributed system, which is easily implemented. If LazyOTF_i discharges some

TO $o \in \ddot{o}$, it asynchronously broadcasts the information about this so called **internal discharge** to all other instances. If LazyOTF$_i$ receives a message that some TO $o \in \ddot{o}$ has been discharged somewhere else, LazyOTF$_i$ also discharges o, called **external discharge**.

Implementation. Broadcasting internal discharges is implemented in a polymorphic method called **pushInternalDischarge**($\ddot{o}$ o). Receiving and processing these messages is implemented in a polymorphic method called **pullExternalDischarges**(). The information about an external discharge of TO $o \in \ddot{o}$ could be pushed into the local system by discharging o the moment the message arrives at LazyOTF$_i$. But this would cause high complexity within LazyOTF$_i$ due to multi-threading and race conditions if run concurrently with traversalSubphase or testExecutionSubphase. Thus pullExternalDischarges is called once at the beginning of each traversalSubphase. The disadvantage of this approach is that external discharges that happen during a traversal sub-phase or test execution sub-phase will be delayed. But they will be processed before the next traversal sub-phase is conducted. Consequently, termination detection of LazyOTF$_i$ might be delayed by one phase. Message passing is implemented by Hazelcast as default, and alternatively by UDP broadcasts and UDP multicasts (cf. Subsec. 3.5.2).

The resulting distributed LazyOTF algorithm has the same structure as Listing 11.1, with its polymorphic methods adapted by:

- traversalSubphase: calls pullExternalDischarges() at the beginning, then proceeds as usual;
- testExecutionSubphase: calls at each discharge of some TO o the method pushInternalDischarge(o), otherwise proceeds as usual.

Therefore, distributed LazyOTF distributes test execution and test generation in a shared-nothing architecture with low coupling and full decentralization. So the nodes form a peer-to-peer network, where discharges of TOs are communicated. Distributed LazyOTF offers the first decentralized parallel MBT.

Notes 11.11. The implementation uses observers for exitCriterion, so termination directly after pullExternalDischarges is possible (cf. Subsec. 13.3.4).

Distributed LazyOTF parallelizes MBT on the level of LazyOTF, with the help of heuristics. An alternative parallelization of MBT would be on a lower level, by parallelizing gen (cf. Subsec. 8.9.3). Parallel gen can implement LazyOTF such that traversal and test execution sub-phases may overlap: If the test execution phase, run in parallel, decides on the outgoing transition, all worker threads of gen traversing a state space partition that has become irrelevant are discarded; therefore, resources become available for new outgoing transitions to investigate. This approach is more suitable for real-time computation, especially since it can interpret inducing states more loosely: if resources are still available, investigation beyond inducing states can be performed before test execution reaches the inducing state. Since real-time computation is not the focus of this thesis, but LazyOTF and heuristics, we investigate distributed LazyOTF and leave this alternative parallelization as future work.

Correctness. Distributed LazyOTF is sound if all instances LazyOTF$_i$ are sound because then no instance produces a false positive. Distributed LazyOTF is exhaustive (resp.

guarantees $\mathrm{P}_{discharge}$, cf. Def. 12.6) if one instance LazyOTF$_i$ is exhaustive (resp. guarantees $\mathrm{P}_{discharge}$) since then at least that instance produces all required TCs. In summary, since all instances run the same LazyOTF algorithm, initialized with the same set $\ddot{o}$, distributed LazyOTF is sound (resp. exhaustive, resp. guarantees $\mathrm{P}_{discharge}$) if LazyOTF is sound (resp. exhaustive, resp. guarantees $\mathrm{P}_{discharge}$).

Distributed LazyOTF is fault-tolerant – more precisely, partition tolerant [Gilbert and Lynch, 2002]: if messages are lost (i.e., communication performance failures or omission failures occur [Cristian, 1993]), the runtime of distributed LazyOTF might increase, but soundness is not broken; for partitions containing an instance LazyOTF$_i$ that is exhaustive (resp. guarantees $\mathrm{P}_{discharge}$), distributed LazyOTF is also exhaustive (resp. guarantees $\mathrm{P}_{discharge}$). Since all instances run the same LazyOTF algorithm and configuration, exhaustiveness and $\mathrm{P}_{discharge}$ guarantees are also not broken by message losses.

Analysis. Because an instance does not need to discharge TOs that other instances have discharged, each instance requires fewer test steps and therefore less runtime for the same overall effect (e.g., coverage, discharging all TOs, exhaustiveness, or fault detection). With fewer test steps, counterexamples become shorter and hence easier to understand and reproduce.

Usually $\ddot{o}$ is small, so discharges take place infrequently; furthermore, information is distributed by efficient asynchronous message passing implementations; thus contention for the network resource is usually not a problem. If $\ddot{o}$ happens to be large with many $o \in \ddot{o}$ easily reachable, there might be a lot of message passing when distributed LazyOTF starts, causing network congestion. For our message passing implementations, contention for the network resource will not cause blocking or transmission retries, but simply delayed or lost messaged. Since distributed LazyOTF is partition tolerant, message losses do not break soundness, exhaustiveness or $\mathrm{P}_{discharge}$ guarantees, and increase the runtime only slightly because the discharged $o \in \ddot{o}$ whose messages got lost are easily reachable. In short, distributed LazyOTF can trade networking resources for runtime.

For SUTs with high uncontrollable nondeterminism (or $\mathcal{S}$ with multiple best choices in one traversal sub-phase, which LazyOTF can choose from, cf. Subsec. 11.1.2), two different LazyOTF$_i$ and LazyOTF$_j$ are likely to diverge quickly, reaching different parts of $\mathcal{S}$ in subsequent phases. So if the test goals used in $\ddot{o}$ are not clustered in a small region of $\mathcal{S}$, LazyOTF$_i$ and LazyOTF$_j$ will likely discharge different $o \in \ddot{o}$, resulting in little work duplication and high work pruning, and hence in good work distribution and high parallel scalability.

Though distributed LazyOTF does not explicitly distribute a test suite, the distribution criteria in [Kapfhammer, 2001] are met:

- **transparent and automatic distribution** is given since each LazyOTF$_i$ runs independently and messages are exchanged fully automatically and transparently. Only the communication (Hazelcast, multicast UDP, or broadcast UDP) needs to be configured before testing, and the instances started on the machines, e.g., with a simple script;
- **test case contamination avoidance**, i.e., execution of different TCs must be independent. This is, as usual, achieved by each instance executing TCs on an own, independent SUT;

- **test load distribution** is achieved for free by our heuristics (cf. Chapter 12), since each instance knows which TOs still have to be discharged and guides accordingly. This load balancing is fully fault-tolerant, since it can handle crash failures and is partition tolerant;
- **test suite integrity** is also given, as described in the paragraph above about correctness. Full fault-tolerance and exhaustiveness are also guaranteed;
- **test execution control**, i.e., control execution of TCs and view their results from a centralized testing environment: Since distributed LazyOTF is decentralized and TCs are generated on-the-fly with our heuristics, test execution cannot be fully controlled, only to the extent of guidance (cf. Chapter 12). With lightweight message passing via UDP broadcasts or UDP multicasts, LazyOTF instances that are started later on (e.g., to resume work after a crash) do not know which TOs have already been discharged (unless they are forked from running instances that know, which is currently not supported). Hence Hazelcast is used as default, despite its high runtime and communication costs if a Hazelcast cluster is set up and clean up frequently. Each LazyOTF_i instance can be used for real-time feedback to monitor which TOs have already been discharged globally. This and all other results are stored on a shared drive by our implementation at the end. The more heavyweight message passing via Hazelcast offers more sophisticated test execution control: Dynamic information could be monitored using the Hazelcast Management Center [Inc, 2015; Johns, 2013] for early feedback. To distribute the actual LazyOTF_i instances and the configuration onto the different machines, and to merge the results into one location, Hazelcast's distributed executor service could be used. Both is future work. Since all LazyOTF_i still communicate with each other and store their results locally, this test execution control does not break full fault-tolerance. As further Hazelcast features, instances can synchronize, communicate reliably, be started or stopped elasticly, and be deployed also to cloud services that do not support UDP broadcasts or multicasts [URL:AMAZONCLOUD]. These features are very relevant in practice [Parveen and Tilley, 2010; Priyanka et al., 2012; Tilley and Parveen, 2013; Nupponen, 2014].

Notes 11.12. Due to Hazelcast's expressiveness, future work can easily be implemented, e.g., adding new TOs lazily, i.e., when instances are already running. MapReduce on top of lightweight message passing is an alternative test execution control without Hazelcast.

In relation to the CAP theorem [Gilbert and Lynch, 2002], distributed LazyOTF hence AP: Available and Partition tolerant, whereas Consistency is only achieved eventually, without a time bound, by additional local computation to discharge the lost $o \in \ddot{o}$. Even with Hazelcast and it's distributed backups, distributed LazyOTF is still AP since Hazelcast is AP.

Distributed LazyOTF can also use multi-core multiprocessing, but a tailored multi-threaded parallelization (cf. Sec. 3.5) is likely more efficient for these cases than distributed LazyOTF, but future work. though more efficient implementations would be possible, cf. Note 11.11). A multi-core variant of LazyOTF is especial useful if all instances should operate on one SUT, or multiple SUTs can run independently and quickly on one computer.

Distributed LazyOTF can also be used to test a distributed SUT: use this SUT only once instead of the n independent instances, and place a LazyOTF instance at each port of the SUT (i.e., each distributed point of control and observation). Because of LazyOTF's flexible implementation of TOs (cf. Subsec. 13.2.3), arbitrarily complex distributed situations can be tested (e.g., *ioco*, *mioco* [Tretmans, 2008], *dioco* [Hierons et al., 2012], *eco* [Frantzen, 2016] or coverage criteria for cloud computing [Chan et al., 2009]). Until recently, there has been little progress on testing large-scale distributed systems [Hughes et al., 2004]; one reason is that inefficient distributed testing (e.g., with a master and contention) can strongly distort the distributed behavior of the SUT [de Almeida et al., 2010; Long and Strooper, 2001]. Our decentralized distributed testing depicts a solution.

11.6. Conclusion

11.6.1. Summary

This chapter investigated the interplay between test generation and test execution and motivated a lazy alternative to on-the-fly MBT, called LazyOTF. It synergetically integrates the advantages of both on-the-fly and offline MBT: test steps are still executed on-the-fly during test case generation, but not strictly in lockstep, but lazily when there is a reason to, encoded as heuristic.

As in offline MBT, backtracking is possible, now within each phase's sub-graph bounded by the phase heuristics. So the test selection heuristics can search the complete sub-graph to choose the potentially most meaningful TC, but unlike offline MBT, LazyOTF can incorporate the dynamic information from previous phases. This enables new guidance heuristics, designed and investigated in detail in Chapter 12. In this chapter, the overview of the LazyOTF approach, algorithm, and heuristics framework is given according to the MBT taxonomy, with a focus on test selection and on the interplay between test generation and test execution. Both are formalized to enable precise descriptions and analyses. We thoroughly consider related work to LazyOTF and to its parallelization, called distributed LazyOTF, which is also introduced.

To express LazyOTF and its laziness as a scheduling of tasks (cf. Def. 3.33), we partition transition tasks into intra-phase transition tasks, i.e., they stay within a sub-phase, and inter-phase transition tasks, i.e., they leave a sub-phase (they leave an inducing state or exceed a bound). Check tasks perform a test step execution on the SUT and check the result of that test step. LazyOTF lazily schedules inter-phase transition tasks only after check task only after intra-phase transition task. Hence it gives up strict on-the-flyness, i.e., level 2 OTF, for better guidance in a level 1 OTF algorithm.

LazyOTF's advantages over offline and on-the-fly MBT are mainly its overall time, space and test case complexities, which are usually exponentially smaller, and the higher meaningfulness of the generated TCs. For slow SUTs, meaningful TCs become even more important, since test case execution becomes the bottleneck. Further advantages in the context of the whole software development process are: stronger on-the-flyness, short and understandable TCs, higher reproducibility, and automatic traceability (cf. Subsec. 11.1.2). Furthermore, LazyOTF offers flexible heuristics via our TOs, which also enables efficient parallelization: instead of distributing TCs (cf. Subsec. 11.5.1), distributed LazyOTF distributes discharging of TOs. This leads to (super-)linear speedup

of the test execution sub-phase and of meaningfulness, almost linear speedup overall (cf. Subsec. 14.3.7), and to the first decentralized MBT, in a peer-to-peer network with full fault-tolerance (partition tolerance and crash failures) and less communication and contention. Finally, LazyOTF also enables further optimizations and improvements, especially via its flexible heuristics framework. In sum, LazyOTF makes MBT of large systems with uncontrollable nondeterminism feasible, which is a major challenge [Huima, 2007]. These advantages will be substantiated by our experiments in Sec. 14.3. Unfortunately, LazyOTF if more complicated than OTF, which simply chooses randomly. Consequently, LazyOTF must tightly integrate MC algorithms with heuristics to make strong use of the available dynamic information. Hence off-the-shelf MC tool cannot be applied efficiently without adaption within LazyOTF. Furthermore, there are many aspects of MBT and possibilities for further heuristics, which are future work. For instance, real-time MBT of timed automata via LazyOTF is a challenge (cf. Subsec. 11.6.3).

11.6.2. Contributions

The main contribution of this chapter is the design and implementation of LazyOTF to increase the feasibility of MBT. We formalized and investigated various aspects of LazyOTF, showed that many aspects of the MBT taxonomy (cf. Sec. 10.2) can be varied within LazyOTF, and analyzed LazyOTF's complexity. We designed and implemented distributed LazyOTF, as exemplary optimization.

11.6.3. Future

Possible future work on LazyOTF includes:

- more elaborate handling of failures: currently, testing stops or restarts when a fail occurs, but coping with failures in a more elaborate way, e.g., via fault-tolerance, could result in more efficient testing: After a failure occurred, it could be taken into account to avoid the same failure again, but find related ones (cf. Note 8.36, Subsec. 8.9.3, and Subsec. 11.1.2). Since these failure handling methods are difficult, require additional constraints, and should be evaluated empirically, they are future work;
- adapting LazyOTF for real-time behavior and testing time automata is possible at different levels: real-time behavior is supported by parallelizing gen (cf. Subsec. 8.9.3, Subsec. 11.5.2, Subsec. 11.4, and below); furthermore, heuristics can be greedy (cf. Subsec. 12.5.3). But this is future work since its efficiency is an open question and it is not the focus of this thesis;
- implementing test selection strategies via mutation testing and model-based mutation testing, and comparing them to the test selection strategies LazyOTF already offers (cf. Note 11.4).

Possible future work on distributed LazyOTF includes:

- implement parallel gen (cf. Subsec. 11.5.2), and then overlapping traversal and test execution sub-phases. Finally, compare the results to distributed LazyOTF;
- due to Hazelcast's expressiveness, many interesting features can easily be implemented and experimented with as future work. For instance, instances can add new TOs lazily. Furthermore, using shell scripts to distribute LazyOTF_i instances onto

machines and to merge their results into one location became unhandy quickly, so applying Hazelcast's distributed executor service as technical solution for deployment and for collecting and post-processing of results is some future work that should cause no difficulties but could strongly simplify test execution control. Since message passing is still used during the execution of all LazyOTF$_i$, full fault-tolerance and many additional features of test execution control, e.g., monitoring, are still applicable;

- pushing the message of the external discharge o into the local system by discharging o in $\ddot{o}$ the moment the message arrives at LazyOTF$_i$, instead of receiving and processing these messages in pullExternalDischarges just before traversal sub-phases. But as described in Subsec. 11.5.2 and suggested by our experiments (cf. Subsec. 14.3.7), pushing will probably not lead to a strong improvement over pulling;
- using distributed LazyOTF to test a distributed SUT (cf. Note 11.12 and Subsec. 14.3.8).

12. Heuristics for Lazy On-the-fly MBT

12.1. Introduction

Heuristics are techniques that follow some general guideline that does not always hold, i.e., is only an approximation to the optimum. Consequently, they trade optimality for lower cost to find a sufficiently satisfactory solution within an acceptable amount of resources (such as time, space and information). Before introducing heuristics for phases (cf. Sec. 12.2) and guidance (cf. Sec. 12.3), this section motivates the use of heuristics and introduces classical heuristics used for MBT.

As Chapter 8 and Chapter 10 have shown, MBT is usually not exhaustive since exhaustive enumeration of all test cases is usually not feasible. Furthermore, it is unclear how often a test case needs to be executed because of uncontrollable nondeterminism. Therefore, MBT needs to perform **test selection**, i.e., choose a usually inexhaustive but finite sequence of TCs $(\mathbb{T}_i)_i$ to execute (cf. Def. 11.8). The standard ioco theory does not deal with test selection. Our ioco theory in Chapter 8 did cover test selection within exhaustive test suites by only considering faultable states and faultable traces and by choosing TCs with an appropriate bound. But in most cases, the infeasibility problems just described do hold (cf. Table 8.1). Then inexhaustive test selection is necessary and not all possible faults can be found by $(\mathbb{T}_i)_i$. Furthermore, it is usually too difficult to chose $(\mathbb{T}_i)_i$ optimally, i.e., most meaningful (cf. Subsec. 2.5) within that amount of test steps, because

- we do not know in advance what resolution of uncontrollable nondeterminism the SUT will take, so we cannot choose in advance the TC that will be the most meaningful during test execution;
- therefore, we can only try to select a potentially most meaningful TC, i.e., most meaningful under all nondeterministic resolutions. But this is difficult, too, since meaningfulness depends on the domain, user scenario and environment, which we do not know in advance, and on the implementation of the SUT, which is unknown in black-box testing. Thus **meaningfulness heuristics** are used to approximate meaningfulness. These heuristics are user-supplied, e.g., coverage criteria, test purposes or more generally test objectives (cf. Subsec. 11.2.4) to estimate how meaningful a test case is;
- OTF and LazyOTF perform test execution before the specification $\mathcal{S}$ has been (fully) explored. So test selection is necessary before further information of $\mathcal{S}$ is available. But an optimal test case of size s_1 might not be an extension of an optimal test case of size $s_2 < s_1$;
- so test selection needs to be flexible enough to subsume some meaningfulness heuristic. State coverage is one of the simplest relevant heuristics, which is sufficiently strong to guarantee exhaustiveness only for the strongest fairness constraint fairness_{test} (cf. Subsec. 8.8.4). Nonetheless, optimal state coverage is already NP

hard since the traveling salesman problem can be reduced to it [Mosk-Aoyama and Yannakakis, 2005; Swain et al., 2012];
- test selection can be improved by taking **dynamic information** into account, i.e., information gained during a test run of the SUT, e.g., the uncontrollable nondeterministic resolution. But hereby, test selection becomes even more elaborate.

Fortunately, the optimal TCs are not required for testing, some suboptimal TCs also detect the errors if they are sufficiently meaningful (and failure traces sufficiently short for the verification engineer to understand). Thus we search for a usually suboptimal but still sufficient solution to meet some requirement, using heuristics (cf. Fig. 12.4 on page 322).

Note. Optimal test selection lies within NP for most meaningfulness heuristics: genTC (cf. Listing 8.2) can nondeterministically guess a TC that meets the heuristics in polynomial time, and for most heuristics it can be checked in polynomial time whether the TC meets the heuristics' criterion (e.g., some coverage or test objectives).

Roadmap. Sec. 12.2 introduces phase heuristics: user-supplied inducing states in Subsec. 12.2.1, bound heuristics in Subsec. 12.2.2. Sec. 12.3 introduces flexible guidance heuristics for test selection: Subsec. 12.3.1 enumerates related work on guidance heuristics, for offline MBT and on-the-fly MBT, and depicts how LazyOTF can make better use of guidance heuristics. Subsec. 12.3.2, resp. Subsec. 12.3.3, introduce provisos that guarantee exhaustiveness, resp. discharging of TOs, for our heuristics. Subsec. 12.3.4 design flexible heuristics based on weights, and a corresponding test generation algorithm. Subsec. 12.3.5, resp. Subsec. 12.3.6, investigate our provisos to guarantee exhaustiveness, resp. discharging of TOs, for our weight heuristics. Subsec. 12.3.7 covers compositionality of TOs for our weight heuristics, and how they meet our provisos. Subsec. 12.3.8 investigates when provisos are not met, and countermeasures. Subsec. 12.3.9 lists related work on guidance heuristics via weights. Sec. 12.4 introduces optimizations for our heuristics: Subsec. 12.4.1 lazy traversal sub-phases to optimize performance via phase heuristics, Subsec. 12.4.2 eager micro-traversal sub-phases to optimize meaningfulness via guidance heuristics, Subsec. 12.4.3 covers optimization of reproducibility, Subsec. 12.4.4 considers more dynamic information for bound heuristics as future work, and Subsec. 12.4.5 investigates quantifying nondeterminism for better meaningfulness as future work. Sec. 12.5 concludes this chapter, summarizing our provisos in Fig. 12.4.

12.2. Phase Heuristics

Since LazyOTF integrates on-the-fly and offline MBT, it has the new feature of swapping flexibly between traversal sub-phases and test execution sub-phases, leading to new heuristics neither present in on-the-fly nor in offline MBT, the so called phase heuristics (cf. Subsec. 11.3.2). They determine how LazyOTF divides $\mathcal{S}$ into (possibly overlapping) sub-graphs, where each traversal sub-phase traverses one of them (cf. Subsec. 11.1.2). Consequently, the phase heuristics control LazyOTF's scheduling of tasks (cf. Def. 3.33 and Subsec. 11.6.1).

A phase heuristic can

- reduce the overall complexity of a traversal sub-phase as well as execution sub-phase;
- control where and when dynamic information is fed back to test selection, by determining where and when traversal sub-phases and execution sub-phases are swapped;
- reduce uncontrollable nondeterminism in TCs, and thus their complexity, by postponing exploration of nondeterministic choices until the dynamic information of the actual choice is fed back to the next traversal sub-phase.

The advantage and effectiveness of phase heuristics strongly depend on the specification and application domain. Thus they are supplied by the verification engineers, either explicitly or via heuristics, in the form of inducing states and bounds $b_{p_{curr}}$ (cf. Def. 11.8). l.10 of Listing 12.2 on page 300 integrates LazyOTF's phase heuristics: **if** $(b == 0$ or $isInducing(\pi))$.

12.2.1. Inducing States

Inducing states are formalized by $I^{lazy}(\cdot)$ (cf. Def. 11.5). $isInducing(\pi)$ is usually implemented via test objectives by **return** $(I^{lazy}(dest(\pi))$ == INDUCING) && $(|\pi_{p_{curr}}|$!= 0); (forcing at least one test step, cf. Note 11.2).

Inducing states are user-supplied and either enumerated explicitly in the specification or in the GUI (cf. Subsec. 13.3.2), or declared programmatically (see Subsec. 13.2.3), e.g., all states with a name in *Exception*, since exceptions cause nondeterminism on output.

12.2.2. Bound Settings and Heuristics

Since inducing states do not guarantee that the sub-graphs for the traversal sub-phases will be sufficiently small for the phase heuristics to be efficient, a **depth bound** $b_{p_{curr}}$ restricts the depth of $\mathbb{T}_{p_{curr}}$, i.e., of the sub-graph's computation tree (cf. Def. 11.8). $b_{p_{curr}}$ can be constant for all phases, or incorporate dynamic information to increase efficiency (cf. Sec. 14.3.6), so that the bound sequence $(b_i)_{i\in[1,\ldots,1+p_{curr})}$ may contain different values. This is implemented by our dynamic bound heuristics, defined in Def. 12.1. The **minimal depth bound**, $\boldsymbol{b_{min}}$, the **maximal depth bound**, $\boldsymbol{b_{max}}$, the **depth bound thresholds**, $\boldsymbol{p_+}$ and $\boldsymbol{p_-}$, the **depth bound increment function**, $\boldsymbol{b_+(\cdot)}$, and the **depth bound decrement function**, $\boldsymbol{b_-(\cdot)}$ are user-supplied (cf. Subsec. 14.3.6), and $b_1 := b_{min}$.

Definition 12.1. Let $b_{min}, b_{max}, p_+, p_- \in \mathbb{N}_{>0}$ and $b_+, b_- : [b_{min}, \ldots, b_{max}] \to [b_{min}, \ldots, b_{max}]$ be given. Then the **dynamic bound heuristics** determines $b_{1+p_{curr}}$ (depending on $(b_i)_{i\in[1,\ldots,1+p_{curr})}$ and $(\pi_i)_{i\in[1,\ldots,1+p_{curr})}$):

$$b_{1+p_{curr}} := \begin{cases} b_+(b_{p_{curr}}) & \text{if } \forall i \in [0,\ldots,p_+-1] : (b_{p_{curr}-i} = b_{p_{curr}} \text{ and} \\ & \underline{\text{testExecutionSubphase}} \text{ in phase } p_{curr}-i \text{ discharged no TO}); \\ b_-(b_{p_{curr}}) & \text{if } \forall i \in [0,\ldots,p_--1] : (b_{p_{curr}-i} = b_{p_{curr}} \text{ and} \\ & \underline{\text{testExecutionSubphase}} \text{ in phase } p_{curr}-i \text{ discharged a TO}); \\ b_{p_{curr}} & \text{otherwise.} \end{cases}$$

Notes. Our dynamic bound heuristics depend on the user-supplied values and the dynamic information of when TOs are discharged. By this restriction, our dynamic bound heuristics are easily configurable and efficiently computable. Furthermore, discharges are the most relevant information for the bound heuristics since discharging TOs is a main goal of testing with LazyOTF. In rare cases, considering further dynamic information might yield better bound heuristics, which is considered in Subsec. 12.4.4.

An increment in $b_{p_{curr}}$ improves meaningfullness of the generated TCs, a decrement improves the performance of traversalSubphase. So supporting increases and decreases of $b_{p_{curr}}$ allows fast and aggressive dynamic bound heuristics since $b_{p_{curr}}$ can be corrected in both directions later on. Furthermore, there might be an important but hard to reach TO, and once it is discharged, less important but easier to reach TOs call for smaller $b_{p_{curr}}$ (see next section).

12.3. Guidance Heuristics

12.3.1. Introduction

Whereas phase heuristics determine the current sub-graph (cf. previous section), **guidance heuristics** resolve the controllable nondeterministic choices of the abstract test case generation algorithm genTC in Listing 8.2:

- whether to offer an input or wait;
- which input to pick;
- whether to restart (cf. Subsec. 11.2.4);
- whether, respectively when, to terminate (cf. exitCriterion in Listing 11.1).

So guidance heuristics determine $i_\delta \in L_I \dot\cup \{\delta\}$ in each step and thus the TCs. Therefore, guidance heuristics are **test selection heuristics** to find potentially meaningful test cases. The importance of test selection has been shown empirically, e.g., in [Jard and Jéron, 2005; Lackner and Schlingloff, 2012].

Related Work on Guidance Heuristics

The following **classical guidance heuristics**, also called **test selection directives** (**test directive** for short), are used in practice as classical test selection heuristics:

- randomness, i.e., a random walk over the graph is performed, which corresponds to TCs being selected randomly from genTC$(\cdot, \textit{init after } \tau)$ (which can be considered the weakest possible heuristic since no information is used to guide selection). Random choice is not directed: if a state s of a specification $\mathcal{S}$ has a large set $in_{\mathcal{S}_{\tau^*}}(s)$ and few lead to meaningful states or traces, random selection likely leads to bad results [Nieminen et al., 2011];
- a user-supplied **test purpose** p: p is a set of $Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$ to be considered (formally, p is a linear time property (cf. Subsec. 4.2.2) with respect to $L_I \dot\cup L_U \dot\cup \{\delta\}$). p is usually described by a regular expression or by a deterministic LTS in $\mathcal{IOTS}_{L_I \dot\cup L_U}(L_I, L_U)$ with two special states pass and inconclusive (roughly similar to a test case, cf. Def. 8.34). p restricts the exploration of $\mathcal{S}$ to the behaviors described by both p and $\mathcal{S}$ (similarly to never claims in SPIN, cf. 3.4.3). So a TC $\mathbb{T}$ generated with the help of p only considers behaviors in p. Thus test

execution of $\mathbb{T}$ yields the verdict inconclusive if the SUT's behavior departs from p due to uncontrollable nondeterminism. Unfortunately, the stronger the state space is decreased by p, the more likely the verdict inconclusive. Furthermore, describing test purposes is a manual task that is error-prone, requires time and expertise [Rusu et al., 2000; Jard and Jéron, 2005], and requires high maintenance whenever $\mathcal{S}$ changes. Finally, test purposes are not compositional as they cannot be combined efficiently: conjunction might not be feasible, concatenation inefficient since some order needs to be determined a priori, and possible redundancy removed manually to achieve synergetic effects (cf. Subsec. 11.2.4). Disjunction is used in [de Vries and Tretmans, 2001] (by flattening the set of test purposes called **plural observation objective**) and also inefficient, as this further weakens guidance. For instance for full transition coverage, the resulting observation objective is simply the specification itself, resulting in testing as if no observation objective was given. While this eventually achieves full transition coverage, it causes no performance improvement at all. Test purposes are also called **singular observation objectives**, **scenarios**, **trace patterns** (e.g., in Spec Explorer [Veanes et al., 2008; Jiang et al., 2011]) and **usage profiles** if they are enriched by probability values (cf. Subsec. 5.5.4 and [Seifert et al., 2008]);

- a user-supplied **coverage criterion** c (cf. Sec. 2.5), such that test selection makes the choice that is best for c: the choice to (potentially) raise the coverage level of c the most, if such a choice exists. Otherwise, the choice should help future selections to raise c the most. So the choice ($\pi_{p_{curr}}$ in LazyOTF's terminology, cf. Def. 11.8) is derived from the history ($(\pi_i^{full})_i$ in LazyOTF's terminology), i.e., what has been covered so far. This guidance is often restricted to certain structural coverage criteria and hard to transfer to other guidance heuristics, e.g., for other test selection directives, for requirements, for composition, and for systems with uncontrollable nondeterminism. It depends on the situation and the coverage criterion c (or mix of coverage and other criteria [Duran and Ntafos, 1984; Abdurazik et al., 2000; Dupuy and Leveson, 2000; Rajan et al., 2008b; Krishnan et al., 2012; Anand et al., 2013; Gay et al., 2015]) whether c selects meaningful test cases or not: In [Staats et al., 2012; Pretschner et al., 2013], c rarely selects meaningful test cases, in [Heimdahl et al., 2004], simpler specifications coverage criteria rarely selects meaningful test cases, in [Weyuker and Jeng, 1991; Frankl et al., 1997; Juzgado et al., 2004; Cadar et al., 2008b; Fraser et al., 2009; Krishnan et al., 2012; Gay et al., 2015] it strongly depends on the situation, and in [Horgan et al., 1994; Gutjahr, 1999; Fraser and Wotawa, 2006; Mockus et al., 2009; Derderian et al., 2006; Weißleder, 2009; Ali et al., 2010; Utting and Legeard, 2007; Godefroid et al., 2005] c supports the meaningfulness of test cases. The most relevant aspect is whether uncontrollable nondeterminism is present: if it is, static coverage levels on $\mathcal{S}$ during traversal can deviate dramatically from the dynamic coverage levels during test execution (cf. Sec. 2.5).

In iterative software development, more directed test selection like test purposes are often better suited than coverage criteria, e.g., to prioritize TCs for new or fixed behaviors (cf. 14.2). But better suited test selections should also be better combinable and more flexible than test purposes, like coverage criteria.

Two other test selection heuristics are introduced in [Feijs et al., 2002; Goga, 2003], which are meant to be used simultaneously:

- a user-supplied **reduction heuristic**, which assumes that few outgoing transitions of a state already show all relevant differing behavior, and hence reduces the number of outgoing transitions via user-supplied transition selection;
- a **cycling heuristic**, which assumes that few unwindings of a cycle (like loop unwindings in Subsec. 7.1.1) show all relevant differing behavior (cf. small scope hypothesis in Subsec. 6.4.3), and hence prioritizes test cases that unwind cycles only a few times. So the cycling heuristics recurrently allows more and more loop unwindings (like BMC with a bound check, cf. Subsec. 5.2.3). But sometimes there is no exhaustiveness threshold for the maximal number of required loop unwindings (cf. Lemma 8.74), unlike BMC's completeness threshold.

Unfortunately, the cycling heuristic is based on a representation of traces that uses cycle enumeration, which is costly [Feijs et al., 2002; Johnson, 1975]. Furthermore, the heuristics work offline, operating on sets of paths of $\mathcal{S}_{det}$. [Goga, 2003] implements the heuristics as proof of concept in an offline approach by firstly unfolding a set of traces from $\mathcal{S}$ and then performing test selection on the set. [Feijs et al., 2002] raises the question whether an on-the-fly implementation of the heuristics on $\mathcal{S}$ is possible. The guidance heuristics introduced in this section for LazyOTF can be used to implement on-the-fly variants of the cycling heuristic, as future work (cf. Subsec. 12.3.9 and Subsec. 12.5.3).

In **search-based software testing** (**SBST**) [Ali et al., 2010; Anand et al., 2013; Su et al., 2015b], classical metaheuristics (e.g., evolutionary algorithms, simulated annealing and hill climbing) are used to maximize the meaningfulness of TCs while minimizing the cost, so the algorithms search for good TCs in the specification.The approaches are often not as expressive and flexible as our heuristics, focus only on coverage criteria, and do not handle oracles well (i.e., deal with input better than output) [Anand et al., 2013]. Many approaches search for a suitable TS of the full state space, i.e., search within the set $2^{\mathcal{TTS}(L_I,L_U,\delta)}$, which quickly becomes infeasible, especially for large, nondeterministic systems.

For instance, **genetic algorithms** (cf. GPA in Sec.11.5.1 and [Geronimo et al., 2012; Fraser and Arcuri, 2011]) iteratively pick the fittest solutions (called survival selection) out of the population and create new solutions by applying reproduction operators (i.e., crossover and mutation). So evolutionary algorithms require a fitness evaluation, which is difficult to choose, requires an initial population (e.g., of random solutions), and is not as flexible (especially for composition) and expressive as LazyOTF's heuristics. LazyOTF's heuristics implementation via weights (cf. Subsec. 12.3.4) could, however, be used as fitness evaluation, e.g., using a larger bound and no full traversal within sub-graphs, but a genetic algorithm.

Genetic algorithms with a fitness evaluation that executes the TSs to measure some coverage criterion only make sense for generating a TS for later regression testing, since all TCs in the reproduction selection have already been executed. For large specifications with uncontrollable nondeterminism, it would have to be investigated whether genetic algorithms are able to implicitly quantify nondeterminism, i.e., that the evolution adapts to the SUT's probability distribution of nondeterministic resolutions, and whether that is possible with a feasible number of iterations. If so, the resulting TS might achieve

sufficient coverage if the SUT's nondeterministic resolution tends to favor specific choices. Otherwise, the resulting TS cannot achieve a coverage as high as LazyOTF, which makes use of dynamic information.

Several of these guidance heuristics are based on some metric, such as the coverage level, the number of cycle unwindings, or the fitness evaluation. These guidance heuristics are sometimes called metric-driven [Anand et al., 2013].

Guidance Heuristics for Offline MBT

Heuristics for offline MBT (cf. Subsec. 10.2.5) can be used for test selection while exploring the specification $\mathcal{S}$, but without dynamic information: since offline MBT is generating TCs a priori, it has no dynamic information from test execution yet, but it can investigate all of $\mathcal{S}$ – or at least parts of $\mathcal{S}$ if $\mathcal{S}$ is too large for full exploration. To select TCs from the explored behaviors of $\mathcal{S}$, offline MBT uses any of the three classical test selection heuristics; newer approaches also allow a combination of them, e.g., the tool TGV (cf. Subsec. 10.2.5) now supports a coverage criterion restricted by a test purpose [Jard and Jéron, 2005]. Furthermore, since $\mathcal{S}$ can cause state space explosion during traversal, offline MBT can restrict the explored behaviors (and thus the TCs) by a bound b_1 on the depth (which is a special case of our bound heuristics, cf. Subsec. 12.2.2).

Besides the deficits already described for each classical test selection heuristic (cf. Subsec 10.2.5 and enumeration above), feasibility is at risk since offline test selection heuristics often generate or operate on a huge test suite $\ddot{\mathbb{T}}$ [Fraser et al., 2009] due to uncontrollable nondeterminism and the lack of dynamic information. Classical guidance heuristics without dynamic information are highly uncertain, e.g., a static coverage level during traversal of $\mathcal{S}$ might strongly deviate from the dynamic coverage level the SUT later exhibits during test execution (cf. Sec. 2.5). Furthermore, if the resolution of uncontrollable nondeterminism during test execution corresponds to only a small part of $\ddot{\mathbb{T}}$, most work for generating $\ddot{\mathbb{T}}$ is unnecessary.

Guidance Heuristics for OTF

On-the-fly MBT mainly uses randomization. As mentioned above, random guidance is a weak test selection since it does not avoid unnecessary cases. Even worse, some $i \in in_{\mathcal{S}_{\tau^*}}(s)$ might move far away from meaningful states or traces that have almost been reached, rendering the previous test steps void (cf. Sec. 14.3). Test purposes can also be used for OTF, with all disadvantages mentioned above. Employing coverage criteria for guidance is too difficult, because OTF only considers the transitions $in_{\mathcal{S}_{\tau^*}}(s) \dot{\cup} \{\tau\}$, not what occurs after that one step. Consequently, OTF usually does not know which transition will help increase the coverage criterion. Of course the coverage level can still be measured to inform the test engineer or as exit criterion.

Guidance Heuristics for LazyOTF

Differing from OTF, LazyOTF needs not perform test selection strictly on-the-fly in each step, but lazily within the current traversal sub-phase. Thus backtracking can be used to search for a potentially best choices i_δ of controllable nondeterminism within the current

sub-graph of $\mathcal{S}$. Backtracking can also make use of dynamic information gathered from previous phases. This new combination enables new heuristics.

Note. So on an abstract level, LazyOTF can itself be considered a guidance heuristic. LazyOTF can also be considered a metaheuristic: it makes use of lower-level heuristics for

- deciding in which states traversal of $\mathcal{S}$ should be postponed in favor of test execution (cf. phase heuristics in Sec. 12.2);
- deciding for individual paths how meaningful they are, (e.g., based on meaningfulness weight heuristics, cf. Subsec. 12.3.4);
- deducing from individual paths how meaningful complete test cases are (e.g., based on heuristics that aggregate weights, cf. Subsec. 12.3.6).

In summary, LazyOTF makes more meaningful choices with the help of guidance heuristics that search in a traversal sub-phase within the current sub-graph, pruned by the depth bound $b_{p_{curr}}$ and by inducing states. If a TO can be reached within $b_{p_{curr}}$ steps, the guidance heuristics can definitely construct a TC $\mathbb{T}_{p_{curr}}$ with a TO. If TOs are further away, test execution can still be guided such that the likelihood of a quick discharge of a TO is higher than for random choices. Sec. 14.3 will give examples. The guidance heuristics can use dynamic information from previous phases.

This also enables guidance via dynamic coverage criteria, which are measured during test execution. New criteria for **coverage of uncontrollable nondeterminism** can also be designed: For instance, TOs can incorporate nondeterminism of the LTS (cf. Subsec. 8.8.4, Subsec. 11.2.4). Furthermore, the following coverage criteria [Faragó, 2011] stipulate that each choice point of nondeterminism on output

- is visited at least once, baptized **1-choice state coverage**;
- can pursue different choices, baptized **2-choices state coverage**;
- makes at least n choices if the specification offers at least n choices, baptized **n-choices state coverage**;
- covers all its choices, baptized **all-choices state coverage**.

One possible implementation of these coverage criteria for uncontrollable nondeterminism is described in Subsec. 12.3.5. Combined with classical coverage criteria, 1-choice state coverage, 2-choices state coverage and all-choices state and transition coverage are a proper superset of the coverage criteria described in [Fraser and Wotawa, 2007].

Our guidance heuristics have several advantages over classical guidance heuristics, as the remainder of this section will show: they subsume all classical test selection heuristics, are simpler to specify than test purposes, compositional, more expressive, and yield more meaningful TCs. They are particularly useful for user-supplied meaningfulness heuristics for specific features. Hence test objectives are a natural fit for meaningfulness in practice (such as iterative software development processes, cf. Sec. 14.2).

Notes. Another example of a possible new guidance that can make use of dynamic information is the integration of mutation testing (cf. Notes 11.4); even for systems with uncontrollable nondeterminism, guidance can dynamically adapt to mutants being killed.

For infinitely branching states (cf. Note 8.44), LazyOTF requires a reduction heuristic (see related work above). This can be implemented by the exploration interface (cf.

Sec. 10.3), which prioritizes the outgoing transitions of a state according to the exploration order. Alternative implementations can wrap the original exploration interface (e.g., by an additional JTorX) explorer) to perform these reductions. They are invisible to LazyOTF, so LazyOTF can assume that the specification only has finitely branching states, i.e., that the system specification description incorporates these reductions already. Our weight heuristics (cf. Subsec. 12.3.4) do subsume reduction heuristics, e.g., via a limited, ordered enumeration from highest to lowest weights, but for infinite branching, we need implicit reduction (cf. Note 8.44).

12.3.2. Exhaustiveness and Test Objectives

In practice, exhaustiveness is usually not achievable (cf. Subsec. 8.8.1, Subsec. 10.1.2, Subsec. 11.3.3). But exhaustiveness of a test case generation algorithm gen is a useful property nonetheless since it shows that for every fault, gen can eventually generate a TC that finds the fault after recurrent execution, no matter how improbable. Rephrased to an SUT that does not exhibit all these faults, gen will recurrently add new test cases that increase the assurance that the SUT conforms to its specification [Peleska, 2013]. But exhaustiveness does not say anything about how efficiently this is performed. So for efficient bug finding, other aspects like finding meaningful test cases and discharging test objectives are more important, so that giving up exhaustiveness can lead to a worthwhile trade-off, similarly to the incomplete approaches CMC, lossy hashing and SBMC with too small a bound (cf. Chapter 5), and EMC and testing with SBMC (cf. Chapter 7). This subsection covers exhaustiveness nonetheless, so that we know whether gen can still potentially find all faults, and that we can investigating trade-offs. Relating exhaustiveness to test objectives (the main practical application of LazyOTF) shows that test objectives and exhaustiveness are not opposing.

Exhaustiveness of LazyOTF can be guaranteed if the heuristics meet certain criteria. Since many heuristics are possible, and their usefulness depends on the situation, we do not want to determine one single heuristic. Instead we constrain the heuristics as little as possible to maintain exhaustiveness, resulting in several provisos, which become more and more detailed and tailored towards our specific approach with test objectives (cf. Fig. 12.4 on page 322). Our implementation (cf. Chapter 13) is even more flexible, so several instantiations do not meet all provisos. Therefore, we can experiment and tune test selection heuristics to effectively handle typical situations and practically relevant TOs, possibly as trade-off for hard and practically less relevant goals, such as exhaustiveness or the guarantee to discharge even pathological TOs, i.e., very complex and unlikely TOs.

On the most abstract level, exhaustiveness of LazyOTF can be guaranteed by reducing it to exhaustiveness of OTF or by covering $\mathcal{S}$ sufficiently (cf. Subsec. 8.8.4), as Lemmas 12.3 and 12.4 show. These conditions are formally defined in Def. 12.2 as provisos $\mathrm{P}_{\rightarrow OTF}$ and $\mathrm{P}_{coverage}$ (cf. Fig. 12.4 on page 322).

Definition 12.2. The **OTF limit proviso** ($\mathbf{P}_{\rightarrow OTF}$) demands that after a finite number of test execution steps without fail, LazyOTF behaves identical to OTF: $\forall \mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau)\ \exists k \in \mathbb{N}:$ fail occurs within k test execution steps or f.a. $t > k$: after t execution steps, LazyOTF gives the same results as OTF, i.e., f.a. $\ddot{s} \in S_{det}$: test

execution from $\ddot{s}$ yields the same full path sequence starting from $\ddot{s}$ for LazyOTF as for OTF.

The **coverage proviso** ($\mathbf{P}_{coverage}$) demands that test execution eventually yields fail or $\text{gen}(\mathcal{S}, \mathbb{S}_{sim\,\mathcal{S}})$ covers

- all states of $faultable(\mathcal{S}_{det})$ for SUT with fairness_{test};
- all $faultable(Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}))$ for SUT with fairness_{spec} or fairness_{model}.

Lemma 12.3. *Let LazyOTF use heuristics that meet $P_{\to OTF}$.*
Then LazyOTF is exhaustive.

Proof. If fail occurs during test execution, the TS is exhaustive for the given specification $\mathcal{S}$ and SUT $\mathbb{S}$. Otherwise, $\text{P}_{\to OTF}$ and the restart property guarantees that there exists a $k \in \mathbb{N}$ such that LazyOTF behaves identical to OTF after k steps and restarts in *init after* τ. Since OTF performs purely random with no dependency on previous steps, OTF without the first k steps generates an exhaustive TS for $\mathcal{S}$ and $\mathbb{S}$. Therefore LazyOTF does as well (and even more so with the first k steps included). □

Lemma 12.4. *Let LazyOTF fulfill $P_{coverage}$.*
Then LazyOTF is exhaustive.

Proof. If fail occurs during test execution, the TS is exhaustive for the given specification $\mathcal{S}$ and SUT $\mathbb{S}$. Otherwise, since $\text{P}_{coverage}$ is fulfilled,

- Lemma 8.61 shows that $\text{gen}(\mathcal{S}, \mathbb{S}_{sim\,\mathcal{S}})$ is exhaustive for SUT with fairness_{test};
- Lemma 8.63 shows that $\text{gen}(\mathcal{S}, \mathbb{S}_{sim\,\mathcal{S}})$ is exhaustive for SUT with fairness_{spec};
- Lemma 8.59 shows that $\text{gen}(\mathcal{S}, \mathbb{S}_{sim\,\mathcal{S}})$ is exhaustive for SUT with fairness_{model}.

Hence Lemma 8.69 shows that LazyOTF with fulfilled $\text{P}_{coverage}$ is exhaustive for $\mathcal{S}$ and $\mathbb{S}$. □

There are many ways to achieve $\text{P}_{\to OTF}$: The simplest is by **user interaction**, i.e., the user can switch dynamically between LazyOTF and OTF at any time. Alternatively, we can mitigate guidance, as given in Def. 12.5.

Definition 12.5. Mitigation of LazyOTF's guidance replaces some decisions made by LazyOTF's guidance with random (or nondeterministic) choices. The frequency of those replacements increases with the number of test execution steps without fail .

Probabilistic mitigation replaces LazyOTF's guidance by OTF's random choices, but only with probability p, which increases over time.

Example. By counting the number of execution steps t made so far, and picking a threshold m at which LazyOTF should be completely mitigated, the next transition can be chosen with OTF's random choice with probability $p = min(1, t/m)$ (i.e., LazyOTF's guidance is used with probability $max(0, 1 - t/m)$).

Notes. LazyOTF need not deactivate its phase heuristics for mitigation: the interleaving between execution and traversal sub-phases is transparent to the TCs because for mitigated LazyOTF as well as for OTF, i_δ is chosen nondeterministically in each $\ddot{s}$.

If MBT_{exec} terminates upon fail , one fail is sufficient that LazyOTF need not behave identical to OTF; otherwise, recurrent fail is necessary.

Depending on the testing hypothesis, Def. 12.2 might contain probability: If some behavior in the SUT occurs recurrently after at most $n \in \mathbb{N}$ tries, no probability is required. But if the behavior occurs with probability p for each try, Def. 12.2 must be reformulated such that the probability of $\mathrm{P}_{\rightarrow OTF}$ converges to 1 (e.g., via probabilistic mitigation).

Since the main application of LazyOTF is with test objectives, implementations of $\mathrm{P}_{\rightarrow OTF}$ that make use of test objectives are suited best. A simple approach is to mitigate LazyOTF's guidance by counting only the number of execution steps t since no objective has been discharged, instead of counting all execution steps. A more powerful approach integrates $\mathrm{P}_{\rightarrow OTF}$ with a **guarantee of discharging all test objectives** $\ddot{o}$ if test execution does not fail. Again many implementations are possible, so we give two provisos (cf. Fig. 12.4 on page 322) $\mathrm{P}_{vanishing}$ and $\mathrm{P}_{discharge}$ in Def. 12.6, such that $(\mathrm{P}_{vanishing} \wedge \mathrm{P}_{discharge}) \Rightarrow \mathrm{P}_{\rightarrow OTF}$, as Lemma 12.7 shows. In practice, $\mathrm{P}_{discharge}$ is more relevant than exhaustiveness. We will investigate $\mathrm{P}_{discharge}$ in the next subsection.

Definition 12.6. The **vanishing proviso** ($\mathbf{P}_{vanishing}$) demands: if all test objectives have been discharged, LazyOTF behaves like OTF (i.e., LazyOTF yields the same full path seq as OTF).

The **discharge proviso** ($\mathbf{P}_{discharge}$) demands: given the set of currently active test objectives $\ddot{o}$ and the current superstate $\ddot{s}$, after a finite number of test execution steps from $\ddot{s}$, a test objective $o \in \ddot{o}$ is discharged if no fail occurs beforehand. In short: LazyOTF is **discharging**.

Lemma 12.7. *If LazyOTF meets $P_{vanishing}$ and $P_{discharge}$, then LazyOTF also meets $P_{\rightarrow OTF}$.*

Proof. Let $\ddot{o}$ be the set of active test objectives.

$\mathrm{P}_{discharge}$ guarantees that some $o \in \ddot{o}$ is discharged in some superstate $\ddot{s}$ of $\mathcal{S}$ after a finite number of test execution steps from $init_{\mathcal{S}}$ *after* τ without fail. We can apply $\mathrm{P}_{discharge}$ inductively on $\ddot{o} \setminus \{o\}$ and $\ddot{s}$. Since $\ddot{o}$ is finite, there is a $k \in \mathbb{N}$ such that after k test execution steps, all test objectives in $\ddot{o}$ are discharged or fail occurs. If no fail occurs, $\mathrm{P}_{vanishing}$ implies $\mathrm{P}_{\rightarrow OTF}$. □

$\mathrm{P}_{\rightarrow OTF}$ is stricter than $\mathrm{P}_{coverage}$ since all solutions of $\mathrm{P}_{\rightarrow OTF}$ also guarantee $\mathrm{P}_{coverage}$ because the coverage criteria for the respective fairness are equivalent to exhaustiveness (cf. Lemmas 8.59 and 8.61 and 8.63) if fail does not occur beforehand. As for $\mathrm{P}_{\rightarrow OTF}$, we also investigate an approach that integrates $\mathrm{P}_{coverage}$ with the **guarantee of discharging all test objectives**. Again many implementations are possible, so we define a further proviso (cf. Fig. 12.4 on page 322) $\mathrm{P}_{coverViaTOs}$ in Def. 12.8 to help achieve $\mathrm{P}_{coverage}$via test objectives. Lemma 12.9 shows that $(\mathrm{P}_{discharge} \wedge \mathrm{P}_{coverViaTOs}) \Rightarrow \mathrm{P}_{coverage}$.

Definition 12.8. The **cover via TOs proviso** ($\mathbf{P}_{coverViaTOs}$) demands that $\ddot{o}$ implements the coverage required by $\mathrm{P}_{coverage}$.

Lemma 12.9. *If LazyOTF meets $P_{discharge}$ and $P_{coverViaTOs}$, then LazyOTF also meets $P_{coverage}$.*

Proof. $\mathrm{P}_{discharge}$ guarantees that eventually all $o \in \ddot{o}$ are discharged if no fail occurs beforehand. $\mathrm{P}_{coverViaTOs}$ guarantees that then all elements required by $\mathrm{P}_{coverage}$ have been covered. □

Coverage via TOs focuses on practical coverage; hence it is difficult to implement $\mathrm{P}_{coverage}$ via $\mathrm{P}_{coverViaTOs}$: Weak fairness and unsuitable TOs can cause too complex situations. To avoid TOs that are unnecessarily complex for the given fairness constraint, we can demand proviso $\mathrm{P}_{exh \Rightarrow disch}$ (cf. Def. 12.10).

Definition 12.10. Let a fairness constraint be given.

Then the **exh⇒disch proviso** ($\mathbf{P}_{exh \Rightarrow disch}$) demands that exhaustiveness implies $\mathrm{P}_{discharge}$: f.a. $\ddot{\mathbb{T}} \subseteq \mathrm{genTC}$: ($\ddot{\mathbb{T}}$ is exhaustive $\Rightarrow$ $\forall o \in \ddot{o}$: $\ddot{\mathbb{T}}$ eventually discharges o if executed recurrently).

For instance, a TO o that is only discharged by one specific path is too complex if we have $\mathrm{fairness}_{test}$ and the states of the path can be reached via multiple traces: all following behavior is independent of the trace, so demanding one specific path is too strict. If one specific path is of importance, $\mathcal{S}$ must be extended to incorporate the differentiating behaviors. So with $\mathrm{P}_{exh \Rightarrow disch}$, full state coverage can be reduced to $faultable(S_{det})$ for $\mathrm{fairness}_{test}$ when dealing with $\mathrm{P}_{discharge}$; likewise for all other fairnesses, full trace coverage can be reduced to $faultable(Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}))$ when dealing with $\mathrm{P}_{discharge}$.

Even with $\mathrm{P}_{exh \Rightarrow disch}$, it depends on the testing hypothesis and specification whether $\mathrm{P}_{coverViaTOs}$ can be implemented in practice:

For $\mathrm{fairness}_{test}$ and finite $\mathcal{S}$, $faultable(S_{det})$ is also finite, so it can be covered by creating one TO $o \in \ddot{o}$ for each element. For infinite $\mathcal{S}$, we would have to allow an infinite set $\ddot{o}$ and describe it declaratively.

Likewise, for $\mathrm{fairness}_{spec}$ and $\mathrm{fairness}_{model}$, all of $faultable(Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}))$ can be covered with finitely many test execution steps only if $\mathcal{S}$ is finite and for all $\ddot{s}, \ddot{s}' \in S_{det}$: $\ddot{s} \xrightarrow{l} \ddot{s}'$ only for finitely many $l \in L$, and the only cycles are $\delta_{\circlearrowleft}$ (cf. Lemma 8.74). In this case, we can again create one TO for each element to be covered. For all other cases, $\mathrm{P}_{coverage}$ again requires the coverage over infinitely many elements.

But $\mathrm{P}_{coverViaTOs}$ helps in practice to achieve $\mathrm{P}_{coverage}$if finitely many elements have to be covered, otherwise a coverage level as high as practically possible. Furthermore, Subsec. 12.3.5 will introduce an alternative based on weights without TOs to achieve $\mathrm{P}_{coverage}$,

Note 12.11. Exhaustiveness and the corresponding coverage are inherently difficult to achieve because we allow uncontrollable nondeterminism, arbitrarily pathological faults, and very general SUTs with infinite state spaces, $\mathrm{fairness}_{model}$ and $\mathrm{underspec}_{U}$ (see also Note 12.18).

The remainder of this section will introduce weight heuristics and compositionality of TOs, which add power and flexibility. For instance, we can compose efficient TOs to cover the most relevant aspects with more elaborate TOs based on weights to also detect the more pathological faults eventually.

12.3.3. Discharging Test Objectives

Besides aiding exhaustiveness via $P_{\rightarrow OTF}$ or $P_{coverage}$, $P_{discharge}$ also aids more practical *ö*, e.g., when guidance focuses on a few elements for a feature of interest. Having only a few specific TOs, they can be discharged efficiently. $P_{discharge}$ gives the guarantee that the TOs are eventually discharged, but does not consider efficiency, which will be considered for our weight heuristics. Discharging a small set *ö* to check some requirements or features of interest is suitable and helpful for the software development process [Lackner and Schlingloff, 2012], especially agile approaches that focus on few features each sprint (cf. Sec. 14.2).

Def. 11.3 and Def. 11.5 imply that every TO can be discharged somehow. This chapter shows that $P_{discharge}$ can become difficult because of uncontrollable nondeterminism and because we allow very general cases with infinite state spaces, fairness_{model} and underspec_{U}. But several solutions are possible, and their usefulness depends on the situation. Thus we do not want to determine one single heuristic, similar to the previous subsection. Instead, $P_{discharge}$ is further split down, into the three provisos (cf. Fig. 12.4 on page 322) P_{phase}, P_{goal} and P_{fair}, defined in Def. 12.12. Lemma 12.16 shows that $(P_{phase} \wedge P_{goal} \wedge P_{fair}) \Rightarrow P_{discharge}$

Definition 12.12. The **phase proviso** ($\mathbf{P}_{phase}$) guarantees that the current LazyOTF search for a TO terminates after a finite number of traversal sub-phases; for this, P_{phase} demands that there exist $k \in \mathbb{N}$ and a termination function ***phaseVariant*** such that:

- $phaseVariant : \mathbb{N} \rightarrow [0, \ldots, k]$
- the domain of *phaseVariant* corresponds to the number of traversal sub-phases LazyOTF executed so far;
- while LazyOTF is still active, *phaseVariant* must **recurrently increase** until a test execution fail occurs or the current sub-graph contains a TO, i.e., $\forall q \in \mathbb{N} \exists p > q$: in phase p, LazyOTF is no longer active, or a test execution fail occurs, or its sub-graph contains a TO, or $phaseVariant(p) \gneq phaseVariant(q)$.

The **goal proviso** ($\mathbf{P}_{goal}$) demands that if LazyOTF is active and the sub-graph of the current traversal sub-phase contains a TO, then so will the selected TC $\mathbb{T}_{p_{curr}}$.

The **fairness proviso** ($\mathbf{P}_{fair}$) demands that the TC sequence is constructed in the following way: if the TC sequence recurrently contains a desired state (to a TO or to a higher value for *phaseVariant*), one will eventually also be reached during test execution if no fail occurs (recurrently) during test execution.

Example 12.13. Exemplary *phaseVariant* map the phase p to

- the number of discharged TOs up to phase p;
- the number of covered elements up to phase p, like states, transitions, paths, cycles;
- the distance to *ö*.

Notes 12.14. Because of uncontrollable nondeterminism, the SUT might choose a nondeterministic resolution during the execution of a TC that might not lead towards a TO, which is usually reflected by $phaseVariant(\cdot)$ not being strictly monotonically increasing. But P_{fair} guarantees that eventually test execution does move towards a TO. If we restricted the domain of $phaseVariant(\cdot)$ to those phases that do move towards a TO during test execution, $phaseVariant(\cdot)$ is strictly monotonically increasing and

$k - phaseVariant(\cdot)$ is a classical termination function, i.e., a strictly monotonically decreasing functions with the codomain $\mathbb{N}_{\geq 0}$. But using *phaseVariant* directly is more natural in our case.

P_{goal} is greedy: if a TO exists in the current sub-graph, the selected TC must contain a TO, even if another TC might lead quicker to more TOs in future phases. We favor the greedy approach since our heuristics and uncontrollable nondeterministic cause high uncertainty whether a TO is really reached during test execution, so many early and short tries are usually better than taking higher risks (cf. Sec. 12.2).

Lemma 12.15. *If LazyOTF meets P_{phase}, P_{fair} and P_{goal} and there are still active TOs, then after finitely many phases, LazyOTF constructs a TC that contains a TO or fail occurs during test execution.*

Proof. If LazyOTF's guidance heuristic eventually turns inactive, controllable nondeterministic choices are made randomly, so together with the restart property, eventually a test case sequence is constructed that contains a TO or fail occurs during test execution.

If LazyOTF's guidance heuristic stays active, then P_{phase} and P_{fair} guarantee that eventually fail occurs during test execution or a sub-graph of a traversal sub-phase p contains a TO. Otherwise, there would be a sequence $(p_i)_{i \in \mathbb{N}}$ with strictly monotonically increasing phase numbers p_i, such that $phaseVariant(p_i)$ is strict monotonically increasing, contradicting the bounded codomain $[0, \ldots, k]$. P_{goal} guarantees that some TO is in the TC generated in the traversal sub-phase p. □

Lemma 12.16. *If LazyOTF meets P_{phase}, P_{goal} and P_{fair}, then LazyOTF also meets $P_{discharge}$.*

Proof. Lemma 12.15 has shown that as long as there are still active TOs, the TC sequence recurrently contains TOs, or test execution recurrently has fail s. Therefore, P_{fair} guarantees that some TO will eventually be reached during test execution if no fail occurs (recurrently) beforehand. □

There are several ways to implement $\mathbf{P}_{phase}$ (cf. Example 12.13):

- using the number of discharged TOs as *phaseVariant* and mitigation of LazyOTF's guidance (or user interaction) can guarantees exhaustiveness and also $\mathrm{P}_{discharge}$ if $\mathrm{P}_{exh \Rightarrow disch}$ is met, but this is as undirected as OTF in the worst case;
- if we have a coverage criterion with a finite number of coverage tasks (e.g., for states, transitions, paths, cycles), we can use the coverage level as *phaseVariant*, which corresponds to implementing $\mathrm{P}_{coverViaTOs}$. For an infinite number of coverage tasks, we can use weights without TOs, as described in Subsec. 12.3.5;
- the most directed guidance is using a distance function as *phaseVariant*, which is covered in Subsec. 12.3.7.

There are many ways to implement $\mathbf{P}_{goal}$: The simplest is the most greedy (cf. Note 12.14): pick a TC $\mathbb{T}$ that contains a TO the moment one is found during the traversal. Since $\mathbb{T}$ might not be a potentially most meaningful TC with a TO of the current sub-graph (e.g., $\mathbb{T}$ might contain a TO only very deeply after many uncontrollable nondeterministic choice points), more elaborate implementations complete the traversal through the sub-graph to search for a potentially most meaningful TC with a TO. Subsec. 12.3.6 will model meaningfulness using weight heuristics and investigate P_{goal} for them.

Note 12.17. For real-time computing, the most greedy approach to satisfy the provisos can be investigated as future work.

To guarantee $\mathbf{P}_{fair}$ for fairness_{test} and fairness_{spec} without underspec_U, recurrent re-executions are sufficient. For fairness_{model} or allowed underspec_U, full $Straces_{\mathcal{S}_{\tau * \delta}}(init_{\mathcal{S}})$ needs to be covered so that test execution can reach a desired state even if it is pathologically hard to reach. This will be shown together with other lemmas about meeting provisos in Subsec. 12.3.6.

Note 12.18. These strong requirements, especially for P_{fair}, are not a deficit of our heuristics: $\text{P}_{discharge}$ is inherently difficult since

- we allow a flexible way of defining TOs;
- P_{fair} is at least as difficult as the reachability problem on $\mathcal{S}_{det}$, and for weak fairness or underspec_U as hard as exhaustiveness to cover all pathological cases (see also Note 12.11, [Anand et al., 2013]);
- it is not clear how often a TC must be executed to show a desired and possible behavior.

Without P_{fair}, the TC seq may contain infinitely many TOs (e.g., one specific TO recurrently via the same path from *init*), but still never discharge any TO (as the proof of Lemma 12.31 will show).

12.3.4. Weight Heuristics

The previous subsection has motivated using weights to guarantee P_{goal}, P_{phase} and P_{fair}, which will be investigated in the remainder of this section.

LazyOTF implements metric-driven guidance with **heuristics using weights** (though alternatives are possible, e.g., via cycling heuristics, cf. Subsec. 12.3.1 and Subsec.12.3.9, or via SMT constraints, cf. Subsec. 3.3.3 and Sec. 13.4). But using weights is flexible and expressive, as the following subsections will show. For weight heuristics, each test case has an associated weight, where higher values indicate potentially more meaningful test case execution, enabling test case prioritization for test selection. Test cases with associated weights are defined in Def. 12.19.

Definition 12.19. Let $\mathbb{W} = (S, \rightarrow, L_\delta, w)$ with $(S, \rightarrow, L_\delta) \in \mathcal{TTS}(L_I, L_U, \delta)$ and w a function $S \rightarrow \mathbb{Z}$. Then:

- w is called a **weight function** on S;
- $\mathbb{W}$ is a **weighted test case** (**WTC**) of the test case $(S \rightarrow, L_\delta)$;
- $\boldsymbol{\mathcal{WTTS}(L_I, L_U, \delta)}$ denotes the set of all weighted test cases of $\mathcal{TTS}(L_I, L_U, \delta)$;
- $\boldsymbol{w_{\mathcal{WTTS}}}$ ($\boldsymbol{w}$ for short): $\mathcal{WTTS}(L_I, L_U, \delta) \rightarrow \mathbb{Z}, \mathbb{W} \mapsto w(init_{\mathbb{W}})$ is called the **weight function on** $\mathcal{WTTS}$, $w_{\mathcal{WTTS}}(\mathbb{W})$ the **weight of** $\mathbb{W}$;
- $\boldsymbol{\mathcal{F}_w} : \mathcal{WTTS}(L_I, L_U, \delta) \rightarrow \mathcal{TTS}(L_I, L_U, \delta), (S, \rightarrow, L_\delta, w) \mapsto (S, \rightarrow, L_\delta)$ is called **forgetful transformation for weights**;
- $\ddot{\mathbb{W}} \subseteq \mathcal{WTTS}(L_I, L_U, \delta)$ with injective $\mathcal{F}_w|_{\ddot{\mathbb{W}}}$ is called a **weighted test suite** (**WTS**);

To show how weights are computed for TCs, genWTC from Listing 12.1 modifies genTC from Listing 8.2: The meaningfulness of the path π from $init_{\mathcal{S}}$ *after* τ to the current superstate is estimated using **meaningfulness weight heuristics**, implemented by

the polymorphic function ***path2W*(π)**. Those weights are in turn **aggregated to weights of TCs** using the polymorphic function ***aggWTCs*** (cf. Fig. 12.1 and Example 12.35 on page 309). Otherwise, genWTC is equal to genTC: It decides termination nondeterministically and does not use weights to resolve other nondeterminism since weight heuristics are only sensible when TCs cannot become arbitrarily large. As for genTC, worst case complexities are not sensible. The main resource consumption for each genWTC call are for assembleWTC and $\ddot{s}$ $after_{\mathcal{S}_{\tau^*\delta}}$ l, similar to genTC, and additionally for $path2W(\pi)$, usually $O(1)$ space and $O(t_{curr} \cdot |\ddot{o}_{init}|)$ time. So for genWTC that returns $\mathbb{W} = (S, T, L_\delta, w) \in \mathcal{TTS}(L_I, L_U, \delta)$, the time complexity is in $O(|S| \cdot (|S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}} + t_{curr} \cdot |\ddot{o}_{init}|))$, the space complexity is the same as for genTC, i.e., in $O(|S| \cdot |S_{\to^*}| + |T| + branch_{\mathcal{S}_{det}})$.

genWTS from Listing 12.2 will add both deterministic termination and weight heuristics.

```
proc WTTS(L_I, L_U, δ) genWTC(LTS(L_I, L_U, τ) S, paths(S_det) π)                1
  2^S s̈ := dest(π);                                                             2
  TreeState s := new TreeState representing s̈;                                  3
  𝔹 terminate := nondet({false, true});                                         4
  L_I ∪̇ {δ} i_δ := nondet(in_{S_τ*}(s̈) ∪̇ {δ});                                 5
  L_U ∪̇ {i_δ} → WTTS(L_I, L_U, δ) l2WTC:=                                       6
                    l ↦ genWTC(S, π · (s̈ --l--> s̈ after_{S_τ*δ} l));
                                                                                7
  if (s̈ == ∅)                                                                   8
  then return new TreeState fail ↦ path2W(π);                                   9
  fi;                                                                           10
  if (terminate)                                                                11
  then return new TreeState pass ↦ path2W(π);                                   12
  fi;                                                                           13
  return assembleWTC(s, path2W(π), l2WTC);                                      14
end;                                                                            15
```

Listing 12.1: Typed nondeterministic genWTC($\mathcal{S}$, π)

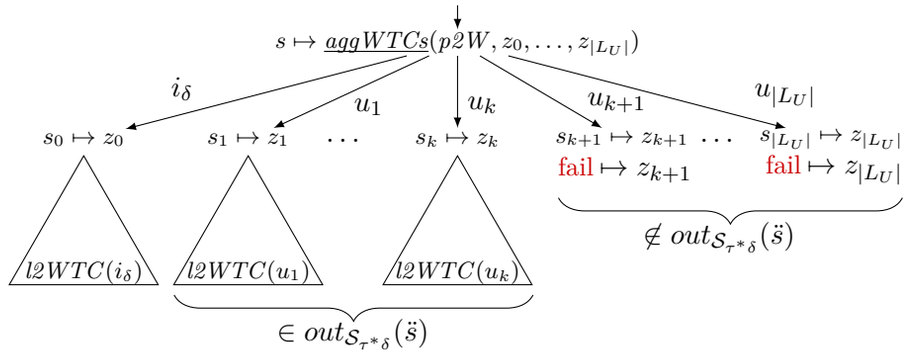


Figure 12.1.: assembleWTC(s, $p2W$, $l2WTC$) (with L_U finite)

Notes 12.20. Since $\pi \in paths(\mathcal{S}_{det})$, π starts from $init_{\mathcal{S}_{det}}$, e.g., is the path $\pi^{full}_{r_{curr}}$. If more information is required, larger parts of dynamicInfo can be passed, e.g., the full path sequence for the full history of previous execution sub-phases. Using the concatenated full path π^{full}, the full path sequence can be encoded in $paths(\mathcal{S}_{det})$ (cf. Def. 11.9). This improves readability of Listing 12.1, but also enables prioritization of faults by weights: the domain of $\underline{path2W}$ is extended accordingly, such that it can weight maximal paths of TCs with verdict fail, e.g., prioritize faults according to their severity and relevance. If the set $paths(\mathcal{S}_{det})$ should not be extended, genWTC can detect that $l \notin out_{\mathcal{S}_{\tau^*\delta}}(dest(\pi))$ and choose fail $\mapsto 0$ as value of *l2WTC* in l.6 without a further recursive call of genWTC.

Since genWTC traverses down sub-graphs in a DFS and $\underline{path2W}((\pi_i^{full})_i)$ can investigate $\pi_{p_{curr}}$, many superstates in the sub-graph can be considered multiple times for weighing. On the two extremes,

- $\underline{path2W}((\pi_i^{full})_i)$ could consider the full path sequence but be called only in the leafs of $\mathbb{T}_{p_{curr}}$. The advantage is that considering the full path sequence enables weights that reflect meaningfulness not just for reachability properties, and can incorporate the full history;
- $\underline{path2W}((\pi_i^{full})_i)$ could consider only $dest(\pi_{p_{curr}})$ but be called for every node of $\mathbb{T}_{p_{curr}}$. The advantage is that the DFS traversal can be used to investigate the superstates efficiently, and each superstate is considered only once, even if many extensions of the current path occur in the DFS.

To allow all advantages, our approach makes no restrictions and hence allows all combinations.

Corollary 12.21. *genWTC*$(\cdot, init\ after\ \tau)$ *is sound and exhaustive.*

Proof. genWTC only adds weights, but otherwise performs identical to genTC, so $\forall \mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau) : \mathcal{F}_w(\text{genWTC}(\mathcal{S}, init_{\mathcal{S}}\ after\ \tau)) = \text{genTC}(\mathcal{S}, init_{\mathcal{S}}\ after\ \tau)$. Thus Theorem 8.48 shows that genWTC$(\cdot, init\ after\ \tau)$ is sound and exhaustive. □

genWTS$(\mathcal{S}, \pi, b)$ from Listing 12.2 is a bounded deterministic version of genWTC, like genTS is of genTC: It generates a WTS containing all WTCs from genWTC$(\mathcal{S}, \pi)$ with paths pruned at depth b and maximal weight. This test selection based on weight heuristics becomes sensible in genWTS since termination is no longer decided nondeterministically, but by the bound b. For more flexible termination, genWTS additionally introduces a polymorphic method *isInducing*(π): For equal behavior to genTS, its implementation simply returns **false**. But it can also implement LazyOTF's phase heuristics, as described in Subsec. 12.2. genWTS computes weights by modifying genTS similarly to genWTC's modification of genTC. The weights are now additionally used for guidance by picking exactly a heaviest WTCs (cf. l.15–l.18), i.e., a potentially most meaningful: since all included i_δ lead to WTCs with the same weight w_{max}, $w(s) = \underline{aggWTCs}(p2W, z_0, \ldots, z_{|L_U|})$ (cf. Fig. 12.1) is also always the same. For the same reason, the i_δ further down in the chosen WTCs (cf. l.14) may vary.

The worst case complexities are similar to genTC and also extremely rough: As for genWTC, we additionally require time to compute $\underline{path2W}(\pi)$ in each node, which is usually in $O(t_{curr} \cdot |\ddot{o}_{init}|)$, but often much less (cf. Note 12.20). Furthermore, we require time to compute *isInducing*(π) in each node, which is usually in $O(|S_{\to^*}| \cdot |\ddot{o}_{init}|)$. The complexity in Landau notation for the aggregation per node does not change. Hence the

overall **worst case time complexity of genWTS**$(\mathcal{S}, init_{\mathcal{S}}\ after\ \tau, b)$ is in $O(branch^{b}_{\mathcal{S}_{det}} \cdot (|S_{\to^*}| \cdot branch_{\mathcal{S}_{\to^*}} + (t_{curr} + |S_{\to^*}|) \cdot |\ddot{o}_{init}|))$, The space for computing $\underline{path2W}(\pi)$ and $isInducing(\pi)$ is negligible. In the worst case, all WTCs have the same weight. Thus the overall **worst case space complexity of genWTS**$(\mathcal{S}, init_{\mathcal{S}}\ after\ \tau, b)$ is the same as for genTS, i.e., in $O(branch^{b}_{\mathcal{S}_{det}} \cdot (|S_{\to^*}| + |L|))$.

```
proc 2^{WTTS(L_I,L_U,δ)} genWTS( LTS(L_I, L_U, τ) S, paths(S_det) π, ℕ_{≥0} b )                1
  2^S s̈ := dest(π);                                                                          2
  TreeState s:= new TreeState representing s̈;                                                3
  2^{WTTS(L_I,L_U,δ)} result:=∅;                                                             4
  int w_max := 0;                                                                            5
                                                                                             6
  if (s̈ == ∅)                                                                               7
  then return {new TreeState fail ↦ path2W(π)};                                              8
  fi;                                                                                        9
  if (b == 0 or isInducing(π))                                                               10
  then return {new TreeState pass ↦ path2W(π)};                                              11
  fi;                                                                                        12
  for each L_I ∪̇ {δ} i_δ ∈ in_{S_{τ*}}(s̈) ∪̇ {δ} do                                          13
    for each L_U ∪̇ {i_δ} → WTTS(L_I, L_U, δ) l2WTC with                                       14
        ∀l ∈ L_U ∪̇ {i_δ} : l2WTC(l) ∈ genWTS(S, π · (s̈ --l--> s̈ after_{S_{τ*δ}} l), b − 1) do
      WTTS(L_I, L_U, δ) 𝕎 := assembleWTC(s, path2W(π), l2WTC);                               15
      if (w(𝕎) < w_max) then break; fi;                                                      16
      if (w(𝕎) > w_max) then w_max := w(𝕎); result:=∅; fi;                                   17
      result.add(𝕎);                                                                         18
    od;                                                                                      19
  od;                                                                                        20
  return result;                                                                             21
end;                                                                                         22
```

Listing 12.2: Typed deterministic genWTS$(\mathcal{S}, \pi, b)$

Note. For $b \in \mathbb{N}$, a WTC $\mathbb{W} \in \text{genWTS}(\cdot, init\ after\ \tau, b+1)$ may not extend any WTC in $\mathbb{W} \in \text{genWTS}(\cdot, init\ after\ \tau, b)$. This is a consequence of the fact that there are most meaningful TCs of size b_1 that are no extensions of most meaningful TCs of size $b_2 < b_1$ (cf. Sec. 12.1). Although this contrasts with Lemma 8.54, exhaustiveness of genWTS$(\cdot, init\ after\ \tau, b+1)$ can still be guaranteed with the help of the provisos from Subsec. 12.3.2, as Theorem 12.22 shows.

Theorem 12.22. *genWTS*$(\cdot, init\ after\ \tau, \mathbb{N})$ *is sound; if proviso* $P_{\to OTF}$ *or* $P_{coverage}$ *is met, genWTS*$(\cdot, init\ after\ \tau, \mathbb{N})$ *is also exhaustive.*

Proof. Soundness follows from Cor. 12.21, exhaustiveness from Lem. 12.3 and 12.4. □

Instead of enumerating all heaviest WTCs, a single heaviest one can be chosen non-deterministically (or randomly): The only modifications for this are l.14 to $L_U \dot\cup \{i_\delta\} \to \mathcal{WTTS}(L_I, L_U, \delta)\ l2WTC := l \mapsto \text{genWTS}(\mathcal{S}, \ddot{s}\ after_{\mathcal{S}_{\tau^*\delta}}\ l, b-1)$; and l.18 to result:={nondet(result$\dot\cup${𝕎})};. For this, the overall **worst case space complexity of genWTS**$(\mathcal{S}, init_{\mathcal{S}}$

after τ, b) reduces to $O((branchout_{\mathcal{S}_{det}} + 1)^b \cdot (|S_{\rightarrow^*}| + |L_U|))$. The resulting LazyOTF with weight heuristics is given in Def. 12.23. Cor. 12.24 lifts Theorem 12.22 to LazyOTF, i.e., shows that LazyOTF retains soundness, and exhaustiveness if $\mathrm{P}_{\rightarrow OTF}$ or $\mathrm{P}_{coverage}$ is met. Furthermore, $\mathrm{P}_{discharge}$ guarantees that LazyOTF discharges all TOs.

Definition 12.23. LazyOTF with weight heuristics performs as described in Chapter 11, but with genWTS in each traversal sub-phase to nondeterministically (or randomly) choose among the heaviest WTCs.

Corollary 12.24. *LazyOTF with weight heuristics is sound.*

If LazyOTF uses weight heuristics that meet $P_{\rightarrow OTF}$ *or* $P_{coverage}$*, then LazyOTF is exhaustive.*

If LazyOTF uses weight heuristics that meet $P_{discharge}$*, then LazyOTF is discharging.*

Note. Having refined LazyOTF to use the test generation algorithm genWTS in each traversal sub-phase, and nondeterministically (or randomly) choose among the heaviest WTCs, we can now refine the general worst case complexities per traversalSubphase that were investigated in Subsec. 11.3.3 on the basis of the bounded variant of the offline MBT algorithm: the worst case time complexity remains identical, but the worst case space complexity of genWTS has the values $|L_U|$, resp. $branchout_{\mathcal{S}_{det}}$, instead of the parameter $branch_{\mathcal{S}_{det}}$ for traversalSubphase, because it is adjusted to storing one heaviest WTC.

Def. 12.25 extends Def. 11.8 by defining (full) weighted test case sequences.

Definition 12.25. Let $\mathcal{S} \in SPEC$. Then we define for a whole run of LazyOTF via weight heuristics:

- the **weighted test case sequence** $(\mathbb{W}_i)_{i \in [1,\ldots,1+p_{curr})}$ (**WTC seq** for short) with $\forall p \in [1, \ldots, 1 + p_{curr}) :$
  $$\mathbb{W}_p \in \begin{cases} \text{genWTS}(\mathcal{S}, init_{\mathcal{S}} \text{ after } \tau, b_p) & \text{if } p \in (r_i)_i, \\ \text{genWTS}(\mathcal{S}, dest(\pi_{p-1}), b_p) & \text{if } p \notin (r_i)_i; \end{cases}$$
- the **full weighted test case sequence** $(\mathbb{W}_i^{full})_{i \in [1,\ldots,1+r_{curr})}$ (**full WTC seq** for short) with $\forall i \in [1, \ldots, r_{curr}) :$
  $\mathbb{W}_i^{full} := \prod_{p \in [r_i,\ldots,r_{i+1})} \mathbb{W}_p$, and $\mathbb{W}_{r_{curr}}^{full} := \prod_{p \in [r_{r_{curr}},\ldots,p_{curr}]} \mathbb{W}_p$ if $p_{curr} < \omega$;
- the TC seq is now $(\mathbb{T}_i)_{i \in [1,\ldots,1+p_{curr})} = (\mathcal{F}_w(\mathbb{W}_i))_{i \in [1,\ldots,1+p_{curr})}$;
- the full TC seq is now $(\mathbb{T}_i^{full})_{i \in [1,\ldots,1+r_{curr})} = (\mathcal{F}_w(\mathbb{W}_i^{full}))_{i \in [1,\ldots,1+r_{curr})}$.

Weights are very flexible since the codomain of *path2W*(π^{full}) is $\mathbb{Z}$: they can quantify meaningfulness of π^{full} and allow various computations, e.g.:

- they can be aggregated by *aggWTCs* (as in Listing 12.1) to derive WTCs, i.e., weights of TCs that describe their potential meaningfulness. This will be investigated in Subsec. 12.3.6;
- they can be aggregated over multiple *path2W* are used, e.g., from different TOs. This will be investigated in Subsec. 12.3.7;
- they can be manipulated to factor in various concerns, e.g., $\mathrm{P}_{\rightarrow OTF}$, $\mathrm{P}_{coverage}$, or further heuristics like cycling. This will be investigated in the next subsection.

12.3.5. Exhaustiveness and Coverage via Weight Heuristics

Having introduced weight heuristics in the previous subsection, we can now investigate our provisos for them. Since weights can be processed in many ways, they can be used to implement various heuristics and construct new ones, as this subsection will exhibit. Since our focus is on discharging TOs, we will thoroughly cover $\mathrm{P}_{discharge}$ and weight heuristics with TOs in the next subsection. Here, we investigate the use of our flexible weight heuristics for other provisos (cf. Fig. 12.4 on page 322): $\mathrm{P}_{\to OTF}$, using mitigation via weights, and $\mathrm{P}_{coverage}$, using *covFiningPath2W*.

To factor in multiple concerns, e.g., proviso $\mathrm{P}_{\to OTF}$ and further heuristics like coverage or cycling, the weights of *path2W* can easily be adapted by multiplying (with rounding) the weights with a factor in $[0, \ldots, 1]$. This kind of compositionality is implemented by wrapping another *path2W* called ***finingPath2W*** around the original *path2W*. Alternatively, *finingPath2W* can be multiplied with a large constant to result in a stand-alone *path2W*.

Mitigation via weights improves probabilistic mitigation: The number of steps t and threshold m no longer determines the probability $p = min(1, t/m)$ to use random choices. Instead, a *finingPath2W* uses the factor $max(0, 1-t/m)$. Thus the prioritization of particularly meaningful paths prevail longer since their weights are still higher. But after at most m steps, all *path2W* vanish (i.e., result in weigh 0) since the factor of the *finingPath2W* converges to 0. Thus all WTCs weigh 0 (for all sensible *aggWTCs*, cf. Subsec. 12.3.6), so a value $i_\delta \in L_I \dot\cup \{\delta\}$ is chosen nondeterministically (or randomly) (cf. Lemma 12.23), as for OTF.

$\mathrm{P}_{coverage}$ via weights can be implemented directly by $\underline{path2W}((\pi_i^{full})_i)$ returning sufficiently high values the moment $\pi_{r_{curr}}^{full}$ visits a new coverage task, and sufficiently low values otherwise. One simple implementation for such a $\underline{path2W}((\pi_i^{full})_i)$ is to return a high constant the moment $\pi_{r_{curr}}^{full}$ reaches a coverage task, and a low constant otherwise (similarly to nonfancy *path2W*, cf. Example 12.43). This results in purely random choices within sub-graphs that have no coverage task. A more directed guidance via $\mathrm{P}_{coverage}$ employs *finingPath2W* to replace randomness with better distribution with regard to the coverage criterion, resulting in *covFiningPath2W*, see Def. 12.26.

Definition 12.26. Let $\mathcal{S} = (S, T, L_\tau) \in SPEC$ and the finite full path sequence $(\pi_i^{full})_i$ be given, π^{full} its representation as concatenated full path (cf. Def. 11.9) and $(\sigma_i^{full})_i$ its full trace sequence. Then:

The **coverage efficiency** $\boldsymbol{f_{new}}((\pi_i^{full})_i) \in (0, \ldots, 1]$ returns higher weights if new elements are preferably added, i.e., to full path sequences that have more differing elements:

$$f_{new}((\pi_i^{full})_i) := \begin{cases} \frac{\left|\left\{\ddot{s} \in S_{det} \middle| \ddot{s} \in \pi^{full}\right\}\right|}{t_{curr}} & \text{for state coverage} \\ \frac{\left|\left\{\ddot{s} \xrightarrow[d]{l} \ddot{s}' \middle| \ddot{s} \xrightarrow[d]{l} \ddot{s}' \in \pi^{full}\right\}\right|}{t_{curr}} & \text{for transition coverage} \\ \frac{\left|\left\{trace(\pi) | \pi \in (\pi_i^{full})_i\right\}\right|}{r_{curr}} & \text{for trace coverage.} \end{cases}$$

The **coverage distribution** $\boldsymbol{f_{distr}}$ $((\pi_i^{full})_i) \in (0, \ldots, 1]$ returns higher weights if infrequent elements are preferably added, i.e., to full path sequences with better distribution of elements: $f_{distr}((\pi_i^{full})_i) :=$

$$\begin{cases} \dfrac{\min\limits_{\ddot{s}\in\pi^{full}\cap S}(\text{number of occurrences of } \ddot{s} \text{ in } \pi^{full})}{\max\limits_{\ddot{s}\in\pi^{full}\cap S}(\text{number of occurrences of } \ddot{s} \text{ in } \pi^{full})} & \text{for state coverage} \\ \dfrac{\min\limits_{l\in\pi^{full}\cap T}(\text{number of occurrences of } l \text{ in } \pi^{full})}{\max\limits_{l\in\pi^{full}\cap T}(\text{number of occurrences of } l \text{ in } \pi^{full})} & \text{for transition coverage} \\ \dfrac{\min\limits_{\pi\in(\pi_i^{full})_i}(\text{number of occurrences of } trace(\pi) \text{ in } (\sigma_i^{full})_i)}{\max\limits_{\pi\in(\pi_i^{full})_i}(\text{number of occurrences of } trace(\pi) \text{ in } (\sigma_i^{full})_i)} & \text{for trace coverage.} \end{cases}$$

$\underline{\boldsymbol{covFiningPath2W}}((\pi_i^{full})_i) := f_{new}((\pi_i^{full})_i) \cdot f_{distr}((\pi_i^{full})_i)$.

Notes. In general, computing $\underline{covFiningPath2W}$ has high runtime and memory requirements.

The resulting weights might favor certain elements at early phases due to high values from the wrapped $\underline{path2W}$, but if they are bounded, $\underline{covFiningPath2W}$ will eventually even them out, i.e., fine those high values sufficiently to switch to other elements. Thus paths to each coverage task will recurrently be ranked highest and chosen, with the selection converging to equidistribution.

If $\underline{covFiningPath2W}$ converges too quickly towards 0, i.e., all wrapped $\underline{path2W}$ are nullified, $\underline{covFiningPath2W}$ can be normalized, i.e., the factor $\underline{covFiningPath2W}((\pi_i^{full})_i)$ is replaced by $\underline{covFiningPath2W}((\pi_i^{full})_i) / \max\limits_{\pi\in(\pi_i^{full})_i}(\underline{covFiningPath2W}((\pi_i^{full})_i))$. This computation is, however, costly.

$\underline{covFiningPath2W}((\pi_i^{full})_i)$ can similarly be used to implement coverage criteria for nondeterminism on output, e.g., n-choices state coverage, all-choices state coverage, or all-choices transition coverage (cf. Subsec. 12.3.1, [Faragó, 2011]), as defined in Def. 12.27.

Definition 12.27. Let $\mathcal{S} = (S, T, L_\tau) \in SPEC$, $n \in \mathbb{N}_{>0}$, the finite full path sequence $(\pi_i^{full})_i$ be given, and π^{full} its representation as concatenated full path. Then:

For convenience and $n \in \mathbb{N}_{>0}$, we define the following sets of state and transitions in $\mathcal{S}_{det}$:

- $S_{\lambda n} := \{\ddot{s} \in S_{det} \,|\, |out_{\mathcal{S}_{det}}(\ddot{s})| \geq n\}$;
- $S_{\lambda n}(\pi^{full}) := \{\ddot{s} \in S_{\lambda 2} \,|\, |out_{\mathcal{S}_{\pi^{full}}}(\ddot{s})| \geq n\}$;
- $S_{all}(\pi^{full}) := \{\ddot{s} \in S_{\lambda 2} \,|\, out_{\mathcal{S}_{det}}(\ddot{s}) = out_{\mathcal{S}_{\pi^{full}}}(\ddot{s})\}$;
- $T_{all}(\pi^{full}) := \{\ddot{s} \xrightarrow[d]{l} \ddot{s}' \in T \,|\, \ddot{s} \in S_{\lambda 2}(\pi^{full}) \text{ and } \ddot{s} \xrightarrow{l} \ddot{s}' \in \pi^{full}\}$.

The **coverage efficiency for nondeterminism on output** is $\boldsymbol{f_{new}}((\pi_i^{full})_i) :=$

$$\begin{cases} \dfrac{|S_{\lambda n}(\pi^{full})|}{t_{curr}} & \text{for } n\text{-choices state coverage} \\ \dfrac{|S_{all}(\pi^{full})|}{t_{curr}} & \text{for all-choices state coverage} \\ \dfrac{|T_{all}(\pi^{full})|}{t_{curr}} & \text{for all-choices transition coverage.} \end{cases}$$

The **coverage distribution for nondeterminism on output** is $\boldsymbol{f_{distr}}((\pi_i^{full})_i) :=$

$$\begin{cases} \dfrac{\min\limits_{\ddot{s}\in S_{\curlywedge n}(\pi^{full})} (\text{number of occurrences of } \ddot{s} \text{ in } \pi^{full})}{\max\limits_{\ddot{s}\in S_{\curlywedge n}(\pi^{full})} (\text{number of occurrences of } \ddot{s} \text{ in } \pi^{full})} & \text{for } n\text{-choices state coverage} \\ \dfrac{\min\limits_{\ddot{s}\in S_{all}(\pi^{full})} (\text{number of occurrences of } \ddot{s} \text{ in } \pi^{full})}{\max\limits_{\ddot{s}\in S_{all}(\pi^{full})} (\text{number of occurrences of } \ddot{s} \text{ in } \pi^{full})} & \text{for all-choices state coverage} \\ \dfrac{\min\limits_{\ddot{s}\xrightarrow{l}\ddot{s}'\in T_{all}(\pi^{full})} (\text{number of occur. of } \ddot{s}\xrightarrow{l}\ddot{s}' \text{ in } \pi^{full})}{\max\limits_{\ddot{s}\xrightarrow{l}\ddot{s}'\in T_{all}(\pi^{full})} (\text{number of occur. of } \ddot{s}\xrightarrow{l}\ddot{s}' \text{ in } \pi^{full})} & \text{for all-choices transition cov.} \end{cases}$$

With this definition, we have the following coverage levels:

- $\dfrac{\left|\left\{\ddot{s}\in S_{det}\middle|\ddot{s}\in\pi^{full}\right\}\right|}{|S_{det}|}$ for state coverage
- $\dfrac{\left|\left\{\ddot{s}\xrightarrow{l}_{\vec{d}}\ddot{s}'\middle|\ddot{s}\xrightarrow{l}_{\vec{d}'}\ddot{s}'\in\pi^{full}\right\}\right|}{|\overrightarrow{d}|}$ for transition coverage
- $\dfrac{\left|\left\{trace(\pi)|\pi\in(\pi_i^{full})_i\right|\right.}{|Straces_{\mathcal{S}_\tau*\delta}|}$ for trace coverage
- $\dfrac{\left|S_{\curlywedge n}(\pi^{full})\right|}{|S_{\curlywedge n}|}$ for n-choices state coverage
- $\dfrac{\left|S_{all}(\pi^{full})\right|}{|S_{\curlywedge 2}|}$ for all-choices state coverage
- $\dfrac{\left|T_{all}(\pi^{full})\right|}{\left|\left\{\ddot{s}\xrightarrow{l}_{\vec{d}}\ddot{s}'|\ddot{s}\in S_{\curlywedge 2}\right\}\right|}$ for all-choices transition coverage.

Since $S_{all}(\pi^{full}) \subseteq S_{\curlywedge 2}(\pi^{full}) \subseteq S_{\curlywedge 1}(\pi^{full})$, the coverage levels are: all-choices state coverage $\leq$ 2-choices state coverage $\leq$ 1-choice state coverage.

path2W(π^{full}) can best approximate meaningfulness and reduce its complexity and the amount of randomness of guidance by incorporating rules individual to the given specification $\mathcal{S}$. Such rules can be described efficiently with the help of TOs and their composition (cf. Subsec. 12.3.7). Hence we investigate the TO-based provisos (cf. Fig. 12.4 on page 322) for weight heuristics in the next subsection. Since TO-based heuristics are the focus of this thesis, most weight computations in this subsection have not yet been implemented.

12.3.6. Discharging Test Objectives Via Weight Heuristics

Before focusing on $\mathrm{P}_{discharge}$, we consider $\mathrm{P}_{vanishing}$, since their conjunction guarantees $\mathrm{P}_{\rightarrow OTF}$ and hence exhaustiveness. $\mathrm{P}_{vanishing}$ can be met easily for our weight heuristics by guaranteeing that all WTCs have identical weight once all test objectives have been discharged, e.g., weight 0. Then repetitive genWTS that returns one nondeterministic TC behaves identical to genTC (except additionally computing weights). Only returning the weight 0 can for instance be achieved by *aggWTCs* and *aggPath2Ws* not adding constants. As for mitigation via weights, the interleaving between execution and traversal sub-phases is transparent to the TCs.

Subsec. 12.3.3 has investigated P_{phase}, P_{goal} and P_{fair} to guarantee $\mathrm{P}_{discharge}$, introduced several general ways to achieve them, and motivated the use of weights for this.

Lemma 12.29 gives sufficient conditions for $w(\mathbb{W}_{p_{curr}})$ to meet P_{phase}. Lemma 12.31, resp. Lemma 12.33, show how our weight heuristics can generally meet P_{fair}, resp. P_{goal}.

Def. 12.28 is used in Lemma 12.29 and is a strict case for a recurrently increasing *phaseVariant* because it does not allow values to decrease in later phases without discharges in between (cf. beginning of Subsec. 12.3.8) for at least one nondeterministic resolution (of uncontrollable nondeterminism and of nondeterministic restarts, cf. Subsec. 11.3.2). We usually do not demand strictness for monotonically increasing sequences because we do not require it and stuttering is helpful in practice (e.g., due to stuttering variable evaluations).

Definition 12.28. A sequence $(a_i)_i \in \mathbb{Z}^{[1,\dots,1+z)}$ of size $z \in \omega + 1$ is:

- **finitely monotonically increasing** $:\Longleftrightarrow$
  $\forall i \in [1,\dots,1+z) : a_i \geq a_{i-1}$ and only finite **stuttering**, i.e., $\forall i \in [1,\dots,1+z) \exists j > i : a_j \gneq a_i$;
- also **strictly monotonically increasing** $:\Longleftrightarrow$
  $\forall i \in [1,\dots,1+z) : a_i \gneq a_{i-1}$.

Lemma 12.29. *Let $w(\mathbb{W}_{p_{curr}})$ be implemented via genWTS.*

If $w(\mathbb{W}_{p_{curr}})$ has a finite codomain and is finitely monotonically increasing for at least one nondeterministic resolution until fail occurs or the current sub-graph contains a TO, then $w(\mathbb{W}_{p_{curr}})$ is a phaseVariant that meets P_{phase}.

Proof. If LazyOTF is still active, without test execution fail, trying to discharge TO o, then $w(\mathbb{W}_{p_{curr}})$ is a *phaseVariant* that has a finite codomain and is recurrently increasing until its sub-graph contains a TO, since the finitely monotonically increasing $w(\mathbb{W}_{p_{curr}})$ is only finitely stuttering. □

Note 12.30. When using this argument of finitely monotonically increasing weights, we guarantee P_{phase} by choosing a fitness function that has a TO in each maximum and consider the choices of $\mathbb{W}_i$ in LazyOTF's phases as steepest ascent hill climbing [Harman et al., 2009] (that does not stop when moving downwards, modulo stuttering, modulo fail, modulo necessary restarts due to nondeterminism).

Lemma 12.31. *Let $(\mathbb{W}_i)_{i\in[1,\dots,1+p_{curr})}$ be the WTC seq that LazyOTF generates.*

For $fairness_{spec}$ (or $fairness_{test}$), with forbidden $underspec_U$, $(\mathbb{W}_i)_{i\in[1,\dots,1+p_{curr})}$ meets P_{fair} if it recurrently restarts and each $\mathcal{F}_w(\mathbb{W}_i)$ is executed recurrently.

For $fairness_{model}$ or allowed $underspec_U$, $(\mathbb{W}_i)_{i\in[1,\dots,1+p_{curr})}$ meets P_{fair} if it eventually covers all $Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$.

Proof. For $\mathrm{fairness}_{spec}$ (or $\mathrm{fairness}_{test}$), with forbidden $\mathrm{underspec}_U$, Lemma 8.67 shows that recurrent execution of some $\mathbb{W}$ will eventually lead to a desired state (to a higher value for *phaseVariant* or to a TO) in the next phase if no fail occurs (recurrently) during test execution.

For $\mathrm{fairness}_{model}$ or any fairness with allowed $\mathrm{underspec}_U$, this is not sufficient: For $\mathbb{M} \in MOD$, $\mathbb{W}_{p_{curr}}$ might only contain a desired state $\ddot{s}$ reachable via path $path_{\ddot{s}}^{full} \in paths^{fin}(\mathcal{F}_w((\mathbb{W}_i)_i))$ such that: $trace(path_{\ddot{s}}^{full}) \notin trace(\mathcal{F}_{mod}(paths_{max}(\mathcal{F}_w(\mathbb{W}_{r_{curr}}^{full})||\mathbb{M}_{\delta\tau^*})))$, i.e., $trace(path_{\ddot{s}}^{full})$ never occurs in $\mathbb{M}$. In the worst case, all but one $\sigma \in$

$Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$ with $\ddot{s} = init_{\mathcal{S}_{\tau^*\delta}}$ *after* σ might not occur in $\mathbb{M}$ (such a $\mathbb{M}$ exists, it can be constructed analogously to the proof of Lemma 8.27). To handle these pathological cases, all $\sigma \in Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$ with $\ddot{s} = init_{\mathcal{S}_{\tau^*\delta}}$ *after* σ must recurrently occur in the full TC seq to guarantee that the desired $\ddot{s}$ will eventually be reached. With full $Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}})$ coverage (or $faultable(Straces_{\mathcal{S}_{\tau^*\delta}}(init_{\mathcal{S}}))$ coverage if $\mathrm{P}_{exh \Rightarrow disch}$ is met) and TCs executed recurrently, test execution must eventually lead to a desired state if no fail occurs (recurrently) during test execution. □

When LazyOTF is active, i.e., selection from $in_{\mathcal{S}_{\tau^*}}(s)$ is not made randomly, $\underline{path2W}$ should give higher weights to paths that probably reach a TO faster, even if the current sub-graph contains no TO. But if it does, $\underline{path2W}(\pi^{full})$ must return a sufficiently high value for $\pi_{p_{curr}}$ containing a TO, so that after aggregation, WTCs that do contain a TO are heavier than WTCs without TOs, to guarantee P_{goal}. Hence Def. 12.32 differentiates paths to TOs and their weights from other paths and their weights.

Definition 12.32. Let $\mathcal{S} \in SPEC$, TO o and $\underline{aggWTCs}$ be given.
Then

- $paths^{TO}(\mathcal{S}_{det})^{r_{curr}} := \{\pi^{full} \in paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} \mid discharge_o(dest(\pi^{full})) = \mathbf{true}\}$
- $paths^{\not TO}(\mathcal{S}_{det})^{r_{curr}} := paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} \setminus paths^{TO}(\mathcal{S}_{det})^{r_{curr}}$
- $p2W^{TO} := \min_{\pi^{full} \in paths^{TO}(\mathcal{S}_{det})^{r_{curr}}} (\underline{path2W}(\pi^{full}))$
- $p2W^{\not TO} := \max_{\pi^{full} \in paths^{\not TO}(\mathcal{S}_{det})^{r_{curr}}} (\underline{path2W}(\pi^{full}))$.
- $\underline{aggWTCs}$ with $\underline{path2W}_o$ **does not suppress TOs** iff
  (the current sub-graph contains TOs $\Rightarrow$ genWTS using $\underline{aggWTCs}$ with $\underline{path2W}_o$ yields the highest values for WTCs containing a TO).

Lemma 12.33. *Let $w(\mathbb{W}_{p_{curr}})$ be implemented by genWTS via $\underline{aggWTCs}$ with $\underline{path2W}$. If $\underline{aggWTCs}$ with $\underline{path2W}$ does not suppress TOs, then $w(\mathbb{W}_{p_{curr}})$ implemented via genWTS meets P_{goal}.*

Proof. If LazyOTF is active, genWTS chooses a heaviest WTC. If the current sub-graph contains a TO and $\underline{aggWTCs}$ with $\underline{path2W}$ does not suppress TOs, each heaviest WTC contains a TO. □

Note. To implement P_{goal}, we could simply select the heaviest WTC only amongst those containing a TO. But since our weights should reflect meaningfulness, they should be configures such that weights of WTCs that contain a TO should be the heaviest anyways.

We investigate various $\underline{aggWTCs}$ and corresponding **inequalities between** $p2W^{TO}$ **and** $p2W^{\not TO}$, summarized in Lemma 12.34: $\underline{aggWTCs}(p2W, z_0, \ldots, z_{|L_U|})$ has many possibilities to aggregate (cf. Table 12.1). The value $p2W$ should always be incorporated in $\underline{aggWTCs}(p2W, z_0, \ldots, z_{|L_U|})$ because if test execution reached the WTC that is currently aggregated, its root has definitely been reached, i.e., it is the most certain node of the tree. We achieve this by taking the sum or the maximum between $p2W$ and the **rest**, i.e., the aggregate for $z_0, \ldots, z_{|L_U|}$. Due to the recursion in genWTS, this sum or maximum aggregates along paths, whereas the rest is the aggregation over all

possible branches. Since fail nodes also yield weights, we need not differentiate between the outputs, $z_1, \ldots, z_{|L_U|}$. To compute this rest, we introduce five aggregation variants (cf. Table 12.1) by considering MBT as a two person game between the tester and its **adversary**, which is the uncontrollable nondeterminism:

- $\underline{aggWTCs}(p2W, z_0, \ldots, z_{|L_U|}) = max(p2W, z_0, \ldots, z_{|L_U|})$ (***max(p2W, max(...))*** for short) results in a WTC weighing the maximal $\underline{path2W}$ it contains. Thus suppression is not possible if the obvious inequality $p2W^{TO} > p2W^{\not{TO}}$ is met. But a WTC with many heavy $\underline{path2W}$ is not heavier than a WTC with only one heavy $\underline{path2W}$;
- thus we consider $\underline{aggWTCs}(p2W, z_0, \ldots, z_{|L_U|}) = p2W + max(z_0, \ldots, z_{|L_U|})$ (for short: ***+(p2W, max(...))***), which is the optimistic approach of assuming the tester can take the most meaningful outgoing transition, i.e., the heaviest. This not only includes input, i.e., controllable nondeterminism, but also output, i.e., uncontrollable nondeterminism. Thus we consider a good adversary who picks the output in favor of the tester. The meaner the adversary gets, the less suitable this heuristic becomes, because it focuses only on the most meaningful path; a different WTC might contain many more promising paths, just not the optimal one (cf. also Note 12.14 on lowering risk). Fortunately, for fairness_{test} or fairness_{spec}, with forbidden underspec_U, the adversary cannot be very mean since all outputs must eventually occur (cf. Lemma 8.67). Otherwise this optimistic approach is too extreme in practice. The advantage is that weights only increase moving up a WTC if all weights are non-negative. Thus a WTC without TOs needs a long path with many heavy $\underline{path2W}$ to suppress TOs. So if $p2W^{TO} > b_{max} \cdot p2W^{\not{TO}}$, then weights for TOs are sufficiently high. Fig. 12.2 with $y \in L_I$ depicts an example where this inequality is not met since the left tree (reachable via δ) weighs 270, but the right tree (reachable via y) weighs only 170;
- $\underline{aggWTCs}(p2W, z_0, \ldots, z_{|L_U|}) = max(p2W, min(z_0, \ldots, z_{|L_U|}))$ (for short: ***max(p2W, min(...))***) and $\underline{aggWTCs}(p2W, z_0, \ldots, z_{|L_U|}) = p2W + min(z_0, \ldots, z_{|L_U|})$ (shortly ***+(p2W, min(...))***) are pessimistic approaches of taking the least meaningful transition, i.e., the lightest. So we consider a mean adversary who always picks the least meaningful output. This approach is usually too pessimistic even if we only have fairness_{model} or underspec_U in a strong form: Since one bad potential output can suppress TOs, the minimum is usually too extreme in practice. For instance, Fig. 12.2 with $a \in L_U$ shows that one small weight (the weight 0 reachable via the trace yba) is already sufficient to suppress a TO;
- thus **balanced aggregations**, which include all values in the aggregate, are usually better in practice. They all take the sum between $p2W$ and the rest. A simple solution is $\underline{aggWTCs}(p2W, z_0, \ldots, z_{|L_U|}) = p2W + z_0 + \cdots + z_{|L_U|}$ (***+(p2W, +(...))*** for short) since $w(\mathbb{W}) = \sum_{\pi \in paths(\mathcal{F}_w(\mathbb{W}))} (\underline{path2W}(\pi^{full} \cdot \pi))$. But for all $\pi \in paths(\mathcal{F}_w(\mathbb{W}))$, $\underline{path2W}(\pi^{full} \cdot \pi)$ is treated equally, independent of π's length, and weights of WTCs grow exponentially fast and can quickly cause an overflow if weights are bounded. Therefore, we prefer the following similar solutions;
- $\underline{aggWTCs}(p2W, z_0, \ldots, z_{|L_U|}) = p2W + arithmeticMean(z_0, \ldots, z_{|L_U|})$ (shortly ***+(p2W, mean(...))***) is the balanced aggregation that assumes **equidistribution** as the adversary's strategy, i.e., over all outgoing transitions, since we do

not know in advance which one will occur how often. Many heavy $\underline{path2W}$ on one path can again suppress TOs, though one such path likely gets evened out by taking the mean of $z_0, \ldots, z_{|L_U|}$. For instance in Fig. 12.2 with $L_U = \{a\}$, the heaviest path (with the trace $\delta b \delta$) gets evened out (by the paths with traces δa and $\delta b a$). On the other hand, for some $\pi^{full} \in paths^{TO}(\mathcal{S}_{det})^{r_{curr}}$, $\underline{path2W}(\pi^{full})$ might also be evened out, i.e., TO in $dest(\pi^{full})$ is suppressed if the containing WTC has many outgoing transitions to small weights and another WTC without TOs has mainly output transitions to heavy weights. For instance in Fig. 12.2 with $L_U = \{b\}, L_I = \{a, y\}$, if the left tree (reachable via δ) had all $\underline{path2W}(\pi^{full})$ of 70 (instead of 90 or 0), its weight would be 175, but the right tree (reachable via y) would only weigh 163. But for a given maximal bound b_{max}, $\underline{path2W}(\pi^{full})$ can be chosen sufficiently high for $\pi^{full} \in paths^{TO}(\mathcal{S}_{det})^{r_{curr}}$, such that TOs are never suppressed: $p2W^{TO} > branchout_{\mathcal{S}_{det}}^{(b_{max}-1)} \cdot b_{max} \cdot p2W^{\not TO}$. For instance as in Fig. 12.2 for $b_{max} = 4$ and $p2W^{\not TO} = 90$, $p2W^{TO} > 2^{(b-1)} \cdot b \cdot 90 = 2880$. Then equidistribution is a good approximation for all variants of fairness without underspec_U. For strong underspec_U, equidistribution is too rough an approximation; but we could quantify the probabilistic distribution on-the-fly (cf. Subsec. 12.4.5);

- a **pragmatic approach of quantifying probability** leads to an easy improvement that only distinguishes choices of the tester and choices of the adversary: For $p \in [0, \ldots, 1]$, $\underline{aggWTCs}(p2W, z_0, \ldots, z_{|L_U|}) = p2W + p \cdot z_0 + (1-p) \cdot arithmeticMean(z_1, \ldots, z_{|L_U|})$ ($\boldsymbol{+(p2W, +(p \cdot z_0, (1-p) \cdot mean(\ldots)))}$ for short) takes into account that often neither the performance of the SUT nor the timeout for quiescence varies over time. Thus p reflects the rate at which an output preempts input or quiescence. For the outputs, all solutions above can be applied. We focus on $arithmeticMean(z_1, \ldots, z_{|L_U|})$ since it is a balanced aggregation, can be improved by quantifying the probability distribution on-the-fly (cf. Subsec. 12.4.5), and evens out most paths with many heavy weights but no TO. Additionally, many output transitions to heavy weights do not suppress TOs as quickly as before if $p > 1/(|L_U|+1)$. For instance in Fig. 12.2 with $L_U = \{b\}, L_I = \{a, y\}$ and $p = 0.9$, the TO is not suppressed since the left tree (reachable via δ) weights 107, but the right tree (reachable via y) weighs 159. For $p = 0.5$, the TO does get suppressed because the left tree weighs 158 and the right 138. Usually $p > (1-p)/(branchout_{\mathcal{S}_{det}} - 1)$, in which case $p2W^{TO} > ((branchout_{\mathcal{S}_{det}} - 1)/(1-p))^{(b_{max}-1)} \cdot b_{max} \cdot p2W^{\not TO}$ avoids suppression.

Lemma 12.34. *Let $w(\mathbb{W}_{p_{curr}})$ be implemented by genWTS via $\underline{aggWTCs}$ with $\underline{path2W}$. Then $\underline{aggWTCs}$ with $\underline{path2W}$ does not suppress TOs if the following corresponding inequality is met:*

- $max(p2W, max(\ldots))$: $p2W^{TO} > p2W^{\not TO}$;
- $+(p2W, max(\ldots))$: $p2W^{TO} > b_{max} \cdot p2W^{\not TO}$;
- $+(p2W, mean(\ldots))$: $p2W^{TO} > branchout_{\mathcal{S}_{det}}^{(b_{max}-1)} \cdot b_{max} \cdot p2W^{\not TO}$;
- $+(p2W, +(p \cdot z_0, (1-p) \cdot mean(\ldots)))$: $p2W^{TO} > ((branchout_{\mathcal{S}_{det}} - 1)/(1-p))^{(b_{max}-1)} \cdot b_{max} \cdot p2W^{\not TO}$.

The user can choose from the pre-built (cf. Subsec. 13.2.3) $max(p2W, max(\ldots))$ (**MaxMax** for short), $+(p2W, max(\ldots))$ (**Max** for short), and $+(p2W, +(p \cdot z_0, (1-$

$p) \cdot mean(\dots))))$ (**PInputDefault** for short). With this selection, all practical classes of $\underline{aggWTCs}$ are already covered: using min is not practical, and PInputDefaultis the most practical balanced aggregation (cf. Subsec. 12.3.6). We set PInputDefaultas default since it is the most meaningful in practice (cf. Subsec. 14.3.5). However, the inequalities between $p2W^{TO}$ and $p2W^{\not{TO}}$ are simpler met for MaxMax and Max.

Example 12.35. Fig. 12.2 depicts a TC (from Subfig. 8.1a for $y = a$, also cf. Fig. 8.4), with $\underline{path2W}(\pi^{full})$ for each node.

Table 12.1 lists all $\underline{aggWTCs}$ mentioned above for the TC and $\underline{path2W}(\pi^{full})$ from Fig. 12.2 for $L_U = \{a, y\}, L_I = \{b\}$.

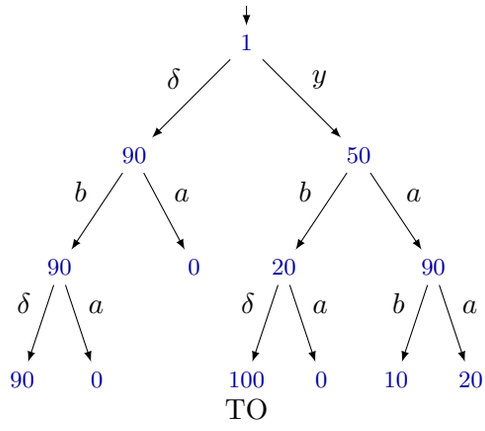


Figure 12.2.: A TC (for $L_U = \{a, y\}, L_I = \{b\}$) and all $\underline{path2W}(\pi^{full})$

Table 12.1.: Our $\underline{aggWTCs}$ for Fig. 12.2 with $L_U = \{a, y\}, L_I = \{b\}$

$\underline{aggWTCs}$	$w(\mathbb{W})$
$max(p2W, max(\dots))$	100
$+(p2W, max(\dots))$	271
$max(p2W, min(\dots))$	50
$+(p2W, min(\dots))$	71
$+(p2W, +(\dots))$	561
$+(p2W, mean(\dots))$	149
$+(p2W, +(0.9 \cdot z_0, 0.1 \cdot mean(\dots)))$	237

Notes. In cases where the tree in Fig. 12.2 is not output-enabled for the given L_U and L_I, we ignored this fact since it can be corrected easily and we wanted to keep the examples as simple as possible.

$\underline{aggWTCs}(p2W, z_0, \dots, z_{|L_U|})$ can be fine-tuned further, e.g., for $+(p2W, +(p \cdot z_0, (1 - p) \cdot mean(\dots)))$, δ can be treated exceptionally, since it usually takes longer than giving an output. This leads to the regular equidistribution $+(p2W, mean(\dots))$ for δ.

All $\underline{aggWTCs}$ that add values can theoretically cause an overflow if the weights are bounded. This, however, is unlikely if no TOs can be suppressed and TOs are inducing (cf. Sec. 12.2).

12.3.7. Composition of Test Objectives

Weight heuristics can be modularized according to features (or requirements or any other elements to be tested). Each feature is modeled by a test objectives, also called **basic test objective**. Composing basic test objectives lead to sophisticated heuristics, e.g., requirements coverage. Def. 12.36 gives a formal definition and extends Def. 11.6 to incorporate our meaningfulness weight heuristics.

Definition 12.36. Let $\mathcal{S} = (S, \to, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau)$. For our heuristics, a **test objective** $\boldsymbol{o}$ consists of

- $\boldsymbol{I_o^{lazy}} : S \to \Sigma_{lazy}$, indicating which states are inducing and test goals for o;
- $\boldsymbol{\underline{path2W}_o} : paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} \to \mathbb{N}, \pi^{full} \mapsto \underline{path2W}_o(\pi^{full})$,
  where higher values indicate that π^{full} is more meaningful for o;
- $\boldsymbol{active_o}$: has type $\mathbb{B}$ and is **true** iff o has not yet been discharged;
- $\boldsymbol{discharge_o} : paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} \twoheadrightarrow \mathbb{B}$, is the **discharge function for** o.
  Furthermore, discharging o must be feasible. Thus $discharge_o$ must be surjective.

If multiple basic TOs $o \in \ddot{o}$ are active, they must all be considered, i.e., their $\underline{path2W}_o$ aggregated and their I_o^{lazy} respected. Thus, they are composed to a new TO, as defined in Def. 12.37. LazyOTF functions with this composed TO as if it were a basic TO.

Definition 12.37. Let $\mathcal{S} = (S, \to, L_\tau) \in \mathcal{LTS}(L_I, L_U, \tau)$ and $\ddot{o}$ a finite, nonempty (finitely nested) set of TOs.

Then $\boldsymbol{\ddot{o}}$ can be considered a **composed test objective**, with

- $\boldsymbol{I_{\ddot{o}}^{lazy}} : S \to \Sigma_{lazy}, s \mapsto \max_{o \in \ddot{o}}(I_o^{lazy}(s))$;
- $\boldsymbol{\underline{path2W}_{\ddot{o}}} : paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} \to \mathbb{N}, \pi^{full} \mapsto \boldsymbol{\underline{aggPath2Ws}}($
  $(\underline{path2W}_o(\pi^{full}))_{o \in \ddot{o}})$, which aggregates the $\underline{path2W}_o(\pi^{full})$ of all $o \in \ddot{o}$;
- $\boldsymbol{active_{\ddot{o}}}$: has type $\mathbb{B}$ and is **true** iff $\forall o \in \ddot{o} : active_o$;
- $\boldsymbol{discharge_{\ddot{o}}} : paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} \twoheadrightarrow \mathbb{B}, \pi \mapsto \bigvee_{o \in \ddot{o}} discharge_o(\pi^{full})$.

$\boldsymbol{\ddot{o}_{init}}$ is the **set of initially active TOs**, i.e., when phase 1 begins.

Notes 12.38. So when using composition of TOs, the composed TO that comprises all currently active basic TOs is currently active.

Since $\ddot{o}$ is not ordered, $\underline{aggPath2Ws}$ must be symmetric in its parameters.

As with other heuristics (cf. Subsec. 12.3.1), compositional TOs can be used as guidance or as metric to give feedback to the test engineer and to decide whether to exit testing. To turn $\ddot{o}$ into a metric (similarly to the conformance index in [Arcaini et al., 2013]), we define the **TO coverage level** to be $1 - |\ddot{o}|/|\ddot{o}_{init}|$.

If we want to nest composition, $\underline{aggPath2Ws}$ should be homomorphic, i.e., for $\ddot{o} = \{\ddot{o}_1, \ddot{o}_2\}$ with $\ddot{o}_1 \cap \ddot{o}_2 = \varnothing$, $\underline{path2W}_{\ddot{o}}(\pi^{full}) = \underline{aggPath2Ws}((\underline{path2W}_{\ddot{o}_1}(\pi^{full}), \underline{path2W}_{\ddot{o}_2}($

$\pi^{full}))) = \underline{aggPath2Ws}(\underline{aggPath2Ws}((\underline{path2W}_o(\pi^{full}))_{o\in\ddot{o}_1}), \underline{aggPath2Ws}((\underline{path2W}_o(\pi^{full}))_{o\in\ddot{o}_2})) \overset{!}{=} \underline{aggPath2Ws}((\underline{path2W}_o(\pi^{full}))_{o\in\ddot{o}_1\dot{\cup}\ddot{o}_2}) = \underline{path2W}_{\ddot{o}_1\dot{\cup}\ddot{o}_2}(\pi^{full})$.

As described in Subsec. 12.3.2, composition is also helpful in practice to combine

- more lightweight heuristics, to ensure efficiency for simpler but also more relevant TOs;
- and heavyweight heuristics, to ensure rigor, or even exhaustiveness.

We implemented interfaces and abstract classes for flexible $\underline{aggPath2Ws}$ (cf. Subsec. 13.2.3), *max* as default and *sum* as alternative. The more balanced $\underline{aggPath2Ws}$ takes different TOs $o \in \ddot{o}$ into account (similar to balanced $\underline{aggWTCs}$), the more difficult exhaustiveness and $\mathrm{P}_{discharge}$ becomes, but the more synergetic TO $\ddot{o}$ becomes.

Since we do not know the order in which the TOs $o \in \ddot{o}_{init}$ are discharged, we would have to check the provisos for each subset $\ddot{o}$ of $\ddot{o}_{init}$ (i.e., for each $\underline{path2W}_{\ddot{o}}$) to be able to give guarantees when using composed TOs with no further restrictions on the TOs $o \in \ddot{o}_{init}$. Since this becomes infeasible quickly, we search for conditions when the provisos are preserved by composition, i.e., we investigate how the provisos can be strengthened such that composition preserves them.

$\mathrm{P}_{\rightarrow OTF}$, $\mathrm{P}_{coverage}$ and $\mathrm{P}_{coverViaTOs}$ are already modular, i.e., they cover compositional TOs, too. For $\mathrm{P}_{vanishing}$, $\underline{aggPath2Ws}$ must correctly handle the set of active TOs being empty (cf. next paragraph). To fulfill $\mathrm{P}_{discharge}$, e.g., via P_{phase}, P_{goal} and P_{fair}, we lift our previous lemmas to composed TOs as defined in Def. 12.37. For this, we firstly consider $\underline{aggPath2Ws}$, then $\underline{aggWTCs}$ and finally $\underline{path2W}$.

$\underline{aggPath2Ws}$

$\underline{aggPath2Ws}_{\ddot{o}}$ can include all values $\underline{path2W}_o$ for all $o \in \ddot{o}$, e.g., via *mean* or *sum* (cf. balanced aggregations of $\underline{aggWTCs}$ in the previous subsection). But this can suppress a TO and thus break P_{goal} (similar to $\underline{aggWTCs}$), especially if we generate TOs automatically (e.g., for a coverage criterion) and hence have no control over the size of $\ddot{o}$. Fortunately, we can use *max* for $\underline{aggPath2Ws}$ without the risk of being too optimistic: unlike $\underline{aggWTCs}$, we have no uncontrollable nondeterminism (i.e., no mean adversary) for $\underline{aggPath2Ws}$ since the tester can decide which TO to focus on. So we mostly use *max* as $\underline{aggPath2Ws}$, and set $\underline{path2W}_{\varnothing}(\cdot) = 0$, such that $\mathrm{P}_{vanishing}$ is also fulfilled.

Example. An example for suppressed TOs can be derived from the TC of Fig. 12.2 for $\ddot{o} := \{o_1, o_2\}$, $\underline{aggPath2Ws}$ the *sum* or the *mean*, and $L_U = \{a, y\}, L_I = \{b\}$. Let the basic TO o_1 have $\underline{path2W}_{o_1}$ as depicted in Fig. 12.2; let o_2 be another basic TO that is equal to o_1 except for

- the node reachable via the trace $yb\delta$ no longer discharges a TO and has the weight 0 instead of 100;
- the node reachable via the trace yba now discharges a TO and has the weight 100 instead of 0;
- the node reachable via the trace ya now has the weight 0 instead of 90.

Then the TOs are not suppressed for $+(p2W, min(\dots))$; but for all six other variants of $\underline{aggWTCs}$ listed in Table 12.1, the TOs are suppressed by the left tree (reachable via δ).

aggWTCs

The inequalities between $p2W^{TO}$ and $p2W^{\not TO}$ from Lemma 12.34 still guarantee that TOs are not suppressed since Def. 12.32 can be lifted to a compositional test objectives $\ddot{o}$ because $\ddot{o}$ is discharged in a superstate $\ddot{s}$ iff some $o \in \ddot{o}$ is discharged in $\ddot{s}$ (but $\ddot{o} \setminus \{o\}$ is not discharged).

path2W$_o$

For fairness$_{test}$ or fairness$_{spec}$, with forbidden underspec$_U$, many implementations of *path2W$_o$* fulfill our provisos, in dependence of *aggWTCs* and further heuristics settings. The investigation of *path2W* in Subsec. 12.3.6 showed that for fairness$_{test}$ or fairness$_{spec}$, with forbidden underspec$_U$, P$_{phase}$ and P$_{fair}$ can be met by choosing $w(\mathbb{W}_{p_{curr}})$ as *phaseVariant* and guaranteeing that it is (finitely or strictly) monotonically increasing for at least one resolution of uncontrollable nondeterminism. By Lemmas 12.40 and 12.41, this holds if *path2W$_o$* for each (basic or composed) TO o is strictly (or only finitely) monotonically increasing towards o, as defined in Def. 12.39. Depending on *aggWTCs*, the additionally required settings are either met easily or are strong restrictions. The lemmas also cover the weaker constraints of finitely (as opposed to strictly) monotonically increasing towards TOs, because this is relevant in practice: a value *path2W$_o$*(π^{full}) is often derived by a certain aspect of $dest(\pi^{full})$ that reflects the distance to o, e.g., a variable evaluation. These aspects often stutter, i.e., do not change for each transition in $\mathcal{S}$ (cf. Example 12.43).

Definition 12.39. Let $\mathcal{S} \in SPEC$, $\pi^{full} \in paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}}$ with $\pi^{full}{=}(\pi_i)_{i\in[1,\dots,1+p_{curr})}$ and o be a TO, and $paths_o^{TO}(\mathcal{S}_{det})^{>r_{curr}} := \bigcup_{k\in\mathbb{N}_{>0}} paths_o^{TO}(\mathcal{S}_{det})^{r_{curr}+k}$.

Then $\boldsymbol{distance_o} : paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} \to \mathbb{N}_{\geq 0}$,

$$\pi^{full} \mapsto \begin{cases} \min\limits_{\pi_e^{full} \in paths_m^{TO}(\mathcal{S}_{det})^{>r_{curr}} \text{ that extends } \pi^{full}} (|\pi_e^{full}| - |\pi^{full}|), \text{ with } m = o \\ \text{if } o \text{ is a basic TO, otherwise } m = \{n \in o | \underline{path2W}_n(\pi^{full}) = \\ \max\limits_{r\in o}(\underline{path2W}_r(\pi^{full}))\} \qquad \text{if } \nexists n \in [0,\dots,|\pi^{full}|] : discharge_o(\pi^{full}_{\leq n}) \\ 0 \qquad \text{otherwise.} \end{cases}$$

π^{full} **moves towards o in the current traversal sub-phase** $:\Leftrightarrow$ $distance_o(\pi^{full}) = distance_o((\pi_i)_{i\in[1,\dots,p_{curr})}) - |\pi_{p_{curr}}|$.

path2W$_o$ is:

- **finitely monotonically increasing towards o** $:\Longleftrightarrow$
  $\forall \pi^{full} \in paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} : \big(distance_o(\pi^{full}) < distance_o(\pi^{full}_{\leq |\pi^{full}|-1})$
  $\implies \underline{path2W}_o(\pi^{full}) \geq \underline{path2W}_o(\pi^{full}_{\leq |\pi^{full}|-1})\big)$
- **strictly monotonically increasing towards o** $:\Longleftrightarrow$
  $\forall \pi^{full} \in paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} : \big(distance_o(\pi^{full}) < distance_o(\pi^{full}_{\leq |\pi^{full}|-1})$
  $\implies \underline{path2W}_o(\pi^{full}) > \underline{path2W}_o(\pi^{full}_{\leq |\pi^{full}|-1})\big)$.

Lemma 12.40. *Let* $\underline{aggPath2Ws} = max$.

Then composition preserves that *path2W* *is finitely (resp. strictly) monotonically increasing towards the TO: for a set $\ddot{o}$ of TOs with $\forall o \in \ddot{o}$:* *path2W$_o$* *is finitely (resp.*

strictly) monotonically increasing towards o, $\underline{path2W}_{\ddot{o}}$ is finitely (resp. strictly) monotonically increasing towards $\ddot{o}$.

Proof. Let $\ddot{o}$ be a set of TOs with $\forall o \in \ddot{o}$: $\underline{path2W}_o$ is finitely (resp. strictly) monotonically increasing towards o. Let $\pi^{full} \in paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}}$ with $distance_{\ddot{o}}(\pi^{full}) < distance_{\ddot{o}}(\pi^{full}_{\leq|\pi^{full}|-1})$. Then $\exists o_m \in \ddot{o}$: $\underline{path2W}_{\ddot{o}}(\pi^{full}) = \max_{o\in\ddot{o}}(\underline{path2W}_o(\pi^{full})) \geq \underline{path2W}_{o_m}(\pi^{full}) \geq \underline{path2W}_{o_m}(\pi^{full}_{\leq|\pi^{full}|-1}) = \max_{o\in\ddot{o}}(\underline{path2W}_o(\pi^{full}_{\leq|\pi^{full}|-1})) = \underline{path2W}_{\ddot{o}}(\pi^{full}_{\leq|\pi^{full}|-1})$ (resp. $\underline{path2W}_{o_m}(\pi^{full}) > \underline{path2W}_{o_m}(\pi^{full}_{\leq|\pi^{full}|-1})$). Therefore, $\underline{path2W}_{\ddot{o}}$ is finitely (resp. strictly) monotonically increasing towards $\ddot{o}$. □

Lemma 12.41. *Let o be a (basic or composed) TO with $\underline{path2W}_o$ finitely (resp. strictly) monotonically increasing towards o. If*

1. *either $\underline{aggWTCs} = max(p2W, max(\dots))$;*
2. *or $\underline{aggWTCs} = +(p2W, max(\dots))$ and all $\underline{path2W}_o$ are non-negative and b_+ is monotonically increasing and I^{lazy}_o retains one successor state reachable via an outgoing transition that moves towards o (i.e., does not set it to INDUCING);*
3. *or $\underline{aggWTCs}$ is balanced (e.g., $+(p2W, +(\dots))$ or $+(p2W, mean(\dots))$ or $+(p2W, +(p{\cdot}z_0, (1{-}p){\cdot}mean(\dots)))$) and all $\underline{path2W}_o$ are non-negative and ($\underline{path2W}_o(\pi^{full}) > 0$ iff $I^{lazy}{}_o(dest(\pi^{full}))$=TESTGOAL, cf. Example 12.43),*

then $w(\mathbb{W}_{p_{curr}})$ is finitely monotonically increasing for at least one resolution of uncontrollable nondeterminism, until o is in the current sub-graph.

Proof. Let $(\mathbb{W}_i)_{i\in[1,\dots,1+p_{curr})}$ be the weighted test case sequence, $p := p_{curr} \geq 2$ (wlog, since otherwise LazyOTF already terminated with discharged o), with o not in the sub-graphs of the traversal sub-phases p, $p-1$. Furthermore, $\pi_{prep} := (\pi^{full}_{r_{curr}})_{\leq|t_{curr}|-|\pi_p|}$ and $\pi_{prep-1} := (\pi^{full}_{r_{curr}})_{\leq|t_{curr}|-|\pi_p|-|\pi_{p-1}|}$.

Case 3: $\underline{path2W}_o(\pi^{full}) = 0$, $w(\mathbb{W}_p) = 0 \geq 0 = w(\mathbb{W}_{p-1})$.

Cases 1 and 2: we can choose a resolution of nondeterminism such that each outgoing transition of π_{p-1} reduces the distance to o, so that in those steps, $\underline{path2W}_o$ increases: For output, the premise allows this; if input is the only possibility to advance successfully (i.e., leave the state without a fail), it also reduces the distance to o. From the choices that reduce the distance to o, the chosen resolution avoids inducing states as long as possible. So if $w(\mathbb{W}_{p-1}) = M \in \mathbb{N}$, we have $\underline{path2W}_o(\pi_{prep}) = M$ for this resolution of nondeterminism.

Case 1: $w(\mathbb{W}_{p-1}) = M = \underline{path2W}_o(\pi_{prep}) \leq \max_{\pi\in\mathcal{F}_w(paths(\mathbb{W}^{full}_{r_{curr}}))}(\underline{path2W}_o(\pi)) = w(\mathbb{W}_p)$.

Case 2: $b_p \geq b_{p-1}$ since o is not in the sub-graph of the traversal sub-phase $p-1$. With analogous resolution of nondeterminism in traversal sub-phase p as in the traversal sub-phase $p-1$, we have $w(\mathbb{W}_{p-1}) = M \leq b_{p-1}\cdot\underline{path2W}_o(\pi_{prep}) \leq b_p\cdot\underline{path2W}_o(\pi_{prep}) \leq w(\mathbb{W}_p)$.

Therefore, for all three cases, $w(\mathbb{W}_{p_{curr}})$ is finitely monotonically increasing for at least one resolution of uncontrollable nondeterminism, until o is in the current sub-graph. □

Def. 12.39 and Lemmas 12.40 and 12.41 allowed $\underline{path2W}_o$ to have $\underline{path2W}_o(\pi^{full}) > \underline{path2W}_o(\pi^{full}_{\leq|\pi^{full}|-1})$ if $distance_o(\pi^{full}) \geq distance_o(\pi^{full}_{\leq|\pi^{full}|-1})$, i.e., we only constrained

$\underline{path2W_o}$ when moving towards o. Consequently, the guidance heuristic may lead away from o, i.e., the TOs that LazyOTF moves towards can change. But this can happen only finitely often if the weights are bounded since $w(\mathbb{W}_{p_{curr}})$ is finitely monotonically increasing. Since such weak constraints on $\underline{path2W_o}$ are counter-intuitive, inefficient, too weak for unbounded weights and usually not necessary, we can employ stricter constraints (i.e., they imply the weak constraints given in Def. 12.39), as defined in Def. 12.42.

Definition 12.42. Let $\mathcal{S} \in SPEC$, $\pi^{full} \in paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}}$ with $\pi^{full} = (\pi_i)_{i\in[1,\dots,1+p_{curr})}$ and o be a TO.

Then $\underline{path2W_o}$ is:

- **finitely monotonical towards and away from o** $:\Longleftrightarrow$
  $\forall \pi^{full} \in paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} : \big(distance_o(\pi^{full}) < distance_o(\pi^{full}_{\leq|\pi^{full}|-1}) \Longleftrightarrow$
  $\underline{path2W_o}(\pi^{full}) \geq \underline{path2W_o}(\pi^{full}_{\leq|\pi^{full}|-1})\big)$
- **strictly monotonical towards and away from o** $:\Longleftrightarrow$
  $\forall \pi^{full} \in paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} : \big(distance_o(\pi^{full}) < distance_o(\pi^{full}_{\leq|\pi^{full}|-1}) \Longleftrightarrow$
  $\underline{path2W_o}(\pi^{full}) > \underline{path2W_o}(\pi^{full}_{\leq|\pi^{full}|-1})\big)$.

Example 12.43. Often, we employ $\underline{path2W_o}$ of one of the following types: $\underline{path2W_o} : paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}} \to \mathbb{N}_{\geq 0}$, $\pi^{full} \mapsto$

- $\mathbf{nonfancy}\ (\pi^{full}) := \begin{cases} k_0 & \text{if } {I^{lazy}}_o(dest(\pi^{full})) = \text{ORDINARY} \\ k_1 & \text{if } {I^{lazy}}_o(dest(\pi^{full})) = \text{INDUCING} \\ k_2 & \text{if } {I^{lazy}}_o(dest(\pi^{full})) = \text{TESTGOAL}; \end{cases}$
- $\mathbf{fancy}_{linear}\ (\pi^{full}) := \begin{cases} k_0 - k_1 \cdot distance_o(\pi^{full}) & \text{if } distance_o(\pi^{full}) \neq 0 \\ k_2 & \text{if } distance_o(\pi^{full}) = 0 \end{cases}$
  (modulo stuttering);
- $\mathbf{fancy}_{nonlinear}\ (\pi^{full}) := \begin{cases} k_o + round(k_1/distance_o(\pi^{full})) & \text{if } distance_o(\pi^{full}) \neq 0 \\ k_2 & \text{if } distance_o(\pi^{full}) = 0 \end{cases}$
  (modulo stuttering).

The distance is often measured on the STS level with the help of a location variable. With appropriate $k_i \in \mathbb{N}_{\geq 0}$, these $\underline{path2W_o}$ are finitely monotonical towards and away from o. The result of fancy_{linear} can become negative for large distances. But if the distance is measured with the help of a variable with bounded domain, k_i can be chosen such that all results are non-negative. Alternatively, we can take the maximum of the result and 0, for which fancy_{linear} remains finitely monotonical towards and away from o. Subsec. 14.3.3 contains examples with concrete values.

12.3.8. Countermeasures for Unmet Provisos

For cases where the premise of Lemma 12.41 is not met (e.g., in Example 12.44), $w(\mathbb{W}_{p_{curr}})$ might not be finitely monotonically increasing for any resolution of uncontrollable nondeterminism. So the increase of $\underline{path2W_o}$ towards o is distorted by the weights of the other reachable paths.

Example 12.44. This example uses a balanced $\underline{aggWTCs}$ and $\underline{path2W_o}$ that are strictly monotonically increasing towards o, but are more sophisticated, so that the premise of

Lemma 12.41 is not met. Fig. 12.3 shows that for $o = o_1$ and $o = o_2$ with $\underline{path2W}_o$ strictly monotonically increasing towards o, $\ddot{o} = \{o_1, o_2\}$ also has $\underline{path2W}_{\ddot{o}}$ that is strictly monotonically increasing towards $\ddot{o}$. But for $\underline{aggWTCs} = +(p2W, mean(\dots))$, both successor WTCs of $\mathbb{W}_1$ are lighter, so $w(\mathbb{W}_{p_{curr}})$ is not finitely monotonically increasing.

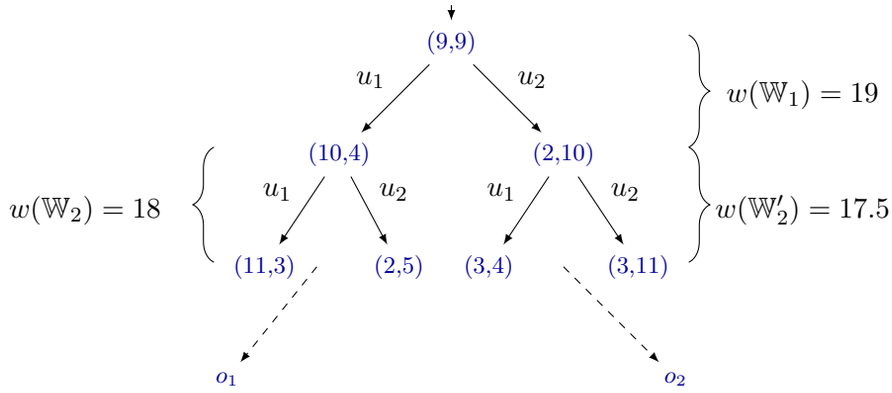


Figure 12.3.: $\mathcal{S}$ with strictly monotonically increasing (path2W_{o_1},path2W_{o_2}) towards the TOs and balanced aggWTCs resulting in no finitely monotonically increasing $w(\mathbb{W}_{p^{curr}})$

Without $w(\mathbb{W}_{p_{curr}})$ finitely monotonically increasing, we can no longer use it as argument to guarantee P_{phase}. But this does not necessarily mean that P_{phase} and $\text{P}_{discharge}$ is not met: If there is a TO o that the $\mathbb{W}_{p_{curr}}$ keep moving towards, even if $w(\mathbb{W}_{p_{curr}})$ decrease, a sub-graph containing a TO will eventually be reached. This scenario can for instance occur when b_+ is not monotonically increasing, but otherwise the premise of case 2 in Lemma 12.41 is met.

But in general, if $w(\mathbb{W}_{p_{curr}})$ is not finitely monotonically increasing, we have no guarantee that the heaviest WTCs keep moving towards o. As long as LazyOTF only changes finitely often between TOs it is moving towards, a sub-graph containing a TO will still eventually be reached. Thus, because there are only finitely many TOs, P_{phase} is still met if the sequence $(\ddot{o}_i)_{i\in[1,\dots,1+p_{curr})}$ of TOs that LazyOTF is moving towards in each phase i does not keep cycling through a **period $(\ddot{o}_i)_{i\in[1,\dots,p]}$** with $\bigcap_{i\in[1,\dots,p]} \ddot{o}_i = \varnothing$, baptized **alternating TO period**. Infinitely cycling through an alternating TO period never occurred in our experiments (cf. Sec. 14.3); it only happens in the unlikely situation that the balanced $\underline{aggWTCs}$ distorts the increase of $\underline{path2W}_o$ towards o, and this results in a choice $i_\delta \in L_I \dot\cup \{\delta\}$ moving towards an o that we moved away from before, and these alternations form a TO period, and no output that the SUT makes while cycling exits the alternating TO period, and the dynamic bound heuristic does not break the alternating TO period, i.e., the alternating TO period must be sustained by all chosen bounds. But a finite amount of cycles occurs often due to uncontrollable nondeterminism (cf. Subsec. 8.7.1, Subsec. 14.3.9). Infinitely cycling through an alternating TO period can be avoided by any of the following **countermeasures**:

- change $\underline{aggWTCs}$ to one that meets the premise of Lemma 12.41;
- or change $\underline{path2W}_o$ (and for case 2 possibly I_o^{lazy} and b_+) for the given $\underline{aggWTCs}$ such that the premise of Lemma 12.41 is met. For $\underline{aggWTCs} = +(p2W, max(\dots))$

with some negative $\underline{path2W}_o$, computations can easily be transformed to purely non-negative weights. But modifying I_o^{lazy} to retain at least one outgoing transition that moves towards o is either implemented easily and strongly restrictive, or less restrictive but complex to implement;
- simply changing the bound b in $\underline{aggWTCs} = +(p2W, mean(\dots))$ or $\underline{aggWTCs} = +(p2W, +(p \cdot z_0, (1-p) \cdot mean(\dots)))$, or the probability distribution in the latter, may resolve the TO period;
- for finite $\mathcal{S}$, use $\underline{finingPath2W}$;
- use mitigation.

Since unbounded cycling through an alternating TO period occurs rarely (never in our experiments, cf. Sec. 14.3), it is useful in practice to accept the risk and only add one of the above countermeasures if alternating TO periods cause problems.

LazyOTF offers a simple **cycle warning** to warn about alternating TO periods for finite $\mathcal{S}$: For finite $\mathcal{S}$, infinite cycling through an alternating TO period only occurs if we infinitely traverse a cycle in $\mathcal{S}$. Furthermore, for finite $\mathcal{S}$ we infinitely traverse a cycle in $\mathcal{S}$ iff we infinitely visit some state. LazyOTF's cycle warning implementation counts how often a superstate in $\mathcal{S}$ is visited without any discharge in between (i.e., without making any progress, cf. Chapter 6). A **cycle warning** is issued whenever some superstate is visited c times, where the user-supplied **cycle warning threshold** c reflects how mean the adversary can be. For fairness_{test} or fairness_{spec}, without underspec_U, false positive!cycle warning can often be avoided if c is chosen sufficiently large (cf. Lemma 8.67 and Subsec. 14.3.9). But for fairness_{model} or complex TOs, cycles might have to be traversed often. This cycle warning can also be used as a more general analytical tool to detect weak guidance heuristics, or mean adversaries, or hot spots in $\mathcal{S}_{det}$ during LazyOTF's traversal and test execution sub-phases, which indicates hot spots in the application.

12.3.9. Related Work on Guidance via Weight Heuristics

There is little related work that uses weights to implement guidance heuristics similar to ours. Hence we broaden the scope and investigate the use of weights for a kind of guidance in KLEE (cf. Subsec. 7.3.2). Then we consider the relation to search-based software testing (cf. Subsec. 11.2.4, Subsec. 12.3.1, and Note 12.30). Finally, we compare our guidance via weight heuristics to the cycling heuristics (cf. Subsec. 12.3.1) and NPC checks (cf. Chapter 6).

KLEE is related work when broadening the scope to also considering gray-box and white-box testing, since KLEE uses weights to prioritize the investigation of symbolic paths. But weights are used only to increase the code coverage level. Furthermore, KLEE only weighs paths, but not trees, i.e., uncontrollable nondeterminism is not supported (cf. Sec. 11.4). KLEE executes concrete instructions immediately, which generally forbids backtracking, and hence also the investigation of future transitions (except for the direct outgoing transition), which renders the guidance heuristic weak, as for on-the-fly MBT.

Within search-based software testing, the metrics used for prioritization can roughly be considered as weights similar to ours. Most of them are used to achieve a high coverage level for a classical coverage criterion. But several techniques from search-based software testing, like metaheuristics, can be transferred to other prioritization metrics

than classical coverage criteria; this has been done in this thesis for general weights (cf. Note 12.30 and Subsec. 12.3.7). Furthermore, parallelization is rarely considered in search-based software testing, but in this thesis, in Sec. 11.5. PGA (cf. Subsec. 11.5.1) is one of the few works on parallelization in search-based software testing. Since the choices of $\mathbb{W}_i$ in LazyOTF's phases (cf. Subsec. 12.3.4) can roughly be considered as steepest ascent hill climbing (cf. Note 12.30), and i_δ caching (cf. Subsec. 13.6.2) as variation of tabu search, distributed LazyOTF can also be roughly considered as parallelization in search-based software testing. Further comparisons and advantages of LazyOTF over genetic algorithms is given in Subsec. 12.3.1.

We first consider the most restrictive case of the cycling heuristics: forbidding all cycles. For fairness_{test} with forbidden underspec_U, we need not traverse cycles in $\mathcal{S}$ to reach all states (cf. Lemma 8.66), and hence not for exhaustiveness (cf. Lemma 8.61) and for $\text{P}_{discharge}$ if $\text{P}_{exh \Rightarrow disch}$ is met. But this is not the case for more complex TOs (e.g., o_{dec}, cf. Subsec. 14.3.3) and not efficient since allowing a cycle when the SUT made an undesired choice in uncontrollable nondeterminism might cost much less than having to restart in $init_{\mathcal{S}}$. For better performance, or weaker fairness, or allowed underspec_U, we can allow a certain amount of cycles via less restrictive cycling heuristics, e.g., by integrating incremental deepening on the number of cycle unwindings i (analogously to $\text{DFS}_{\text{incremental}}$, cf. Subsec. 6.4.1) into genWTS. Alternatively, we can allow only specific cycles via progress states by integrating NPC checks into genWTS, and pruning whenever an NPC is detected. For direct and efficient guidance, cycles that do not make progress towards coverage or discharging TOs should be avoided, so the test engineer has to accurately model which cycles are making progress, e.g., by marking states or transitions in those cycles as progress. With more and more relaxed cycling, successively larger *Straces* coverage can be achieved. A *finingPath2W* wrapper can also be used to easily implement a variant of the cycling heuristic (cf. Subsec. 12.3.1): *cycleFiningPath2W* fines paths the more cycles (without discharges) they have. All these cycling heuristics operate on-the-fly (as opposed to [Feijs et al., 2002; Goga, 2003]).

Because different cycles and progress cycles have no priority, guidance is not sufficiently fine-grained to guide effectively towards TOs, i.e., meaningfulness is indicated less accurately. Thus using weights are more suitable, and can also be used to implement variants of the cycling heuristic. Hence we investigated weights in this chapter and leave all alternatives via cycling heuristics from this subsection as future work.

12.4. Optimizations

LazyOTF's flexible heuristics and open implementation allow many optimizations of its resource consumption, its meaningfulness, and its usability. This section introduces three implemented optimizations and two for future work:

The interplay between the phase heuristics (cf. Sec. 12.2) and the guidance heuristics (cf. Sec. 12.3) has been optimized for lower runtime and memory requirements: Since states directly after the current sub-graph have not been considered at all in the last traversal sub-phase, incorporating them at the end of the current test execution sub-phase can yield improvements in two special cases:

- if they do not contain controllable nondeterminism, optimizations via **lazy traversal sub-phases** are possible;

- if they are lighter than alternative states (i.e., other states reachable with the same amount of steps), optimizations via **eager micro-traversal sub-phases** are possible.

Besides these two implemented optimizations, **reproducibility** of failures can be optimized by tuning our flexible heuristics, without any modifications to our code. This is important in practice for usability, but difficult to solve for SUTs with uncontrollable nondeterminism.

Interesting future work is optimizing our bound heuristics by considering further dynamic information to increase performance, and quantifying nondeterminism to generate further dynamic information to increase meaningfulness.

Feasibility can be improved best if optimizations can be combined efficiently (cf. Chapter 6). For LazyOTF, parallelization, lazy traversal sub-phases and eager micro-traversal sub-phases are orthogonal and can be combined efficiently (cf. Sec. 14.3). They do not restrict the heuristics, so reproducibility can additionally be improved.

12.4.1. Lazy Traversal Sub-phases

If a test execution sub-phase ends with the trace $\sigma_{r_{curr}}^{full}$ and $in_{\mathcal{S}_{\tau^*}}(\ddot{s}) = \varnothing$ with $\ddot{s} = init_{\mathcal{S}}\ after\ \sigma_{r_{curr}}^{full}$, i.e., $\ddot{s}$ does not contain controllable nondeterminism, there is no reason to start the next traversal sub-phase at $\ddot{s}$. Therefore, the **lazy traversal sub-phase optimization** extends the current test execution sub-phase (and therefore the trace $\sigma_{r_{curr}}^{full}$) via OTF (i.e., via test steps on the SUT and in $\mathcal{S}$ in lockstep) until a superstate is reached with $in_{\mathcal{S}_{\tau^*}}(init_{\mathcal{S}}\ after\ \sigma_{r_{curr}}^{full}) \neq \varnothing$. Then the next traversal sub-phase begins. Lazy traversal sub-phase optimization can reduce runtime and memory requirements if sequences of superstates without controllable nondeterminism exist in $\mathcal{S}_{det}$, especially if those states have high uncontrollable nondeterminism, i.e., a large branching, since the next traversal sub-phase and corresponding TC would need to cover all resolutions of uncontrollable nondeterminism, whereas test execution only gives one result.

Example. Fig. B.1 in Appendix B.1.1 gives an example with a clean web service following the request-response pattern: in all $*$Requested states, output only occurs together with quiescence, but no input. Input is, however, again possible in all successor states, so traversal sub-phases are delayed at most one step.

12.4.2. Eager Micro-traversal Sub-phases

Since the traversal sub-phase p_{curr} choses a TC $\mathbb{T}_{p_{curr}}$ among the heaviest, and since exhaustiveness, coverage or discharging is guaranteed if a corresponding proviso holds, $\mathbb{T}_{p_{curr}}$ on the whole is meaningful. However, since the traversal sub-phase does not consider the states directly after the current sub-graph, $\mathbb{T}_{p_{curr}}$'s last resolutions of controllable nondeterminism (i.e., its last i_δ transitions) might not be the most meaningful if the weight heuristics cannot accurately indicate meaningfulness at the granularity of single transitions (e.g., using variable evaluations). Thus the **eager micro-traversal sub-phase optimization** (also called **MarginSafetyMiniTT** or **OracleSafetyTree**) does consider states directly after the current sub-graph before the last test steps of $\mathbb{T}_{p_{curr}}$ are executed, as defined in Def. 12.45.

Definition 12.45. Let $b_{exec}, b_{future} \in [1, b_{p_{curr}})$, $\mathbb{T}_{p_{curr}} = (S, \rightarrow, L_\delta)$ and $s \in S$ the first state during the test execution sub-phase p_{curr} such that $\exists \pi \in paths_{max}(\mathbb{T}_{p_{curr}})$ with $dest(\pi) = \text{pass}$ and $\exists \pi_1, \pi_2$ with $\pi = \pi_1 \cdot \pi_2$, $dest(\pi_1) = s$ and $|\pi_2| \leq b_{exec}$. Furthermore, let $\mathbb{T}'$ be the complete subtree of $\mathbb{T}_{p_{curr}}$ rooted in s.

Then the **eager micro-traversal sub-phase** with the parameters $\boldsymbol{b_{exec}}$, $\boldsymbol{b_{future}}$ considers all possible b_{future} states directly after the current sub-graph before the last b_{exec} test steps of $\mathbb{T}_{p_{curr}}$ are executed: It performs a small traversal sub-phase with the bound $b_{exec} + b_{future}$ the moment s is reached, to check whether there exists a heaviest TC with the bound $b_{exec} + b_{future}$ that extends $\mathbb{T}'$.

- If so, $\mathbb{T}_{p_{curr}}$ contains no inefficiency in the end, and test execution of $\mathbb{T}'$ is performed, too, i.e., $\mathbb{T}_{p_{curr}}$ is fully executed and then the next traversal sub-phase is started, as without the optimization.
- If not, $\mathbb{T}_{p_{curr}}$ contains an inefficiency in the end, so $\mathbb{T}'$ is not executed and the next traversal sub-phase is started in s already, avoiding the less meaningful $\mathbb{T}'$.

b_{future} should be chosen small to avoid increased runtime due to the additional traversal sub-phases. In summary, traversal sub-phases and their TCs guarantee exhaustiveness, coverage or discharging, whereas eager micro-traversal sub-phases avoid inefficiencies in the end of those TCs due to coarse-grained weight heuristics.

Example. Again Fig. B.1 in Appendix B.1.1 following the request-response pattern is an example: most ∗Input transitions represent requests and only the corresponding ∗Output transitions update location variables as response. Hence the weight heuristics do not sufficiently reflect meaningfulness for ∗Input transitions. Therefore, eager micro-traversal sub-phases do improve meaningfulness, but $b_{exec} = b_{future} = 1$ is sufficient.

12.4.3. Reproducibility

LazyOTF and its heuristics can optimize MBT for a high degree of determinism of specific TCs, i.e., to avoid uncontrollable nondeterminism as much as possible. This leads to higher **repeatability** of verdicts for these TCs, and thus higher **reproducibility** of specific failures.

Our heuristics help avoid uncontrollable nondeterminism in multiple ways:

- guidance heuristics reduce the randomization of on-the-fly MBT (cf. Subsec. 11.1.2);
- uncontrollable nondeterminism can be reduced by the phase heuristics choosing states with high uncontrollable nondeterminism as inducing states, lowering their weights by forbidding subtrees;
- since our guidance heuristics find shorter test cases, they usually contain less uncontrollable nondeterminism;
- *finingPath2W* can be used to further fine uncontrollable nondeterminism (cf. Subsec. 12.3.5).

Note. For fully deterministic test case generation, the weights of different WTCs should differ.

12.4.4. More Dynamic Information for Bound Heuristics

The bound heuristics in Subsec. 12.2.2 only checked in which sub-phases TOs were discharged, and did not depend on any further information. This was an intentional

design choice to keep the dynamic bound heuristics easily configurable and efficiently computable.

For strong guidance via TOs, a small bound is usually sufficient since the TOs guide in the right direction over multiple phases. For weak guidance via TOs, this is not the case, so a larger bound helps in finding those TOs and thus improves the meaningfulness (cf. Subsec. 14.3.6). But a larger bound increases test case complexity and the resources needed for test case generation (cf. Fig. 14.5). So the bound heuristics must continuously balance the trade-off between meaningfulness and performance for the set of currently active TOs. In specific cases, more information can help the bound heuristics to optimize this trade-off faster and better. For instance, in rare cases the following might happen over multiple phases if the guidance heuristcs are configured badly: $b_{p_{curr}} = 5$ and a TO is discharged every $p_+ - 1$ phases, so $b_{p_{curr}}$ is not increased; but for $b_{p_{curr}} = 6$, a TO would be discharged in every phase. This shows that in certain cases, extending b_+ and b_- to consider more dynamic information would yield better heuristics. An exemplary implementation is b_-, b_+: dynamicInfo $\mapsto \max(b_{min}, \min(b_{max}, \mathit{length}_{discharge}))$, where $\mathit{length}_{discharge}$ is the mean number of test steps between two discharges. Since the average distance between all active TOs can change over time, the *mean* could be taken over the last $m \in \mathbb{N}_{>0}$ discharges only. So b_-, b_+ reflects the current average distance, and $b_{p_{curr}}$ quickly adopts a suitable value; hence p_+, p_- can be set to 1. But since computing b_-, b_+ is expensive, higher values of p_+, p_- might lead to a better overall performance.

Since these heuristic and their computations are complex and might only improve LazyOTF in rare cases, this extension is future work.

12.4.5. Quantifying Nondeterminism

The testing hypothesis (cf. Subsec. 8.1.2) abstracts from all probabilities and uses unquantified choices in form of nondeterminism instead. If we had probability distributions instead, they could be integrated into our weights heuristics, resulting in more accurate **probabilistic weights**.

Example. A first step in this direction was the pragmatic approach of quantifying probability in $\underline{aggWTCs} = +(p2W, +(p \cdot z_0, (1-p) \cdot mean(\dots)))$ (cf. Lemma 12.34).

The probability distributions can be formalized by labeling transitions with probability values, as for discrete-time Markov chains (cf. Subsec. 5.5.4). The probability values can be user-supplied, derived from usage scenarios, or from observations during execution.

A simple solution counts the number of visits for each superstate and transition; the probability that transition $\ddot{s} \xrightarrow{l} \ddot{s}'$ is taken in $\ddot{s}$ can thus be approximated by $\mathbf{P}_{\boldsymbol{enabled(\ddot{s})}}[\boldsymbol{\ddot{s} \xrightarrow{l} \ddot{s}'}] := count(\ddot{s} \xrightarrow{l} \ddot{s})/count(\ddot{s})$. For this, $count(\cdot)$ is incremented by one at each visit, and initialized with $count(\ddot{s}) := enabled(\ddot{s})$ for each superstate $\ddot{s}$ and $count(l) := 1$ for each transition l, which yields equidistribution of $\mathbf{P}_{enabled(\ddot{s})}\left[\ddot{s} \xrightarrow{l} \ddot{s}'\right]$ at the beginning. Other distributions are also possible.

Then $\mathbf{P}_{enabled(\ddot{s})}\left[\ddot{s} \xrightarrow{l} \ddot{s}'\right]$ can be used by assembleWTC$(s, p2W, l2WTC)$ to compute $\underline{aggWTCs}(p2W, z_0, \dots, z_{|L_U|})$, e.g., by $+(p2W, z_o \cdot \mathbf{P}_{enabled(s)}\left[s \xrightarrow{i_\delta} s_0\right], \Sigma_{i=1}^{k} z_i \cdot \mathbf{P}_{enabled(s)}\left[s \xrightarrow{u_i} s_i\right], mean(z_{k+1}, \dots, z_{|L_U|}))$, as extension to $+(p2W, +(p \cdot z_0, (1-p) \cdot$

$mean(\dots)))$. Alternatively, $\mathbf{P}_{enabled(\ddot{s})}\left[\ddot{s} \xrightarrow{l} \ddot{s}'\right]$ can already be factored in as weight when computing *path2W* during traversalSubphase, e.g., via *finingPath2W*. A straight forward approach computes ***path2Wprob*** $(\pi^{full}) := path2W(\pi^{full}) \cdot \mathbf{P}_{paths(\mathcal{S},source(\pi^{full}))}\left[\pi^{full}\right]$, with $\mathbf{P}_{paths(\mathcal{S},source(\pi^{full}))}\left[\pi^{full}\right] = \prod_{i\in[1,\dots,1+|\pi|)} \mathbf{P}_{enabled(s_{i-1})}\left[s_{i-1} \xrightarrow{l_i} s_i\right]$ (cf. Subsec. 5.5.4) and $\pi^{full} = (s_{i-1} \xrightarrow{l_i} s_i)$.

Quantifying nondeterminism could be implemented by extending the JTorX partitioner (see Subsec. 10.3.3 and [de Vries et al., 2002; Belinfante, 2014]) to also cover output and quiescence. But this is future work since it is unclear how much improvement quantifying nondeterminism will bring in practice, whether it is misleading for abstract specifications, how to retain exhaustiveness, and how costly and stable the numerical computations are. Roughly similar approaches of counting to bias choices according to the number of traces, transitions, or states reachable via each successor are given in [Denise et al., 2008], but they count traces (or decomposable structures) to uniformly generate traces of a given length, and have to count a large amount in an offline manner.

12.5. Conclusion

12.5.1. Summary

This chapter has shown that exhaustiveness, coverage and discharging TOs is in general inherently difficult because our ioco theory is very general and allows infinite state space, uncontrollable nondeterminism, weak fairness constraints and underspec_U (cf. Note 12.11 and Note 12.18). Thus this chapter designed elaborate heuristics: phase heuristics on the one hand, abstract guidance heuristics on the other.

Our phase heuristics (cf. Sec. 12.2) break down the graph into appropriate sub-graphs to simplify the traversal and test case generation, to improve their performance, and to enable dynamic information. Sub-graphs are derived by inducing states, which are user-supplied, and dynamic bound heuristics, which vary the bound from phase to phase to adapt to the current situation.

Our abstract guidance heuristics (cf. Sec. 12.3) are kept flexible and offer the provisos $\text{P}_{\rightarrow OTF}$, $\text{P}_{vanishing}$, $\text{P}_{discharge}$, $\text{P}_{coverage}$ and $\text{P}_{coverViaTOs}$ to guarantee exhaustiveness, coverage, and discharging if the guidance heuristics comply with the corresponding provisos. Since our focus is on discharging TOs, the further provisos P_{phase}, P_{goal} and P_{fair} are provided to guarantee $\text{P}_{discharge}$. Furthermore, advice on how to achieve these provisos is given. Thereafter, weights heuristics are introduced by extending TCs, TSs and the algorithms genTC and genTS, resulting in the algorithms genWTC and genWTS. LazyOTF applies genWTS in each traversal sub-phase to generate a potentially most meaningful TC; LazyOTF is shown to be sound and, if the corresponding provisos are met, exhaustive or discharging TOs. Then these provisos are further investigated for our weights heuristics, giving criteria on how to implement P_{phase}, P_{goal} and P_{fair}. This chapter has also shown how meaningfulness is configured by defining and composing TOs. We have seen that TOs can efficiently describe user-supplied information and are useful in practice, e.g., for specific requirements and iterative software development. Hence our provisos are further investigated for TOs, restricting our criteria such that they are retained by

composition. Finally, we introduced cycle warnings and countermeasures in case our provisos are unmet. In summary, our provisos and their relationship are depicted as **provisos framework** in Fig. 12.4.

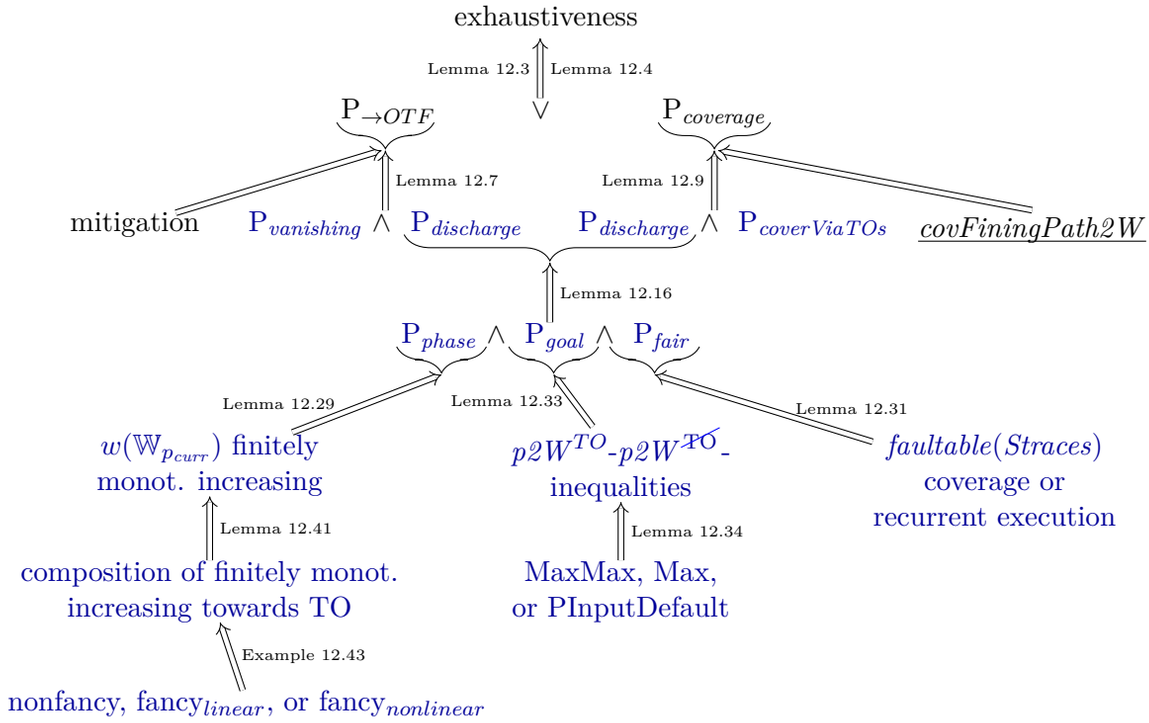


Figure 12.4.: Guarantees for exhaustiveness and TO discharge of LazyOTF by a framework of (TO-based and other) provisos

Our guidance heuristics subsume all classical test selection heuristics and are more general and flexible:

- compared to randomness, our guidance heuristics are more directed;
- compared to test purposes, our guidance heuristics are easier to specify, better composable, and more expressive. But test purposes are a suitable and simpler solution if a specific purpose needs to be tested (e.g., a single path) and uncontrollable nondeterminism does not impede this purpose;
- compared to coverage, our guidance heuristics have finer granularity, can cover arbitrary artifacts and cope with infinitely many of them, can handle and measure uncontrollable nondeterminism, and are more expressive;
- compared to the cycling heuristics, our guidance heuristics are easier to specify, have finer granularity, and are more expressive.

In summary, our guidance heuristics, compositional TOs and weights lead to better guidance and more meaningful test cases. Our provisos framework offers flexibility, which is important since MBT tests a wide variety of SUTs and properties, in very different domains, so different guarantees and performance criteria are needed. Unfortunately, our guidance heuristics are quite elaborate and are hence difficult to transfer to other test case generation algorithms that might be interesting to employ for LazyOTF (cf. Subsec. 11.2.1, Sec. 13.4). But here, our provisos framework can also help.

12.5.2. Contributions

The main contributions in this chapter are:

- phase heuristics for LazyOTF's phases: inducing states for user-supplied heuristics with high usability, and a dynamic bound heuristic to determine $b_{p_{curr}}$ on the basis of dynamic information;
- abstract guidance heuristics with a provisos framework, which enables flexible configurations to guarantee exhaustiveness, coverage, and discharging, and shows the cost of those guarantees;
- flexible weights heuristics and the investigation and extension of our provisos framework for them, showing that strong guarantees and efficiency are often conflicting;
- TOs for configuration of our heuristics, the investigation and extension of our provisos framework for them, and composition of TOs to simpler meet the provisos and unite the guarantees with efficiency;
- further advice, criteria and examples on how to achieve the provisos;
- optimizations for our heuristics.

12.5.3. Future

Possible future work includes:

- implement bound heuristics that employ a higher amount of dynamic information than simply in which sub-phases TOs were discharged. These are future work since they are complex and might improve LazyOTF only rarely (cf. Subsec. 12.4.4);
- quantifying nondeterminism is future work since it is unclear how much improvement it will bring in practice, and it might break other desired aspects of LazyOTF (cf. Subsec. 12.4.5);
- let LazyOTF decide how to behave when a failure occurs (cf. Subsec. 8.9.3), instead of always simply restarting or terminating;
- many weight computations in Subsection 12.3.5 not based on TOs have not yet been implemented, but are future work since our focus was on TO-based heuristics;
- likewise, mitigations that are not based on TOs were not in our focus, have not been implemented yet and are future work;
- for real-time computing, investigate greedy approaches (cf. Note 12.17) to satisfy the provisos;
- implement on-the-fly variants of the cycling heuristic: via counting cycles within genWTS, via *cycleFiningPath2W*, or via NPC checks.

13. Implementation of Lazy On-the-fly MBT

13.1. Introduction

LazyOTF is implemented in Java and has about 28 000 LoC. It is an extension to STSimulator (cf. Subsec. 13.2.1), an API for STSs that expands STSs to LTSs. Therefore the LazyOTF implementation operates on the level of LTSs. JTorX (cf. Subsec. 10.3.3) is used as GUI for configuration, interaction and visualization, but also for functionality that LazyOTF can better delegate JTorX: the exploration of the specification, its determinization, and test execution. To connect LazyOTF to JTorX, LazyOTF is embedded in the adapter between JTorX and STSimulator, called **SymToSim**. Furthermore, extensions and a few modifications to JTorX were necessary.

Roadmap. The implementation of LazyOTF's core test case generation based on STSimulator is described in Sec. 13.2. The integration and implementations in JTorX are described in Sec. 13.3. An alternative implementation on the symbolic level is depicted as proof of concept in Sec. 13.4. Finally, Sec. 13.5 covers several technical optimizations in test case execution and test case generation.

13.2. Core LazyOTF

The core test case generation algorithm is described in Chapter 11, the applied phase heuristics and guidance heuristics via weights in Chapter 12. Therefore, this section only depicts deviations and extensions. Further technical details are described in the **LazyOTF manual** [Kutzner and Faragó, 2015].

13.2.1. STSimulator

STSimulator [URL:STSimulator; Frantzen, 2007; Larysch, 2012; Kutzner, 2014; Frantzen, 2016] is a Java library to model and simulate STSs (cf. Def. 3.30). Since FOL is undecidable [Church, 1936; Turing, 1936; Kleene, 1967] and unfamiliar to most test engineers, STSimulator's guards and updates use the **Dumont language** [Frantzen, 2007; Larysch, 2012; Kutzner, 2014] instead, which reduces the expressiveness for decidability and adopts Java's syntax. It is typed and supports bounded non-negative integers, $\mathbb{B}$, user-supplied strings, enums and records. For expressions of type integer and $\mathbb{B}$, Dumont offers the standard relational and arithmetic operators, otherwise only equality and inequality (so no quantifiers). The experimental support for fixed-point decimals, calendar dates, lists and external calls of Java code is strongly restricted, since they cannot be used in expressions due to the applied solver. To solve guards (in combination with

updates), STSimulator uses **treeSolver** as constraint solver, which is a simple solver based on Prolog.

To simulate an STS $\mathcal{S}$, STSimulator keeps track of the current superstate in $[[\mathcal{S}_{\mathcal{V}^d}]]_{det}$ and allows checking and giving inputs in L_I, outputs in L_U, and δ. Thus STSimulator simulates an STS on the level of LTSs (cf. Sec. 9.4). If $\mathcal{S}$ has a switch $\xi := s \xrightarrow{l<type(l)>,[F],\rho} s'$ with an interaction variable $v \in type(l)$ of some infeasibly large (e.g., unbounded) type, ξ might be expanded into infeasibly (e.g., infinitely) many transitions in $[[\mathcal{S}_{\mathcal{V}^d}]]$: If the guard F only allows few transitions to states other than fail, the traversal and generated TC remain feasible for an efficient implementation (cf. next subsection). Otherwise, the implementation remains feasible by applying reduction heuristics (cf. Subsec. 12.3.1): v's value space for ξ is reduced to a feasible size by inspecting $\mathcal{S}$ or by only taking values specified by the test engineer.

Several minor bug fixes in STSimulator were made, and changes to improve its robustness, its performance (e.g., for τ-closures, cf. Def. 8.7 and Subsec. 13.5.2) and its functionality (via exit criteria, STS editor, STS Guides, alternative simulation). A few extensions to employ STSimulator for LazyOTF were also necessary.

13.2.2. Test Cases

In practice, adding verdict leafs in test cases (and transitions to the verdict leafs) can become inefficient for large $|L_U|$. Instead, we do not demand leafs to be pass or fail (cf. Def. 8.49, Chapter 11): If an output transition is missing in the test case, it is an **implicit output transition to fail**. For OTF MBT, pass only occurs when test execution terminates (by user interaction or the SUT terminating), so it costs little. It can still be omitted if pass is indicated by termination without fail.

13.2.3. Heuristics

Bound Heuristics

Bound heuristics are implemented as described in Sec. 12.2, but not in its full generality: For higher usability, there are pre-built bound heuristic templates, so that some of the parameters given in Def. 12.1 are already set in an appropriate way;

- the so called **sawtooth phase heuristic** [Kutzner, 2014] has aggressive reduction and is used when TGs of different TOs are clustered: $b_+ : b \mapsto min(b_{max}, b+3)$, $b_- : b \mapsto b_{min}$, $p_- = 1$. p_+, b_{min}, and b_{max} are still configurable;
- the so called **triangle phase heuristic** has a conservative reduction that only reduces $b_{p_{curr}}$ when there is multiple evidence that TOs are reachable within the bound: $b_+ : b \mapsto min(b_{max}, b+1)$, $b_- : b \mapsto max(b_{min}, b-1)$. $p_+ = p_-$, b_{min}, and b_{max} are still configurable.

Guidance Heuristics

The application of our guidance heuristics via weights is described in Sec. 12.3. As codomain of the weight function w, we mainly use the finite set $[0, \ldots, 2^{32}-1]$. For higher flexibility and usability of a TO o, its $I_o^{lazy}(\cdot)$ and $\underline{path2W}_o(\cdot)$ are refined in the implementation; the artifacts for applying the guidance heuristics are introduced

here, whereas the configuration and management of TOs (e.g., for coverage) is covered in Sec. 13.3. But for total freedom, $\underline{path2W_o}(\cdot)$, $I_o^{lazy}(\cdot)$, $discharge_o(\cdot)$, and their helper functions can be implemented from scratch by implementing, resp. extending, the corresponding interface, resp. class (e.g., via sub-TOs and observers, cf. o_{dec} in Subsec. 14.3.3).

The implementation generalizes $I_o^{lazy}(\cdot)$ (cf. Def.11.5): Besides the elements in Σ_{lazy}, the verification engineer can use the element **DECIDED_BY_STRATEGY** when marking states for $I_o^{lazy}(\cdot)$, to combine programmatic and enumerative $I^{lazy}(\cdot)$ implementations. He must then also provide a **decision strategy** $S \rightarrow \Sigma_{lazy}$ (or $S_{det} \rightarrow \Sigma_{lazy}$, cf. Subsec. 11.3.1) within the test objective o. The strategy is used dynamically during traversalSubphase to resolve the type of a state marked as DECIDED_BY_STRATEGY to some type in Σ_{lazy}. If no decision strategy is given in o, in spite of DECIDED_BY_STRATEGY markings, then implicitly the **default decision strategy** that maps all DECIDED _BY_STRATEGY to ORDINARY is used. An employed decision strategy can be a new implementation of the interface **Location. LocationTestType.DecisionStrategy**. Alternatively, a pre-built strategy can be configured, e.g., a **MappingStrategy** that defines a map from (abstract) states to Σ_{lazy}, or a **SatStrategy** that defines a map from concrete states to Σ_{lazy} with the help of Dumont expressions (cf. Subsec. 13.2.1). With this approach, $I_o^{lazy}(\cdot)$ can be declared flexibly and concisely.

For high flexibility and usability, the computation of $\underline{path2W_o}(\cdot)$ is distributed over two layers (similarly to distributing the computation of WTCs' weight between $\underline{path2W_o}(\cdot)$ and *aggWTCs*, cf. Subsec. 12.3.4): a layer for states and superstates, and another layer that aggregates them for paths. Analogously, *aggPath2Ws* that deviates from the default *max* can also be implemented over the two layers.

Weights can be assigned to states and superstates with a new implementation of the interface **NodeValue2Weight<N>**, with the formal type parameter N being the node value type, for flexible weights. Alternatively, a pre-built NodeValue2Weight<N> can be configured: **LocationValuation2Weight** maps concrete states to weights, configured via Dumont expressions. **Locations2Weight** maps abstract states to weights. For simplicity, JTorX embeds Locations2Weight as **TestTypeDefaultNode2Weight** to assign a weight to a superstate $\ddot{s}$ according to $I^{lazy}(\ddot{s})$. For superstates, weights of its states can be aggregated similar to aggregation over TOs and TCs, e.g., via *max*, *sum*, or *mean*. While *max* is canonical to deriving I^{lazy} on superstates, and to aggregating TOs, the alternative *mean* and *sum* are more balanced and are fining nondeterminism (cf. Subsec. 14.3.3). Nondeterminism can also be considered (fined and rewarded) by the prebuilt **NondeterminismFiningNodeValue2Weight** and **SuperLocationSizeNodeValue2–Weight**. *aggPath2Ws* that deviate from the default *max* and the alternative *sum* can be implemented on the layer of states by implementing NodeValue2Weight<N> that has other nested NodeValue2Weight<N>, e.g., as for **SumOfOtherNodeValue2Weight<N>**.

The weight for a path can be computed by implementing the interface **Path2Weight <N, E>** (with parameter E being the edge value type). Alternatively, various prebuilt **Path2Weight<N, E>** can be configured: to aggregate the weights of all abstract superstates on a path (e.g., addition via **NodeValueSummingPath2W**), or to only consider the final one (**FinalLocationSetDefaultPath2W**, see also Note 12.20), or concrete superstates (**CollapsedLocationsSetPath2W**), or to take the length of the path into account (**LengthPath2W**), or its transitions (**MessageSetPath2W**), or to fine

nondeterminism (**NondetFiningFinalLocationSetPath2W**). If a Path2Weight<N, E>, P, only uses one NodeValue2Weight<N>, N, we shortly write $P(N)$. If the defaults TestTypeDefaultNode2Weight and FinalLocationSetDefaultPath2W are used, the user only has to configure $I_o^{lazy}(\cdot)$. *aggPath2Ws* that deviate from the default max and the alternative sum can also be implemented on the layer of paths by implementing the interface Path2Weight<N, E> that has other nested Path2Weight<N, E>, which has been done for AggSumOfOtherPath2W<N, E extends stsimulator.sts.InputDecider>.

If the original specification is an STS, all these heuristics have access to the artifacts on the STS level, e.g., to location variables. Furthermore, all heuristics can be configured and managed in the context of TOs, within the JTorX integration, described in the next section.

Our main *aggWTCs* $max(p2W, max(\dots))$, $+(p2W, max(\dots))$, $+(p2W, +(p \cdot z_0, (1-p) \cdot mean(\dots)))$ (cf. Subsec. 12.3.4) are pre-built, others can easily be implemented by extending the abstract class AbstractEdgeWeightAggregator, which offers multiple helper functions and hooks to implement an own *aggWTCs*, or by implementing the interface EdgeWeightAggregator from scratch. We set PInputDefault with $p = 0.5$ as default since it is the most meaningful in practice (cf. Subsec. 14.3.5).

13.2.4. Quality Assurance

The core LazyOTF contains the abstract $2^{(\mathcal{TTS}(L_I,L_U,\delta)\times\mathbb{V})}$ LazyOTF($\mathcal{LTS}(L_I, L_U, \tau)$ $\mathcal{S}$, *TOs* ö, *SUT* $\mathbb{S}$) and the test case generation algorithm genWTS, and is hence the most critical and formal part of LazyOTF. To assure its quality and correctness, we did not take our own medicine of applying LazyOTF on LazyOTF's code since genWTS behaves contrary to embedded and reactive systems: it takes one large input and then performs complex algorithms and constructs large data structures to then finally give one large output and terminate. Hence genWTS is not suitable for MBT; self application is an interesting case study as future work, but case studies on more suitable (reactive or embedded) software are more important (cf. Sec. 14.3).

We chose other measures to assure its quality and correctness: 441 unit and integration tests, 44 large parametrized system tests in the form of experiments (cf. Appendix B.1.5), and 16 large acceptance tests achieve a coverage level of 73% for instructions and 66% for branches. These values would be much higher if the coverage measurements with EclEmma [URL:EclEHP] could also detect covered exceptional code as such [URL:EclFAQHP], since our code contains a lot of exception handling for fault tolerance.

Furthermore, pluggable type checking via the Checker Framework [Papi et al., 2008; Ernst et al., 2011] enhances Java's type system (cf. Subsec. 1.1.3), to statically check immutability and nullness, including pure and covariant types.

To accumulate as much information as possible for debugging and for our experiments, we implemented and usually activated thorough **logging**:

- **storing communication between JTorX and LazyOTF** in memory, which can be presented or logged via observers, for debugging purposes (cf. LazyOTF debug console in Subsec. 13.3.5);
- **verbose logging** about test generation, test execution, and their performance is written on disk and stored as experiment archive;

- **low-level logging** on various verbosity levels (with SLF4J) enables further debugging.

13.3. JTorX Integration

All aspects of LazyOTF that are related to user interaction, visualization and features already offered by JTorX (cf. Subsec. 10.3.3), e.g., test execution, are embedded in JTorX: Subsec. 13.3.1 describes how JTorX handles the management of TOs. Subsec. 13.3.2 sketches how JTorX's GUI helps in configuring LazyOTF. Subsec. 13.3.3 states how and to what extent coverage criteria are integrated in JTorX's GUI. Subsec. 13.3.4 sketches how JTorX's test execution is connected to LazyOTF. Subsec. 13.3.5 describes the GUI for feedback and interaction during test execution.

13.3.1. Test Objective Management

TOs are a central feature of LazyOTF, as they configure the phase and guidance heuristics. The computations of $\underline{path2W}_o(\cdot)$, $I_o^{lazy}(\cdot)$ (and thus decision strategies) are performed if and only if they are contained in an active TO o, as defined in Def. 12.36, because each of those computations should help test some requirement or feature, represented by a TO. Being such a central feature, TO management should be usable and efficiently: The configurations for LazyOTF can be structured clearly by TOs, and TOs should be discharged efficiently when updating dynamic information from test execution. For extensibility and performance, $I_o^{lazy}(\cdot)$ is implemented by a separation of mechanism and policy [Wulf et al., 1974]. Therefore, discharging TOs is optimized for the default $discharge_o(\cdot)$ (cf. Def. 12.36). But individual discharge functions are also facilitated by flexible observer patterns [Gamma et al., 1995] implementing the interface AbstractDriverObserver.

Since both configuring LazyOTF (cf. Subsec. 13.3.2) and updating dynamic information from test execution (cf. Subsec. 13.3.4) are embedded in JTorX, so is TO management.

13.3.2. Configuration

In JTorX's GUI, a checkbox enables guided testing via LazyOTF and a new tab for configurations, implemented by Felix Kutzner and depicted in Fig. 13.1. It offers the direct configuration of simple settings and gives access for more elaborate configurations.

The simple configurations the main tab offers are:

- for the bound heuristics, values for b_{min}, b_{max}, and $b_- = b_+$ can be set;
- for guidance heuristics, $\underline{aggWTCs}$ can be configured, as well as TOs and weight computations for the simplest configurations of guidance heuristic. This is achieved via **standard configurations** or via $I_o^{lazy}(\cdot)$, using the defaults (cf. Subsec. 13.2.3) TestTypeDefaultNode2Weight and FinalLocationSetDefaultPath2W;
- storing and loading all of LazyOTF's configuration via LazyOTF's XML configuration files.

Infrequent configurations for debugging, logging and exit criteria are hidden in a submenu.

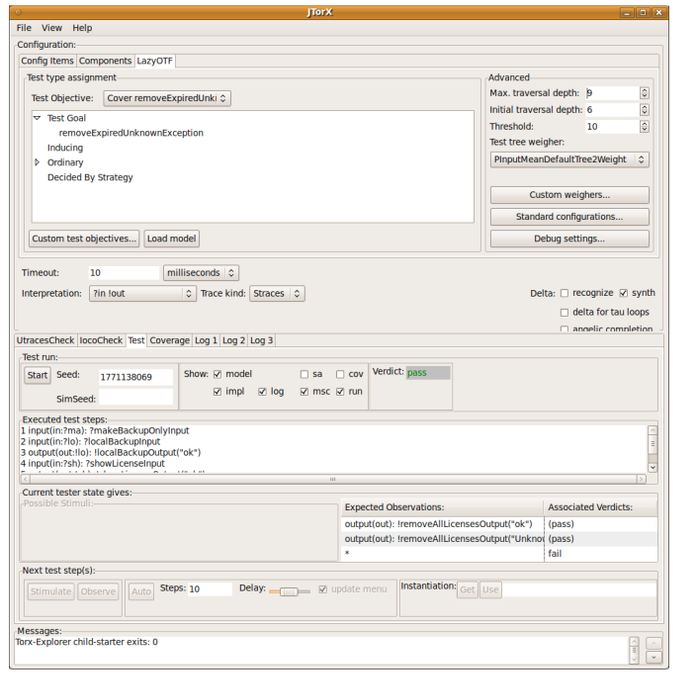


Figure 13.1.: LazyOTF's main tab

For more elaborate configurations of the guidance heuristics (cf. Subsec. 13.2.3), the LazyOTF GUI offers a multi-level **TO editor** and **weights editor** to configure an arbitrary amount of:

- sets of states for configuration purposes;
- Dumont expressions used for guidance heuristics;
- decision strategies;
- $I_o^{lazy}(\cdot)$;
- NodeValue2Weight<N>;
- Path2Weight<N, E>;
- TOs.

Although artifacts are only used by the guidance heuristics if they are contained in an active TO, the TO editor also allows to define these artifacts independently, for higher usability (e.g., loading and storing elaborate configurations). For even higher flexibility, computations can be implemented programmatically by implementing the corresponding interface, as described in Subsec. 13.2.3.

13.3.3. Coverage Criteria

Besides the default settings for the heuristics functions (cf. previous subsection), general coverage criteria (cf. Sec. 2.5) are important settings. Currently, only abstract state coverage on the STS level (cf. Sec. 9.4) is offered as proof of concept for a **general coverage criterion**, i.e., a coverage criterion implemented by automatically generating a suitable set of TOs from the given specification. Offering further general coverage criteria is future work and facilitated by the flexible implementation (cf. Subsec. 13.3.1).

Currently, our applied heuristics (cf. Sec. 14.3) mainly use defaults and some settings specific to the given specification, including further coverage criteria. These specific settings have been configured manually (cf. previous subsection) or programmatically and stored for automatic application.

Alternatively, coverage criteria can be implemented via *covFiningPath2W*$((\pi_i^{full})_i)$. This is also future work since this thesis focuses on core guidance heuristics (cf. Sec. 14.3). Hence the coverage criteria for nondeterminism on output as defined in Def. 12.27 were also not yet implemented.

Note. In Sec. 14.3, we achieved 100% S_{isol} (i.e., coverage of $faultable(S)$ in isolation, which incorporate nondeterminism of the LTS, cf. Subsec. 8.8.4) and 100% abstract state coverage on the STS. which implies 100% abstract 1-choice coverage.

13.3.4. Test Execution

Our implementation of testExecutionSubphase updates dynamicInfo immediately (cf. Subsec. 11.2.3). This is not necessary for exitCriterion and traversalSubphase, but for the GUI that visualizes dynamicInfo (cf. Subsec. 13.3.5) and for distributed LazyOTF (cf. Subsec. 11.5.2).

Efficient and flexible updates are made possible by an open and clean architecture (cf. Subsec. 10.3.3, Sec. 11.3 and Subsec. 13.3.1). Updating dynamicInfo immediately and having flexible observers, multiple exitCriterion can be integrated efficiently in the updating mechanism.

13.3.5. Dynamic Feedback and Interaction

The **LazyOTF introspection window** (cf. [Kutzner and Faragó, 2015]) offers feedback to the test engineer during test execution, as well as interaction. Its main tab is the **trace/tree explorer**, shown in Fig. 13.2. It gives feedback to the test engineer about the dynamic information during test execution by showing the path sequence of all previous test execution sub-phases (cf. Def. 11.8); directly below, the current WTC is displayed. So the trace/tree explorer visualizes the results of the last traversalSubphase and all testExecutionSubphases (cf. Listing 11.1). Furthermore, flags indicate additional information about the source superstate $\ddot{s}$ of each transition:

- flag **T** indicates that $\ddot{s}$ discharged a TO;
- otherwise, flag **I** indicates that $\ddot{s}$ is inducing;
- flag **F** in the path sequence indicates that the transition was the first one in the corresponding WTC, i.e., traversalSubphase was executed in $\ddot{s}$;
- flag **C** indicates that $\ddot{s}$ caused a cycle warning (cf. Subsec. 12.3.7).

Interaction is also possible in the trace/tree explorer: through context menus, the test engineer can choose to

- stimulate with LazyOTF-proposed input: this executes the first input in the current WTC (the green transition), i.e., a most meaningful input;
- observe: this observes an output or δ;
- recompute test tree, which executes no test step but traversalSubphase;
- abandon test tree, which switches to JTorX's pure on-the-fly mode.

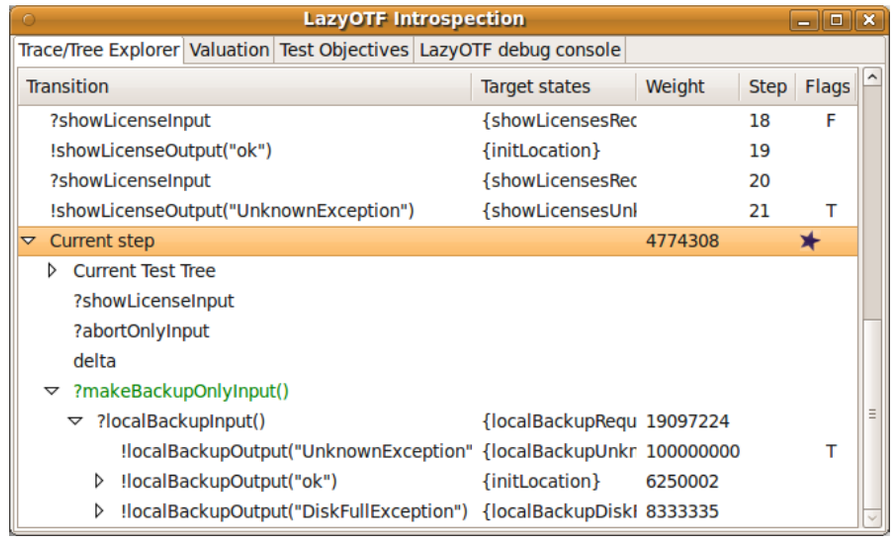


Figure 13.2.: LazyOTF's introspection window

Details of the data on the STS level is given in the **valuation tab**, on the discharged TOs in the **test objectives tab**. JTorX and the **LazyOTF debug console** give access to all dynamic information by allowing interaction with the LazyOTF core (cf. Sec. 13.2). The help command of the debug console describes its available commands:

- get: gets information about the current superstate $\ddot{s}$ and their states $s \in \ddot{s}$ (including I^{lazy} and decision strategies), the current node n in $\mathbb{W}_{p_{curr}}$ (including its weight $w(n)$), and further technical details (e.g., visitors for the STS traversal method);
- propose: get $i_\delta \in L_I \dot{\cup} \{\delta\}$ for the current node in $\mathbb{W}_{p_{curr}}$;
- took−transition: inform LazyOTF about an interaction with the SUT, i.e., that test execution has taken the given transition step in the test run path. The extended version took−transition−ext offers convenience features;
- set−location−type: set I^{lazy} for some state s;
- strategy: manages the decision strategies for $DECIDED\_BY\_STRATEGY$ test types;
- set−max−recursion−depth: specify the next traversalSubphase's depth bound $b_{p_{curr}+1}$;
- set−tt2w−aggregate: set $aggPath2Ws$ (currently max and sum is implemented);
- set−test−tree−weigher: set $aggWTCs$;
- path2weight: manage $path2W_o(\cdot)$ (including Path2Weight);
- nodevaluetoweight: manage NodeValue2Weight;
- compute−**new**−test−tree: compute a new $\mathbb{W}_{p_{curr}}$ (or a WTC for any other visited superstate);
- select −visitor: select a visitor for the STS traversal method in genWTS (for deviation from the default DFS);
- discharge−test−objectives: information about an observed $discharge_o(\pi^{full})$ from a programmatic TO o;
- oraclesafetytree : manages the eager micro-traversal sub-phase optimization (cf. Subsec. 12.4.2);
- get−cycle−warning: get a boolean value that is **true** if the current node has a cycle

warning;
- set−cycle−warning−threshold: set the cycle warning threshold c (cf. Subsec. 12.3.7);
- torx−get: interface to TorX interna;
- debug: low level debugging for various purposes;
- help: prints out this list of available commands.

Since user interaction is optional, JTorX also offers a **text mode** for fully automatic MBT without any GUI. In text mode, LazyOTF is configured via LazyOTF's XML configuration files, and all dynamic information is stored in the specified log file. This functionality is used heavily in the case study presented in Sec. 14.3.

Note. For TOs that describe requirements (or other specification or design artifacts), the test engineer can use the feedback of dynamic information for traceability between TCs and those artifacts (cf. Subsec. 11.1.2): For a simple form of traceability, the test engineer only has to look for **T** flags to find out where TOs have been discharged. If it is not obvious which TOs have been discharged, the test engineer must use the debug console to find out. Acquiring this information could easily be automated, e.g., implemented in an own traceability tab, which is future work. Our guidance heuristics can provide information that goes beyond classical traceability: Besides reporting TOs that have been discharged in the current TC $\mathbb{T}^{full}_{r_{curr}}$, a TO o that $\mathbb{T}^{full}_{r_{curr}}$ has almost discharged can be reported (for instance all but the final decrease for o_{dec} has been performed, cf. Subsec. 14.3.3). If o corresponds to some feature, aspect or behavior that can cause a failure before having fully been executed, o is helpful traceability information, too. To determine whether o has almost been discharged in the current test step t_{curr}, its $path2W_o$ difference since the last reset or since the last discharge can be computed. In each state, all TOs o whose $\underline{path2W_o}$ difference has just increased beyond some threshold can be reported, or simply the $\underline{path2W_o}$ difference for all active TOs o (cf. Def. 12.39). Such **fuzzy traceability** is interesting future work, and there is little related work: [Gaur and Soni, 2013] uses fuzzy sets to model vague requirements and to trace them up to a desired degree of relevance; [Noppen et al., 2008; Turban, 2011] consider fuzzy traceability between requirements and design artifacts. All this related work is motivated by partial knowledge of some requirement or other artifact, as opposed to our motivation by partial coverage of a requirement. A related work that also investigates coverage metrics for requirements-based testing and its traceability is [Banka and Kolla, 2015]. The property relevance investigated in [Fraser and Wotawa, 2006] helps determine the relationship between TCs and properties, and hence also traceability if the properties are requirements. But the determined relationship "is too weak since it only checks if the property is relevant to the TC. It does not determine whether the property is exercised by TCs in several interesting ways" [Banka and Kolla, 2015], which LazyOTF can do since arbitrary many TOs can be defined for a requirement.

13.4. Alternative Implementation with Symbolic Execution

The LazyOTF implementation described in the previous sections is based on STSimulator and therefore has the following deficits:
- it operates on the level of LTSs: So for $\mathcal{S} \in \mathbb{S}_{STS}$, the structural information of $\mathcal{S}$ is lost and state space explosion becomes more severe due to expansion (cf.

Sec. 9.4). Furthermore, the expansion needs to be handled technically (for which JTorX interfaces with STSimulator using parameterized transition systems [Jéron et al., 2013; Belinfante, 2014]);
- the language Dumont for STS guards and updates is used, which is a very simple constraint language (cf. Subsec. 13.2.1).

13.4.1. STSExplorer

This subsection depicts **STSExplorer**, which is a proof of concept implemented by Florian Larysch [Larysch, 2012] for replacing STSimulator to avoid the deficits from above:
- it additionally offers symbolic execution, i.e., execution of transitions of the STS on symbolic instead of concrete values by constructing constraints on the symbolic values along paths, resulting in symbolic paths (cf. Subsec. 7.3.2). Therefore, data embedded in STSs can be handled more efficiently, the possible values need not be expanded, allowing unbounded data types without the need for reduction heuristics;
- it supports a more expressive language for guards and updates, baptized **Zuul**, an extension of Dumont that offers quantifiers, integer and real data types, container data types with user-specified functions, and external calls, e.g., of Java code.

STSExplorer is a TorX Explorer (cf. Sec. 10.3) as a drop-in replacement for SymToSim (cf. Sec. 13.3). It offers simulation as STSimulator (the operations take, receive, jump), and additionally symbolic exploration (the operations explore, backtrack, jump).

For symbolic execution, constraints constructed from guards and updates must be managed and solved. For constructing and managing constraints, STSExplorer uses SMT-LIB 2.0, for solving the constraints the SMT solver Z3 (cf. Subsec. 3.3.3). The implementation can handle container data types, incremental solving, external calls, quantifiers, and partly extremal and randomized solutions with the help of quantifiers [Larysch, 2012]. Therefore, STSExplorer can be used for the core LazyOTF test case generation algorithm. LazyOTF's heuristics (cf. Chapter 12) can either be handled outside of STSExplorer, as in Sec. 13.2, or encoded into SMT-LIB, which can also handle various coverage criteria like interaction coverage [Grieskamp et al., 2009] and data flow coverage [Su et al., 2015a].

Three problems were detected during implementation:
- technical difficulties necessitate work-arounds to embed symbolic execution in JTorX due to its protocols between components (cf. Subsec. 10.3.3);
- the average runtime of MBT with STSExplorer is six times as high as the average runtime with STSimulator. Investigations showed that handling nondeterminism of the LTS manually in STSExplorer is a main cause, since the nondeterminism is resolved by iterating over all states of a superstate (via jump). This does not fit into Z3's incremental solving and hence results in poor performance of the solver. For higher efficiency and flexibility, nondeterminism of the LTS should be embedded in SMT by logically encoding superstates. Furthermore, many time consuming calls to the SMT solver could be avoided by employing compiler optimization heuristics similar to Subsec. 7.1.3. Both improvements are future work;
- using quantifiers in SMT can cause undecidability for some theories (e.g., uninter-

preted functions and arrays, cf. Subsec. 3.3.3), but some heuristics are difficult to compute within SMT without quantifiers.

Therefore, a reimplementation of JTorX in combination with STSExplorer, considering these problems from the beginning, is the best approach and future work. Due to its workarounds and low performance, STSExplorer is only a proof of concept that SMT solving can be used to implement on-the-fly MBT and LazyOTF on the level of STSs with an expressive constraint language.

13.4.2. Related Work

There are several other testing tools that use constraint solvers. The most related ones are other advanced MBT tools that operate on specifications containing variables:

- JTorX operating on STSs using treeSolver (cf. Subsec. 13.2.1) can be considered related work (and hence also the main implementation of LazyOTF), but Dumont is not very expressive and treeSolver not very powerful; for most tasks, JTorX expands STSs into LTS;
- a variant of TGV called STG (cf. Subsec. 10.3.4) operates on variants of STSs using the Omega constraint solver [URL:omegaprojectHP], which has its focus on program analysis;
- Spec Explorer [URL:Spec Explorer; Veanes et al., 2008], which uses and strongly influenced the SMT solver Z3.

All three MBT tools are described in Subsec. 10.3.4, where their advantages and disadvantages are investigated.

LazyOTF with symbolic execution also has similarities to dynamic symbolic execution tools like EXE, KLEE, DART and CUTE (cf. Subsec. 7.3.2): They all intertwine execution with symbolic processing, including the use of constraint solving. When symbolic processing becomes infeasible, they all fall back on concrete values. KLEE also has optimizing heuristics via weights, but uses weights differently (cf. Subsec. 12.3.9) and is a white-box testing tool. Using the static testing approach for SBMC (cf. Chapter 7) is white-box static testing and hence further away from LazyOTF with symbolic execution. But static testing with SBMC can also be used for test case generation (cf. Note 7.1), which again has stronger similarities to LazyOTF with symbolic execution.

Instead of the core idea of executing instrumented code to detect runtime errors, LazyOTF's core is the ioco theory to check conformance. Since the specifications are usually not executable, their traversal is implemented in LazyOTF from scratch with elaborate algorithms (e.g., in Subsec. 8.8.3), lazy scheduling (cf. Sec. 3.6 and Chapter 11), and heuristics (cf. Chapter 12) that can additionally handle uncontrollable nondeterminism and sophisticated guidance.

13.5. Optimizations

The two main tasks of MBT are test case generation and test case execution, which LazyOTF improves compared to on-the-fly and offline MBT. The implementation of LazyOTF further optimizes test execution and test case generation.

13.5.1. Test Case Execution

Since the SUT is fixed, test steps cannot be accelerated, but only reduced and distributed. Thus LazyOTF speeds up test case execution in two ways:

Firstly, the guidance heuristics dynamically generate meaningful test cases, reducing the number of test steps that need to be executed. The implementation is described in Subsec. 13.2.3. Meaningfulness is further improved by our eager micro-traversal sub-phase optimization (cf. Subsec. 12.4.2), which is simply implemented via genWTS.

Secondly, distributed LazyOTF has (super-)linear speedup of meaningful test execution (cf. Subsec. 14.3.7). Integrating distribution into LazyOTF required only a few classes and changes, due to our powerful heuristics and LazyOTF's open architecture (cf. Subsec. 13.3.4). To reduce communication overhead and contention, and inform other instances efficiently about a discharge, efficient implementations of asynchronous message passing have been integrated in LazyOTF: by UDP broadcasts, UDP multicasts, and the more heavyweight Hazelcast; the implementation is described in Subsec. 11.5.2.

13.5.2. Test Case Generation

For test case generation, graph traversal is the bottleneck (cf. Sec. 14.3). LazyOTF speeds up graph traversal in two ways:

Firstly, by using phase heuristics and leveraging dynamic information (which also reduces memory requirements). The implementation is described in Subsec. 13.2.3. An optimization to the phase heuristics are the lazy traversal sub-phases described in Subsec. 12.4.1, which is simply implemented by JTorX's OTF.

Secondly, distributed LazyOTF parallelizes graph traversal with almost linear speedup.

There are, however, further possibilities to optimize graph traversal algorithms on the technical level:

One possibility is optimizing τ-closures by avoiding redundant work if a state is revisited during τ traversals (similar to TGV's optimization [Jard and Jéron, 2005]). This might improve performance in some cases where there is a high degree of nondeterminism of the LTS (cf. Subsec. 14.3.9). Since τ-closures are performed by JTorX using STSimulator, we had to adapt STSimulator to implement this optimization. It is activated by the JTorX JVM Parameter −Dstsimulator.enableTauClosureOptimization=**true**.

Another possibility is using caching of outgoing transitions in $L_I \dot{\cup} \{\delta\}$ in each state (or in superstates), so that the transitions to the heaviest successors (called $\boldsymbol{i_\delta}$ **transitions**, cf. Listing 12.2) need not be computed each visit. This $\boldsymbol{i_\delta}$ **caching** seems promising since genWTS can hereby reduce much work, as well as the randomization amongst heaviest i_δ transitions. i_δ caching is, however, a full heuristic that might cause inefficiencies or even inexhaustiveness in some cases. Therefore, it has not yet been implemented, so future work needs to implement and investigate it.

Many implementations of i_δ caching are possible; a simple version creates for each superstate $\ddot{s}$ a cache $\boldsymbol{c_{\ddot{s}}}$. Then in each superstate $\ddot{s}$, genWTS returns a random $i_\delta \in c_{\ddot{s}}$, after filling $c_{\ddot{s}}$ with the heaviest $i_\delta \in in_{\mathcal{S}_{det}}(\ddot{s}) \dot{\cup} \{\delta\}$ whenever $c_{\ddot{s}}$ is empty. All caches together are implemented by **cacheMap**, of type $2^S \to L_I \dot{\cup} \{\delta\}$ and initialized to map all superstates to $\varnothing$. This results in Listing 13.1, replacing l.13 – l.21 of genWTS from Listing 12.2. During execution of $\mathbb{T}_{p_{curr}}$ with the resulting test run path $\pi_{p_{curr}}$ (π for

short), we remove all taken i_δ from the corresponding cache: for all $\ddot{s} \xrightarrow{i_\delta} \ddot{s}'$ in π, remove i_δ from $c_{\ddot{s}}$. Alternatively, we can only remove all taken i_δ that were leading to a discharge: if the last TO discharge in the current execution sub-phase is by $\pi_{\leq k}$, then for all $\ddot{s} \xrightarrow{i_\delta} \ddot{s}'$ in $\pi_{\leq k}$, remove i_δ from $c_{\ddot{s}}$; so if no TO is discharged during execution of $\mathbb{T}_{p_{curr}}$, no i_δ is removed from any cache.

```
if (cacheMap(s̈) == ∅)                                                          1
then for each L_I ∪̇ {δ} i_δ ∈ in_{S_τ*}(s̈) ∪̇ {δ} do                            2
        L_U ∪̇ {i_δ} → WTTS(L_I, L_U, δ) l2WTC := l ↦                           3
          genWTS(S, s̈ after_{S_τ*δ} l, b − 1);
        WTTS(L_I, L_U, δ) 𝕎:=assembleWTC(s, path2W(π), l2WTC);                 4
        if (w(𝕎) < w_max) then break; fi;                                      5
        if (w(𝕎) > w_max)                                                      6
        then w_max := w(𝕎); cacheMap(s̈):=∅;                                    7
        fi;                                                                    8
        cacheMap(s̈).add(i_δ);                                                  9
     od;                                                                       10
fi;                                                                            11
L_I ∪̇ {δ} i_δ :=cacheMap(s̈).pollRandom();                                      12
L_U ∪̇ {i_δ} → WTTS(L_I, L_U, δ) l2WTC := l ↦                                   13
            genWTS(S, s̈ after_{S_τ*δ} l, b − 1);
WTTS(L_I, L_U, δ) 𝕎 := assembleWTC(s, path2W(π), l2WTC);                        14
return {𝕎};                                                                    15
```

Listing 13.1: Typed caching routine for genWTS($\mathcal{S}$, π, b)

i_δ caching has two advantages: whenever a cache is not empty, recursive calls of genWTS are not required for choosing i_δ, so we have lower runtime. As second advantage, the nondeterministic or random choice amongst all i_δ transitions to the heaviest WTCs, performed in the original genWTS, is restricted. Therefore, variance in the random algorithm is lowered and hence reproducibility is raised (cf. Subsec. 12.4.3). Due to the second advantage, i_δ caching is a refined heuristic that incorporates dynamic information. Since we could store the i_δ that should be avoided instead of those that should still be taken, this heuristic is a variation of the tabu search metaheuristic [Glover, 1989; Gass and Fu, 2013] for search-based software testing (cf. Subsec. 12.3.1). But computing once the set of outgoing i_δ that should be taken and then removing elements from, i_δ caching also has the first advantage of lower runtime.

This simple i_δ caching has three deficits by ignoring certain aspects:

Firstly, some $i_\delta \in c_{\ddot{s}}$ might lead to a lighter WTC, $\mathbb{W}_{i_\delta}$, than some $i'_\delta \notin c_{\ddot{s}}$. This can be caused by a $\underline{path2W_o}(\pi^{full})$ that depends on the path leading to $dest(\pi^{full}) = \ddot{s}$. So when π^{full}) is extended and $\ddot{s}$ visited again, the remaining i_δ in $c_{\ddot{s}}$ might no longer lead to the heaviest WTCs. But $\underline{path2W_o}$ is independent of the path leading to $\ddot{s}$ if it only considers $\ddot{s}$ (cf. FinalLocationSetDefaultPath2W in Subsec. 13.3.2), or if it accounts for $\mathcal{S}$ having fairness_{test} (cf. Def. 8.60).

Secondly, a discharge of a TO o that was still active the moment $c_{\ddot{s}}$ was computed also influences the weights. To countervail that some $i'_\delta \notin c_{\ddot{s}}$ leading to a heavier WTC is ignored, we can use an advanced variant of i_δ caching: Store in $c_{\ddot{s}}$ pairs $(i_\delta, \ddot{a})$, with

$\ddot{o}$ being the set of TOs active the moment $c_{\ddot{s}}$ is filled and discharging $o \in \ddot{a}$ causes the strongest decrease in $w(\mathbb{W}_{i_\delta})$. To compute $\ddot{a}$ efficiently, all *path2W* and all aggregation functions can be extended to yield $\ddot{a}$. When test execution discharges $o \in \ddot{a}$, all pairs from all caches that contain $\ddot{a}$ are removed. If the employed data structures cannot perform this sufficiently fast, genWTS can remove those pairs lazily in $c_{\ddot{s}}$ when visiting $\ddot{s}$.

Finally, the remaining depth $b - 1$ at $\ddot{s}$ is not taken into account in $c_{\ddot{s}}$ (cf. l.13 of genWTS($\mathcal{S}$, π, b) from Listing 13.1). But larger bounds might yield meaningful WTCs that are no extension of a WTC in genWTS($\mathcal{S}$, π, b') with $b' < b$. To yield more meaningful TCs and guarantee exhaustiveness, $c_{\ddot{s}}$ can store the depth of the computed WTCs and cause a recomputation if a greater depth is required.

If i_δ caching is employed in a parallel setting, the use of $c_{\ddot{s}}$ should also be parallelized since filling $c_{\ddot{s}}$ is performed with genWTS, which has high runtime. So $c_{\ddot{s}}$ could be shared between distributed nodes: whenever a distributed node finds an empty concurrent cache (or one with smaller depth), the node performs the computation needed to fill the cache. This can easily be implemented via Hazelcast, but with a risk of high contention.

13.6. Conclusion

13.6.1. Summary

LazyOTF has been implemented on top of JTorX, such that LazyOTF can delegate many tasks to JTorX, e.g., the exploration of the specification, its determinization, and test execution. LazyOTF has been integrated in JTorX and is available at [URL:JTorXwiki].

Core LazyOTF is implemented on top of STSimulator as own package within SymToSim. The configuration, management and user interaction is integrated in JTorX. Table 13.1 shows the rough structure and size of the implementation.

Table 13.1.: Structure and size of the implementation of LazyOTF

	core LazyOTF	JTorX integration: non-GUI	JTorX integration: GUI
packets	5	18	10
interfaces	12	39	29
classes	38	179	130
methods	285	1095	988
source LoC (SLOC)	2915	11950	12834

Furthermore, we introduced optimizations and an alternative implementation that does not use STSimulator and the treeSolver, but symbolic execution and SMT solving, as proof of concept.

13.6.2. Contributions

The implementations described in this chapter contributes:

1. the core LazyOTF (cf. Chapter 11), on top of STSimulator;

2. the heuristics framework (cf. Chapter 12), covering phase heuristics (via inducing states and bound heuristics) and guidance heuristics (via weights). For guidance heuristics, the implementation is even more flexible than the formal heuristics in Chapter 12;
3. connections to JTorX, its command line interface and its GUI, with extensions to configure (main LazyOTF tab, TO editor and weights editor) and operate (introspection window) LazyOTF;
4. minor extensions to STSimulator and JTorX (τ-closure optimization, exit criteria, alternative simulation, STS editor, STS Guides based on JTorX Guides), as well as bug fixes;
5. test adapter;
6. a proof of concept implementation for JTorX and LazyOTF with symbolic execution;
7. several optimizations like lazy traversal sub-phases, eager micro-traversal sub-phases, and parallelization.

Items 3, 4, and 5 were completely implemented by Felix Kutzner [Kutzner, 2014], item 6 by Florian Larysch [Larysch, 2012], both under the supervision of David Faragó.

13.6.3. Future

Sec. 13.5 designed i_δ caching, but its implementation and investigation left as future work, because i_δ caching is a full heuristic that might cause inefficiencies or even break exhaustiveness or other provisos (cf. Chapter 12).Furthermore, i_δ caching is not orthogonal to the lazy traversal sub-phase optimization and the eager micro-traversal sub-phase optimization (cf. Sec. 12.4.

Offering more coverage criteria is also future work: further general coverage criteria and coverage criteria that are not based on our core guidance heuristics, e.g., deterministic and nondeterministic coverage criteria via $\underline{covFiningPath2W}((\pi_i^{full})_i)$.

LazyOTF with symbolic execution (cf. Sec. 13.4) also offers future work, e.g., encoding superstates in SMT and employing compiler optimizations similar to Subsec. 7.1.3. However, a reimplementation of JTorX in combination with STSExplorer is the cleanest approach.

Notes. Further possible future work is the integration of test selection strategies via mutation testing and model-based mutation testing (cf. Subsec. 11.6.3) into our alternative implementation with symbolic execution, which speeds up model-based mutation testing [Aichernig and Tappler, 2015].

Furthermore, LLBMC's approach to detect equivalent mutants (cf. Subsec. 7.3.4) could be integrated. Since efficiently coping with equivalent mutants (resp. model-based mutants) would strongly improve test selection via mutation testing (resp. via model-based mutation testing), this is very promising.

Future work in the domain of software engineering is the traceability tab and fuzzy traceability.

14. Applications

14.1. Introduction

For the application of MBT, multiple aspects are relevant:

1. how efficient it is, i.e., how fast TCs are generated and how fast and meaningful the TCs are.
2. how usable it is: MBT must be understandable to the test engineer and software engineer, and easily applicable (usable specification language, usable configuration, automation);
3. how well it fits into the software engineering process; what aspects of software engineering it covers (regression testing; short failing test runs; reproducibility; traceability; metrics; communication);

This thesis thoroughly covered feasibility (mainly item 1, but occasionally also items 2 and 3) theoretically. Additionally, the case study in Sec. 14.3 investigates feasibility empirically. To investigate items 2 and 3 further, large industrial case studies have to be performed (e.g., similar to [Sijtema et al., 2014] for JTorX, possibly larger), which is future work. These aspects are a major part in the articles cited in the next paragraph. Furthermore, item 3 is influenced by items 1 and 2. Item 3 is investigated in Sec. 14.2 for agile software development.

Taking all these aspects into account by financially quantifying their costs and benefits results in the **return on investment** (**ROI**). Though it is difficult to sufficiently consider all aspects and quantify their costs and benefits, the ROI is an important metric for MBT's acceptance and adoption. Hence, many case studies, surveys and publications investigate the ROI [Weißleder et al., 2011; Binder, 2011; Mlynarski et al., 2012; Faragó et al., 2013; URL:ROIblogpost].

14.2. Agile Software Development and *refines*

14.2.1. Introduction

To apply MBT, it needs to be embedded in the software development process, and agile software development (AD) is the state of the art: two out of three organizations have adopted AD according to the study [Ambler, 2008], one out of three according to [Kerner, 2009]. Furthermore, the **Agile Conference** had a growth of 40% in 2009. This section investigates the integration of MBT and AD and is based on [Faragó, 2010a; Faragó, 2010].

MBT and AD are two major approaches to increase the quality of software, but AD's strength is validation, as opposed to MBT's strength of dynamic verification. Therefore, the advantages of the combination are investigated, too.

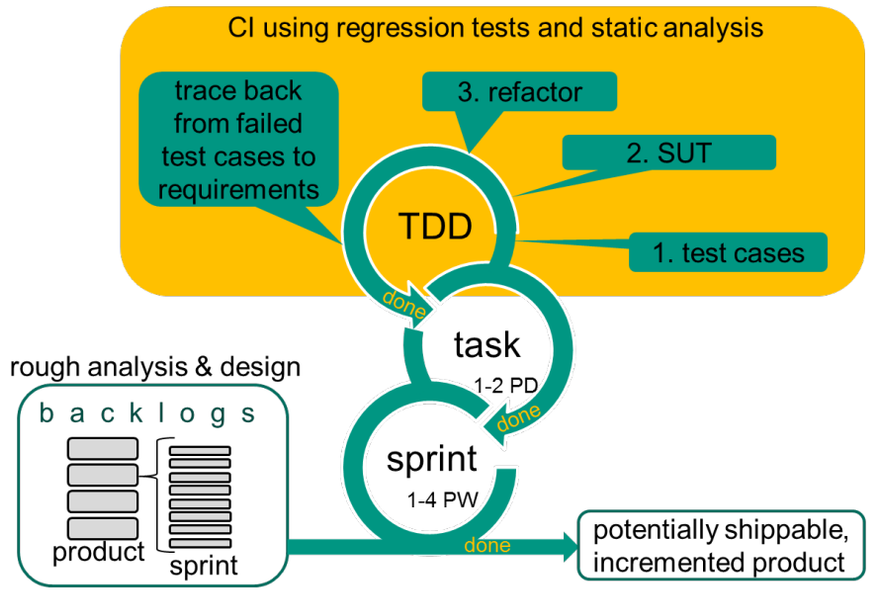


Figure 14.1.: Exemplary agile methods: XP and Scrum

Roadmap. Subsec. 14.2.2 gives an overview of AD's values and then shortly introduces the two most popular agile methods: XP and Scrum, which we will focus on. Subsec. 14.2.3 motivates the integration of AD and MBT, lists the demands on tightly integrating AD and MBT, describes how they are solved with the help of underspecification and refinement hierarchies, and describes all resulting benefits. Finally, related work is given.

14.2.2. Agile Development

In short, **agile software development** (**AD**) [Shore and Warden, 2007; Faragó, 2010] is iterative software development in short cycles, such that requirements and solutions can evolve. This is supported by a set of engineering best practices, such that high-quality software increments can be delivered rapidly.

The big picture on how AD aims at better software development is given by **the values of AD**, stated in the **Manifesto for Agile Software Development** [Fowler and Highsmith, 2001]:

1. value: individuals and interactions over processes and tools;
2. value: working software over comprehensive documentation;
3. value: customer collaboration over contract negotiation;
4. value: responding to change over following a plan.

Fig. 14.1 sketches an example of how software development achieves these values. Two of the most prominent agile methods are applied, which can be combined easily: Extreme Programming (XP) [Team", 1998; Jeffries et al., 2000] and Scrum [Takeuchi and Nonaka, 1986]. Other agile methods (e.g., the Agile Unified Process [Ambler, 2002] or Feature Driven Development [Palmer and Felsing, 2002]) lead to the same implications on MBT.

AD achieves rapid delivery (cf. 2. value) by short (a few weeks) development iterations,

called **sprints** in Scrum: In each sprint, the team implements one or more pending features/customer requirements, managed in the **product backlog**. These are often formulated as **user stories**, which are lightweight requirements – a few sentences in natural language, but too abstract to be understood on their own. The user stories are refined into tasks, which are put into the **sprint backlog**. By iterating over all tasks, the sprint is implemented. The result is a potentially shippable, incremented product.

In more detail, each task should be completable in about one person day. Within a task, the developer practices even shorter iterations using **test-driven development** (**TDD**), where software is developed in a very short cycle called **TDD cycle** [Beck, 2002]:

1. Before writing production code, one TC that fails is added to (or modified in) the TS $\ddot{\mathbb{T}}$ on the unit level;
2. only then the production code is changed, in a minimal way to pass $\ddot{\mathbb{T}}$ again;
3. finally, refactor the production code (and $\ddot{\mathbb{T}}$) for cleaner code according to some standard.

Therefore, this cycle is also called **red-green-refactor-cycle**. Similarly to formal methods, automation also helps in AD to increase efficiency and decrease errors. Automation is thus supported by many tools. For writing production code such that $\ddot{\mathbb{T}}$ passes again, intelligent code completion of modern IDEs offer semi-automation in writing source code. For more complex TC failures, the developer can use debugging and trace back from the TC to the corresponding feature/requirement (cf. Chapter 2).

To always assure the 2. value in spite of flexibly being able to respond to change and in spite of many small increments via the TDD cycle, AD practices **continuous integration** (**CI**) [Duvall et al., 2007]: the work of all developers is continuously (several times a day) integrated into a shared mainline to prevent problems later on. The shared mainline is checked by a CI framework in the background (e.g., Jenkins or CruiseControl) after each integration whether

- the source code is clean, follows standards and best practices using static analysis;
- and whether it is correct using $\ddot{\mathbb{T}}$ and TCs on higher levels of the V-model (cf. Sec. 2.4) as regression tests. Usually, AD focuses on unit tests and acceptance tests [Rainsberger, 2009], but if less modular software is developed, more tests have to be performed by integration instead of unit tests. For high usability, testing within CI should be fully automatic and fast, such that developers get quick feedback (optimally, a 10-minute build including all of CI's tasks is reached).

The agile team defines exit criteria for tasks and sprints (mainly based on metrics from static analysis and testing). These **definitions of done** (**DoD**) are then configured within the CI framework to be checked automatically.

Using agile processes, rapid delivery of high quality working software increments can be achieved. These are shown to the customer, so that customer feedback can be early and flexibly incorporated in the following iterations. Therefore, AD has strong **validation** (derived from the Latin word for "to be worthy"), i.e., checking that the SUT really fulfills the needs of its users (often formulated as "checking that the right product is built") [Boehm, 1984; Balzert, 1997; Prenninger and Pretschner, 2005; Pezzé and Young, 2007]. This is orthogonal to verification (derived from the Latin word for "truth"), i.e., checking that the SUT really fulfills its specification (often formulated as "checking that the product is built the right way"). From the perspective of a given specification $\mathcal{S}$,

verification checks that the implementation conforms to $\mathcal{S}$ and validation checks that $\mathcal{S}$ is what the customer wants.

Since validation narrows the gap between the specification and what the customer actually wants, integrating AD with MBT is helpful in software development. This integration is investigated in the following subsection.

14.2.3. Integrating MBT and Agile Development

Motivation

By integrating MBT and AD, deficits from both AD and MBT can be avoided:

A main deficit of AD is that too little specification and documentation is delivered. Even though working software is more important (cf. 2. value), specifications and documentations are necessary in today's component-oriented software development, since components need to be specified to reuse, distribute and certify them (especially for safety-critical applications). More precise documentation is also helpful for developers so they have direction and knowledge of the purpose while navigating through source code.

AD also has some difficulties in testing, even though testing is an integral part of AD (cf. the previous subsection): The used test coverage is still often insufficient and deceptive (e.g., 60% statement coverage, cf. [Lawrance et al., 2005], more generally see Subsec. 12.3.1). Manually written test cases are less flexible and require more maintenance than the specifications that MBT requires. For instance, if exception handling is refined, a lot of test cases might have to be modified to incorporate this.

Finally, tracing back from failed test cases to user stories is often difficult in AD.

The deficit of most applications of MBT (and most other formal methods) is their **big design up front** (**BDUF**): MBT requires the entire specification of the final product. This results in a rigid process, where gains of MBT are only possible after investing a lot of time into specifying, faults are detected late, and validation takes place only at the end or not at all.

Tight Integration of MBT and AD

Applying MBT within AD, the deficits of MBT must be avoided: Firstly, being **flexible**, as result from the 1. value, the 3. value and the 4. value. Secondly, avoiding a BDUF by **rapidly delivering** working products, as result from the 2. value. Furthermore, dynamic verification should be fast and automatic, to keep up with AD's fast iterations and flexibility, and to be able to integrate the dynamic verification process into the CI framework.

These requirements imply using lightweight formal methods. Very lightweight static analysis tools (such as FindBugs [Ayewah et al., 2007]) are already an integral part of AD since they are often plugged into IDEs and the CI framework. But they unfortunately produce many false negatives or false positives. MBT is hence a relevant alternative since it is lightweight, generates tests automatically from specifications, can easily be integrated into AD (technically as well as psychologically since agile teams are accustomed to testing), and yields further advantages such as traceability and metrics.

To be flexible and avoid a BDUF [de Vries et al., 2002], MBT needs to use abstractions, e.g., underspecification. To be used in AD's iterations, it must be iteratively refineable, so that details can be introduced lazily to the specifications, and test generation must efficiently handle this. Therefore, this section uses a refinement hierarchy and the *refines* relation (cf. Sec. 9.3). This increases MBT's usability in all fields [Peleska, 2013].

Notes. The level of abstraction could also be influenced by the test adapter (cf. Subsec. 8.7.2 and Sec. 9.3), but those are lossless abstractions. Underspecification is lossy and thus more helpful if not all information is available yet.

Example 9.10 on page 231 shows a refinement hierarchy for some web services. Web services are a prominent application of MBT and are also frequently used in AD (and sometimes called **Agile Applications**), since their design concept supports loose coupling, rapid delivery and AD.

Integrating MBT and AD, the iterative and incremental processes in AD can also be applied on the specification-level, i.e., to the **refinement hierarchy**:

- starting with the most abstract specifications (e.g., Figure 9.1(a)),which replace user stories and support flexibility and communication;
- within sprints, iteratively add aspects by refining more abstract specifications into less abstract specifications, for rapid delivery;
- the most refined specifications are sufficiently detailed for MBT.

Using the refinement hierarchy within AD, MBT and AD can be tightly integrated, in both the CI framework and the TDD cycle:

- the CI framework uses MBT for metrics and for regression testing;
- in the TDD cycle, tests are no longer written manually; instead, specifications are written (and refactored) resulting in **specification-driven development**. The tests are generated from the specifications automatically via MBT.

The result is called **MBTAD** and depicted in Fig. 14.2.

MBTAD avoids the deficits of both AD and MBT, and yields further advantages:

MBTAD produces working software increments plus the corresponding specification increments, resulting in a **specification hierarchy** (cf. V-model in Sec. 2.4). The specification hierarchy aids certification, re-usability, communication and orientation during programming.

Testing is now based on the ioco theory, performing **continuous** *refines* **checks**. Therefore, TCs can be generated efficiently and automatically, with high coverage and meaningfulness. MBT provides more coverage criteria, which not only produce better tests, but also better measurements for quality management. For instance, andrena object's agile quality management ISIS [Rauch et al., 2008] is state of the art and also uses test coverage, measured using EclEmma; but that only allows limited coverage criteria, namely basic blocks, lines, bytecode instructions, methods and types (cf. Subsec. 13.2.4). Since more sophisticated coverage criteria are often more meaningful (cf. Sec. 2.5, Subsec. 12.3.1), MBTAD can also improve agile quality management.

The specification hierarchy is more flexible to change, and maintenance is easier than of the TS. For instance, if exception handling is refined, MBTAD no long has to modify a lot of test cases, but only the concise changes in the specification hierarchy that exactly reflects the exception handling refinement (cf. Fig. 9.1 (c) to (d)). This concise formalism also increases efficiency in the TDD cycle.

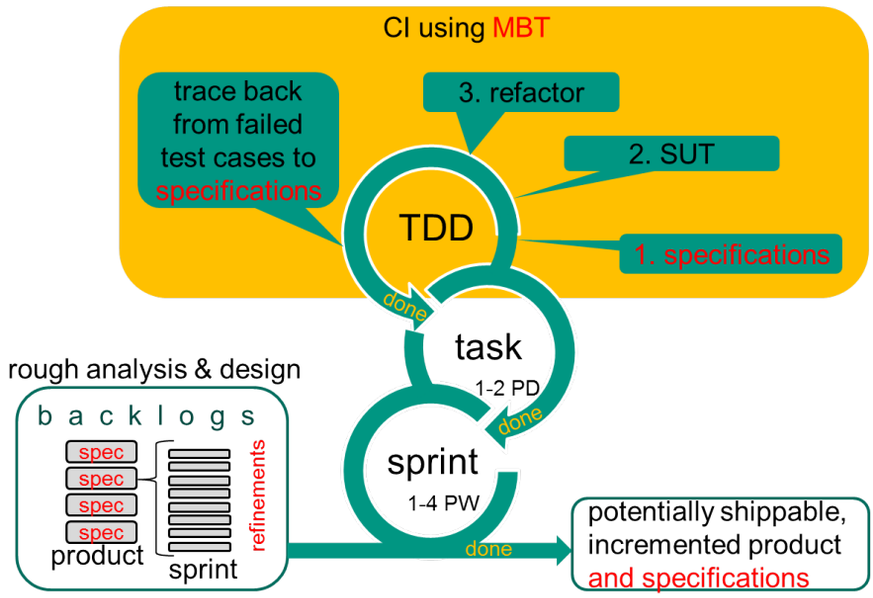


Figure 14.2.: MBT in our exemplary agile methods

MBTAD empowers one modeling language with many levels of abstraction, improving expressiveness, usability, redundancy and maintenance. The more abstract specifications are used for business-facing [Crispin and Gregory, 2009], in particular for communication and to give an overview, and are a formal (graphical) replacement for user stories. They are, however, also the origin for refinements to the more concrete specifications, which are used for technology-facing, in particular for automatic test case generation. Although formal abstract specifications are slightly less intuitive than user stories, the higher precision is more important to avoid errors and misunderstanding, especially in safety-critical domains. In cases when technical details need to be considered early and complex architectures, business logic or product logic need to be modeled, UML state machines and use cases are often employed [Cockburn, 2000; Weißleder, 2009; Mlynarski, 2011]). Our specifications can replace these and need not determine whether business- or technology-facing and which degree of abstraction should be used.

Our automatic generation of TCs can easily implement traceability between requirements and TCs as described in Subsec. 13.3.5, but also vertical traceability between two consecutive levels $\mathcal{S}', \mathcal{S} \in \mathit{SPEC}$ of the specification hierarchy: $\mathcal{S}'$ *refines* $\mathcal{S}$ guarantees that a trace $\sigma \in \mathit{Straces}_{\mathcal{S}'_{\tau^*\delta}} \cup \mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}$ is a common trace of both $\mathcal{S}'$ and $\mathcal{S}$ or corresponds to a feature change: a new feature via added input on the refinement level of $\mathcal{S}'$, i.e., $\sigma \in \mathit{Straces}_{\mathcal{S}'_{\tau^*\delta}} \setminus \mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}$, or a restricted feature via removed output on the refinement level of $\mathcal{S}'$, i.e., $\sigma \in \mathit{Straces}_{\mathcal{S}_{\tau^*\delta}} \setminus \mathit{Straces}_{\mathcal{S}'_{\tau^*\delta}}$. Therefore, we choose a trace link (which we call **strong trace link**) between a specification element (state or transition) $e' \in \mathcal{S}'$ and $e \in \mathcal{S}$ if there is a common trace $\sigma \in \mathit{Straces}_{\mathcal{S}'_{\tau^*\delta}} \cap \mathit{Straces}_{\mathcal{S}_{\tau^*\delta}}$ that leads to e' in $\mathcal{S}'$ and to e in $\mathcal{S}$. If e' corresponds to a new feature, then we do not add a strong trace link. We can, however, add another kind of trace link, which we call **weak trace link**, between e' and the element $e \in \mathcal{S}$ that is the last element on σ that has a strong trace

link, i.e., if $\exists k \in \mathbb{N}$ with $k < |\sigma|$ and $\sigma_{\leq k}$ leads to e and $\sigma_{\leq k} \in \mathit{Straces}_{\mathcal{S}'_{\tau * \delta}} \cap \mathit{Straces}_{\mathcal{S}_{\tau * \delta}}$ and $\sigma_{\leq k+1} \notin \mathit{Straces}_{\mathcal{S}_{\tau * \delta}}$. Therefore, the weak trace link indicates that $\sigma_{\geq k+1}$ directly after e (resp. directly after $\mathit{strongTraceLink}(e)$) is the reason why e' has no strong trace link. Symmetrically, if e corresponds to a restricted feature, then we add a weak trace link between e and the element $e' \in \mathcal{S}'$ that is the last element on σ that has a strong trace link. In short, vertical traceability in our specification hierarchy is determined by reachability through common traces and allows tracing back from failed TCs through the specification hierarchy all the way up to the most abstract specifications that replace user stories. The continuous *refines* checks performed by MBT on the most refined specifications also guarantee conformance with each more abstract specification, thus showing the conformance with the most abstract specification, which replaced user stories.

Note. If the partial mapping between the elements of $\mathcal{S}'$ and $\mathcal{S}$ defined by trace links should be a bijection between its domain and its image, then $\mathcal{S}$ and $\mathcal{S}'$ should be deterministic and each trace link identified by the trace σ leading to the elements e', resp. e; alternatively, if $\mathcal{S}$ and $\mathcal{S}'$ are computation trees, each state is only reachable by a unique trace, so identifying each trace link by its corresponding trace is no longer necessary.

The implementation of traceability can make use of the implementation of *refines* (cf. Subsec. 9.3.1): Since the implementation of *refines* must guarantee that no input is remove in e' compared to e, and no output added, the implementation already considers the strong trace links and must only record them. To implement weak trace links, the implementation of *refines* must be extended to store added inputs and removed outputs, or perform a reachability analysis. Our implementation of *refines* via the iocoChecker performs such a reachability analysis with the help of a DFS anyways. Just like the regular implementation of automatic traceability (cf. Subsec. 13.3.5) is left as future work, so is the implementation of this vertical traceability in our specification hierarchy.

Refinement and AD applied to specifications avoid a BDUF for specifications, which leads to higher usability and ROI, to fail fast and lower risks. Furthermore, agile methods as pair programming, reviews, and CI help find defects in the specifications early. Because of AD's strong validation, differences between the specification and the customers expectations are also detected early.

Using LazyOTF for MBTAD, abstract specifications and nondeterministic systems are handled efficiently. LazyOTF's strong guidance generates meaningful TCs with high coverage levels, resulting in higher quality of testing and hence better software development. The meaningful TCs are short and can avoid nondeterminism as much as possible, resulting in high usability, reproducibility and traceability. For the CI framework, better coverage criteria improve the metrics for quality management, and reproducibility improves automated regression testing.

In summary, AD requires rapid delivery and CI with regression tests. Hence MBT can profitably be applied to AD: Efficient tests can be generated and executed automatically with an appropriate coverage.

Related Work

There is some work on MBT with AD previous to [Faragó, 2010]: [Utting and Legeard, 2007] scarcely considers using AD to improve MBT and also MBT within AD. It suggests

MBT outside the AD team, i.e., not strongly integrated. [Puolitaival, 2008] aims to adapt MBT for AD and also shortly motivates using MBT within the AD team, but does not investigate in detail how to modify AD for fruitful integration, e.g., adjusting specifications and CI. It rather focuses on a case study, which empirically shows that abstraction is very important in MBT for AD. [Katara and Kervinen, 2006] uses a strict domain and a limited property language. It uses very restricted $\mathcal{S}$ that are lists of exemplary paths. [Rumpe, 2006] gives a good overview of MBT when evolution (as in AD) is involved. It uses the same modelling language for the production system and the tests, but not the same specifications. This section links abstract specifications of the product and more detailed specifications for MBT, reducing work and redundancy. [Rainsberger, 2009] states that contract-based testing should replace integration tests in AD. The combination of AD and safety-critical software is being investigated in the project **AGILE** [OpenDO]. More generally, [Black et al., 2009] gives a broad overview of the combination of AD and formal methods. In **agile modeling** [Ambler, 2002], the relationship between specifications and AD are investigated, but testing is not considered thoroughly.

More recently, [Binder, 2011] showed that AD is already used by some applications of MBT.

14.2.4. Summary

This section showed how MBT and AD can be combined, using underspecification and *refines* for the specifications to avoid rigidness and a BDUF. Tightly integrating MBT and AD yields the highest profits: specifications with high flexibility, low redundancy and low maintenance, effective CI with high coverage and continuous ioco checks. Various overage criteria can also be used for quality management. Further advantages are traceability, higher reproducibility and efficiently handling abstract specifications and nondeterministic systems, especially when applying LazyOTF.

14.3. Experiments

14.3.1. Introduction

This thesis theoretically investigated the efficiency of LazyOTF (cf. Chapter 11) and the applied heuristics (cf. Chapter 12). In this section, the efficiency is measured empirically with experiments, considering how meaningful the results are and how fast its TC generation is. Since meaningfulness is approximated by guidance heuristics, configured by TOs, this section uses several sets $\ddot{o}$ of TOs. Meaningfulness of the generated TCs is measured by counting the overall number of test execution steps t_{curr}^{max} required to discharge all TOs in $\ddot{o}$. The meaningfulness of the generated TCs also influences other aspects of applying MBT (see also Sec. 14.1, [Nieminen et al., 2011; Anand et al., 2013; Gay et al., 2015]): If less test steps are required, TC generation and TC execution are faster (cf. Subsec. 11.2.3), TCs and failing test runs are more understandable and reproducible. Despite having an effect on these performance attributes, t_{curr}^{max} is less machine-dependent than those attributes [Bader et al., 2000]. Finally, meaningfulness can have a stronger impact on the required resources than the complexities per test step

(cf. Chapter 11). Therefore, this section focuses on measuring meaningfulness, like other work [Anand et al., 2013; Gay et al., 2015].

Roadmap. After describing the specification and configuration in the next two subsections, Subsec. 14.3.4 compares JTorX's OTF and LazyOTF. Subsec. 14.3.5 compares guidance heuristics settings. Subsec. 14.3.6 investigates dynamic bound heuristics settings. Subsec. 14.3.7 investigates parallelization using distributed LazyOTF. Subsec. 14.3.8 covers testing of a real SUT. Subsec. 14.3.9 depicts results of preliminary further experiments. Finally, Subsec. 14.3.10 concludes our experiments, especially by considering their validity.

14.3.2. Case Study and Its Specification

The experiments in this section are conducted within a case study (cf. Example 9.10) from the domain of service-oriented architecture: web services from WIBU SYSTEM AG's **CodeMeter License Central** [URL:LC], for managing and distributing licenses; licenses can be generated, removed, backed up and inspected. This SUT is called $\mathbb{S}_{LC}$ and was developed mainly by Stefan Nikolaus.

We created a strongly abstracted specification, described by an STS called $\boldsymbol{\mathcal{S}_{LC}}$ and specified in detail in Appendix B.1.1. $\mathcal{S}_{LC}$ contains a hard-coded parameter: the constant $MAXPRT \in \mathbb{N}_{>0}$ describing the maximal portion for pagination (see next paragraph).

$\mathbb{S}_{LC}$ and $\mathcal{S}_{LC}$ are designed for web services, using the request-response pattern [W3C, 2001] and fault handling via response messages and additional exceptional behaviors. Managing licenses is specified by the following features, all having appropriate exception handling:

- **generateLicense** generates a new license, returning its ID;
- **removeLicense** removes the license with the given ID;
- **removeExpired** removes all licenses that are currently expired. Since expiration is not modeled by $\mathcal{S}_{LC}$, the abstraction nondeterministically removes a subset of all present licenses;
- **removeAll** removes all licenses;
- **showLicenses** lists all present licenses using portions of maximal $MAXPRT \in \mathbb{N}_{>0}$ many licenses, to support pagination and **packages for transmission**, i.e., successively $MAXPRT$ many licenses can be requested;
- **remoteBackup**, resp. **localBackup**, to perform remote, resp. local, backups.

Since STSs used in JTorX do not offer container data types (cf. Sec. 13.4), $\mathcal{S}_{LC}$ abstracts from the set of licenses to its cardinality. Consequently, the License Central web services are described very abstractly, reducing the fault detection capability. To check concrete licenses, more expressive STSs would have to be used (cf. Sec. 13.4), or the test adapter would have to keep track of the licenses. But we prefer to stay on this abstract level for our experiments since this strong abstraction simplifies the case study, such that several aspects can be investigated quicker and clearer, and JTorX's expansion of $\mathcal{S}_{LC}$ to the level of LTSs (cf. Sec. 9.4 and Sec. 13.4) causes no problem: no reduction heuristic can break exhaustiveness for $\mathcal{S}_{LC}$ since guards only allow few transitions to states other than fail (cf. Subsec. 13.2.1). Furthermore, $\mathcal{S}_{LC}$ exhibits the relevant aspects of an LTS:

- all kinds of nondeterminism:
  - controllable nondeterminism, especially in the abstract state `initLocation`, as $\mathcal{S}_{LC}$ models a reactive system;
  - uncontrollable nondeterminism on output, mainly due to exception handling, in all abstract states `*Requested`;
  - uncontrollable nondeterminism of the LTS due to nondeterministic branching: `removeLicenseRequested` $\xrightarrow{\texttt{!removeLicenseOutput<returnCode>,[returnCode==}\texttt{"UnknownException"]}}$ `removeLicenseUnknownException` and `removeLicenseRequested` $\xrightarrow{\texttt{!removeLicenseOutput<returnCode>,[returnCode==}\texttt{"LicenseNotFoundException"||(returnCode=="UnknownException"\&\&licenseCount==0)],}}$ `removeLicenseLicenseNotFoundException`;
  - uncontrollable nondeterminism of the LTS due to τ transitions in `removeExpiredRequested`;
- a τ-cycle: `removeExpiredRequested` $\xrightarrow{(unobservable)}$ `removeExpiredRequested`;
- irregular underspecification of input for superstates containing multiple `initLocation` with `licenseCount` $= 0$ and `licenseCount` > 0 due to the switch `initLocation` $\xrightarrow{\texttt{!removeExpiredLicensesInput<>,[licenseCount>0],}}$ `removeExpiredRequested`;
- the size of $[[(\mathcal{S}_{LC})_{\mathcal{V}^d_{LC}}]]$ ($[[\mathcal{S}_{\mathcal{V}^d_{LC}}]]$ for short) and of its resulting superstates is only bounded by the size of the type used for bounded non-negative integers. Since the 32 bit integer type is used, $[[\mathcal{S}_{\mathcal{V}^d_{LC}}]]$ and superstates can become huge if they are not effectively processed on-the-fly;
- we can easily express test objectives over $\mathcal{S}_{LC}$ that become increasingly difficult to discharge with increasing $MAXPRT$, e.g., the test goal $\ddot{s} := \{\texttt{showMoreLicenses}\} \times D^{\mathcal{V}}$ since the abstract state `showMoreLicenses` is a deep state in $[[\mathcal{S}_{\mathcal{V}^d_{LC}}]]$: Due to its guard, `showMoreLicenses` is only reachable via paths of length greater $2 \cdot MAXPRT + 3$. Reaching $\ddot{s}$ is particularly difficult for random testing: If some transitions are selected with probability smaller 1, the probability of reaching `showMoreLicenses` from `initLocation` with $b \in \mathbb{N}$ steps quickly decreases with increasing $MAXPRT$ (shown empirically in the next subsection).

14.3.3. Configuration

Our experiments will vary some parameters for comparison: $\ddot{o}$, *aggWTCs*, bound heuristics, $MAXPRT$, the number P of parallel instances, the SUT, and the MBT approach. Other parameters will be fixed throughout this section: the specification $\mathcal{S}_{LC}$, the implementation relation *ioco*, no user interaction, *aggPath2Ws*$(\cdot)$, the software and hardware, and almost always the activated optimizations, communication, and logging.

Varying Parameters

SUT. As SUT, this section mainly uses $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$, i.e., $\mathcal{S}_{LC}$ with the simulation capabilities of JTorX (cf. Subsec. 10.3.3), to factor out the runtime of test execution as much as possible when measuring LazyOTF's test case generation and heuristics: Since they are independent of the test case execution, simulation investigates them more accurately since the simulated test execution is fast, exactly on the abstraction level of

$\mathcal{S}_{LC}$, and independent of any real SUT and its technicalities. The simulation is *ioco*, has fairness_{test} without underspec_U, and a reliable reset capability since $\mathcal{S}_{LC}$ is an SCC. For strong statistical randomness yet fast performance, we integrated the pseudorandom number generator Mersenne Twister [Matsumoto and Nishimura, 1998] into the simulation. When test execution is performed on the real SUT $\mathbb{S}_{LC}$ (cf. Subsec. 14.3.8), we use JTorX with an own test adapter, which is a simple solution allowing comparison with our other experiments.

Bound heuristics. Most experiments do not focus on the different bound heuristics settings. For them, we get clean measurements by deactivating the bound heuristics and using a fixed bound of 5 instead. For Subsec. 14.3.6, we vary the bound heuristics settings as described there.

aggWTCs. Most experiments do not focus on the different guidance heuristics settings, for which we fix PInputDefault; for Subsec. 14.3.5, we do vary over our pre-built *aggWTCs* to compare them: PInputDefault, MaxMax, and Max.

Distributed LazyOTF. We mostly consider the sequential LazyOTF, i.e., fix $P = 1$. To measure the parallel speedup in Subsec. 14.3.7, the number P of parallel instances of distributed LazyOTF is varied.

Test Objectives. In most subsections, we consider the following TOs o (resp. $\ddot{o}$) to cover various use cases:

$\boldsymbol{o_{RSML}}$ is a reachability TO for the abstract state `showMoreLicenses`, testing pagination (i.e., the safety property $\exists\Diamond \text{pageNo} > 0$). It is composed of two auxiliary TOs:

$$\underline{path2W}_{o_{RSML}}(\pi^{full}) := \{awardLicenseCount(\pi^{full}), awardTG(\pi^{full})\}$$

with

- $awardLicenseCount(\pi^{full})$ being $dest(\pi^{full})$'s *mean* `licenseCount` $\cdot 1000$, implemented via FinalLocationSetDefaultPath2W(LocationValuation2Weight). For *max* instead of *mean*, *awardLicenseCount* would be a pure fancy_{linear} *path2W* (cf. Example 12.43); *mean* approximates a *distance* function, but additionally fines nondeterminism. Since o_{RSML} was still always discharged in our experiments, this choice is better in practice;
- $awardTG(\pi^{full}) := \begin{cases} 10^7 & \text{if } I^{lazy}_{o_{RSML}}(dest(\pi^{full})) = \text{TESTGOAL} \\ 500 & \text{if } I^{lazy}_{o_{RSML}}(dest(\pi^{full})) = \text{ORDINARY}, \\ 1 & \text{if } I^{lazy}_{o_{RSML}}(dest(\pi^{full})) = \text{INDUCING}; \end{cases}$
  which is a nonfancy *path2W* (cf. Example 12.43), implemented via FinalLocationSet−DefaultPath2W(TestTypeDefaultNode2Weight).

$discharge_{o_{RSML}}$ is the default discharge function.

$$I^{lazy}_{o_{RSML}} : s \mapsto \begin{cases} \text{TESTGOAL} & \text{if } s = \texttt{showMoreLicenses} \\ \text{ORDINARY} & \text{otherwise}; \end{cases}$$

$\ddot{o}_{cov}$ is a TO composed of a set of reachability TOs, implementing abstract state coverage of $\mathcal{S}_{LC}$, except `showMoreLicenses`, which is covered by either o_{RSML} or o_{NFRSML} below. Thus exactly for each $x \in S_{LC} \setminus \{\texttt{showMoreLicenses}\}$, $\ddot{o}_{cov}$ contains a coverage task TO o_x with

$$I^{lazy}_{o_x} : s \mapsto \begin{cases} \text{TESTGOAL} & \text{if } s \in x \times D^{\mathcal{V}} \\ \text{ORDINARY} & \text{otherwise;} \end{cases}$$

and

$$\underline{path2W}_{o_x}(\pi^{full}) := awardTG(\pi^{full}).$$

$discharge_{o_x}$ is the default discharge function.

Since two abstract states in $S_{LC} \setminus \{\texttt{showMoreLicenses}\}$ have at most a distance of 6 transitions, all TOs in $\ddot{o}_{cov}$ are discharged efficiently. Hence we consider $\ddot{o}_{cov}$ as fancy TO, i.e., plot it together with other TOs that have fancy $\underline{path2W}$.

o_{NFRSML} is a reachability TO defined just like the coverage task TOs in $\ddot{o}_{cov}$, but for the abstract state `showMoreLicenses`. So it has nonfancy guidance that does not take `licenseCount` into account, as opposed to o_{RSML}, which employs *awardLicenseCount*. For nonfancy guidance, we can still use the function I^{lazy} to reduce the size of the sub-graphs by setting `removeAllRequested` to INDUCING.

$\ddot{o}_{S_{isol}}$ is a TO composed of a set of reachability TOs, implementing S_{isol}, i.e., reachable state coverage in isolation (cf. Subsec. 8.8.4, Subsec. 11.2.4, and Subsec. 13.3.3) of $\mathcal{S}_{LC}$, but only up to a constant $k \in \mathbb{N}_{>0}$ to stay feasible. As for $\ddot{o}_{cov}$, `showMoreLicenses` is excluded, which can only be reached for `licenseCount`$> MAXPRT$ anyways. Thus exactly for each reachable state x in $[[\mathcal{S}^d_{\mathcal{V}_{LC}}]]$ that has a `licenseCount`$\leq k$ and is not a `showMoreLicenses`, $\ddot{o}_{S_{isol}}$ contains a coverage task TO o_x with

$$I^{lazy}_{o_x} : S_{det} \to \Sigma_{lazy}, \ddot{s} \mapsto \begin{cases} \text{TESTGOAL} & \text{if } \ddot{s} = \{x\} \\ \text{ORDINARY} & \text{otherwise,} \end{cases}$$

implemented with the help of the SatStrategy decision strategy, and the nonfancy

$$\underline{path2W}_{o_x}(\pi^{full}) := awardTGdet(\pi^{full}),$$

with

$$awardTGdet(\pi^{full}) := \begin{cases} 10^6 & \text{if } I^{lazy}_{o_{RSML}}(dest(\pi^{full})) = \text{TESTGOAL} \\ 100 & \text{if } I^{lazy}_{o_{RSML}}(dest(\pi^{full})) = \text{ORDINARY} \\ 1 & \text{if } I^{lazy}_{o_{RSML}}(dest(\pi^{full})) = \text{INDUCING,} \end{cases}$$

implemented via DetFinalLocationSetPath2W(TestTypeDefaultNode2Weight). $discharge_{\ddot{o}_{S_{isol}}}$ only discharges if the final superstate is a TG, i.e., $\ddot{s} = \{x\}$.

Since we expect nondeterminism of the LTS to cause only few faults, $\ddot{o}_{S_{isol}}$ is suitable for testing the real web service with the abstract $\mathcal{S}_{LC}$, and for testing the simulated SUT. The bound k is necessary since the state variables are 32 bit integers; hence the user-supplied bound k limits the state variable `licenseCount` to the interval $[0, \ldots, k]$; all other variables are hereby also strongly limited.

o_{dec} is a TO that goes beyond reachability: It is discharged when `licenseCount` is decreased $d \in \mathbb{N}_{>0}$ times via `removeLicenseRequested`, without increasing `licenseCount`

in between. The function $discharge_{o_{dec}}$ is implemented programmatically using the corresponding observer interface. The function $\underline{path2W}_{o_{dec}}(\pi^{full})$ is more complex than for other TOs and implemented with two auxiliary TOs o_{up}, o_{down}:

- o_{up} is for guidance to a state with sufficiently high `licenseCount` for subsequent d decreases, with the fancy *awardLicenseCount* as $\underline{path2W}_{o_{up}}$, and discharged by setting o_{inc}'s test goals correspondingly;
- o_{down} is for guidance in the decreasing phase, with fancy$_{nonlinear}$ $\underline{path2W}_{o_{down}} = 1000+2000/(\overline{lc}+1)$ and $\overline{lc} = dest(\pi^{full})$'s *mean* `licenseCount` (cf. configuration of o_{RSML}), implemented via FinalLocationSetDefaultPath2W(LocationValuation2Weight). o_{down} is discharged when o_{dec} is fully achieved, via the programmatic $discharge_{o_{dec}}$ observer.

The function I^{lazy} is also used to aid guidance by setting `removeExpiredRequested`, `removeAllRequested`, and `backupMenu` to INDUCING.

MAXPRT. We scale $\mathcal{S}_{LC}$ in size and complexity via *MAXPRT*. The TOs o_{RSML} and o_{NFRSML} (cf. state variable `licensesDisplayed`) depend on *MAXPRT* and are increasingly difficult to discharge with increasing *MAXPRT*. $\ddot{o}_{S_{isol}}$ (resp. o_{dec}) are increasingly difficult to discharge with increasing k (resp. d). Because of similarity and simplicity, we set $k = d = MAXPRT$ in our experiments. Therefore, we can measure whether the efficiency and meaningfulness of LazyOTF with the above TOs (and of JTorX) scales with increasing *MAXPRT*. Scalability of $\ddot{o}_{cov}$, however, is trivial, since it does not depend on *MAXPRT*.

Variable parameters that are measured. The following parameters are neither fixed nor iterated over by the experiment settings, but measured to observe the experiment's outcome. We will mainly measure meaningfulness by counting the **overall number of test steps**, $\boldsymbol{t_{curr}^{max}}$ until termination, i.e., until all TOs are discharged. Furthermore, we will measure the number of (internal) discharges and many performance quantities: memory requirements (MiB), CPU time (s), and various **wallclock times** (s), which measure the real time that has passed: the overall WC time, called **all WC time** (**WC time** for short), the WC time of the actual LazyOTF algorithm without setup and cleanup overheads (for JTorX, TreeSolver, and the network in the parallel setting), called **test WC time**, and the WC time of the SUT $\mathbb{S}$ (i.e., test execution), called **WC time on** $\mathbb{S}$. For most experiments, measurements will depend on $MAXPRT \in [1, \ldots, 10]$.

Fixed Parameters

Since $\mathcal{S}_{LC}$ exhibits the relevant aspects of an LTS and can make TOs arbitrarily difficult by increasing *MAXPRT*, this case study always uses $\mathcal{S}_{LC}$ and no other specification. For the expansion to LTSs, $[[\mathcal{S}_{\mathcal{V}_{LC}^{d}}]]$ is used (i.e., with the default variable initialization).

This case study only employs the implementation relation *ioco* (cf. Sec. 8.5) since it is a standard and simple to implement. But other implementation relation could be used instead (cf. Subsec. 11.2.2).

Most implemented optimizations are activated throughout the case study: the lazy traversal sub-phase optimization and the eager micro-traversal sub-phase optimization. The latter always uses $b_{future} = b_{exec} = 1$, which is sufficient for the cleanly designed

reactive $\mathcal{S}_{LC}$. For consistency, all described experiments (except in Subsec. 14.3.9) were conducted without τ-closure optimization, which was implemented only after several experiments had already been conducted. Implementing and experimenting with more dynamic information for bound heuristics and with quantifying nondeterminism is future work, for the reasons given in Sec. 12.4.

No user interaction or GUI was used in the experiments since the large number of iterations for statistically accurate results (see below) require fully automatic MBT. Therefore, all experiments are performed in text mode (cf. Subsec. 13.3.5), which also allows recomputation [Gent, 2013; Arabas et al., 2014; URL:recomputationHP].

No cycle warnings were active (except for the preliminary experiments on the cycle warning feature, cf. Subsec. 14.3.9).

This section always uses $\underline{aggPath2Ws}(\cdot) = max$ to avoid suppressing a TO (cf. Subsec. 12.3.7).

The following software was used:

- JTorX version 1.10.0-beta8 with LazyOTF support [URL:JTorXwiki] and various small extensions [Kutzner, 2014];
- Oracle® Java™ JDK version 1.7.0_45, Java™ SE Runtime Environment version 1.7.0_45-b18), Java HotSpot™ 64-Bit Server VM version 24.45-b08, mixed mode;
- Linux kernel 2.6.27.7-9-default;
- Hazelcast [URL:Hazel] version 3.1.2 (and version 3.4.2);
- License Central [URL:LC] version 2.0a and version 2.01.

All experiments were performed on the Acamar computing cluster from the research group "Verification Meets Algorithm Engineering" [URL:Verialg] at the Karlsruhe Institute of Technology, on 17 nodes, each containing two Intel Xeon E5430 CPUs with four cores, 2.66 GHz, 12 MiB second-level cache, 64bit architecture, Enhanced Intel SpeedStep and Demand Based Switching, no hyper-threading, and 32 GiB of RAM. The License Central web services were hosted on a separate machine within Oracle VM VirtualBox [URL:VirtualBox].

For evaluation and for debugging, we accumulate as much information as possible during our experiments (cf. Subsec. 13.2.4): storing communication between JTorX and LazyOTF and verbose logging are active as default. Samples with storing communication deactivated show for most of our experiments that performance is influenced little by storing communication, as expected since our experiments are CPU-bound. However, for long running experiments with large bounds, i.e., high amount of communication in each traversalSubphase, storing communication in memory sometimes caused a strong increase in memory, up to out-of-memory (oom) and garbage collection (gc) exceptions, and increased CPU- and WC-time due to the gc algorithm (cf. Appendix B.1.2). In these cases, we deactivated storing communication between JTorX and LazyOTF. For evaluation of all aspects (see plots below) and for debugging, we use verbose logging (but low-level logging on a low verbosity level). We write the log files in experiment archives onto a high-performance hardware raid, which never caused disk I/O contention for our experiments (cf. Appendix B.1.2), and allows convenient access to all log files, even for our parallel experiments and when a node with an experiment is not accessible.

For performance evaluations, cluster nodes were used exclusively, i.e. no other jobs could be scheduled on such nodes while the experiments ran. For memory measurements, the swap space was set to 0 bytes and the maximum Java heap size to 16 GiB (using the option `-Xmx`) when required.

Statistics

LazyOTF and JTorX are randomized algorithms [Motwani and Raghavan, 1995; Sijtema et al., 2014]:

- the simulation of the SUT contains randomization due to uncontrollable nondeterminism;
- for test case generation, JTorX performs a random walk on $\mathcal{S}_{LC}$;
- for test case generation, LazyOTF chooses a TC randomly amongst the heaviest (cf. Subsec. 12.3.4).

To measure a probabilistic value x in our experiments, we perform lightweight statistical tests: We measure

- the **sample mean** (also called **mean value** or **expected value**) $\boldsymbol{\overline{x}} := 1/n \cdot \Sigma_{i=1}^{n} x_i$ of x, taken over n samples $(x_i)_{i \in [1,\ldots,n]}$ (i.e., independent observations), so $\boldsymbol{n}$ is the **sample size**;
- the **maximal value** $\boldsymbol{max(x)} := max_{i=1}^{n} x_i$ and the **minimal value** $\boldsymbol{min(x)} := min_{i=1}^{n} x_i$;
- the **median** value $\boldsymbol{median(x)} := (sorted((x_i)_i)_{\lfloor (n+1)/2 \rfloor} + sorted((x_i)_i)_{\lceil (n+1)/2 \rceil})/2$;
- the **corrected sample standard deviation** $\boldsymbol{\sigma(x)} := \sqrt{1/(n-1) \cdot \Sigma_{i=1}^{n} (x_i - \overline{x})^2}$ of x;
- the **standard error of mean** $\boldsymbol{SEM(x)} := \sigma(x)/\sqrt{n}$;
- **relative standard error of means** $\boldsymbol{RSEM(x)} := SEM(x)/\overline{x}$.

We will usually executed our LazyOTF experiments between 90 and 4000 times to push the RSEM of t_{curr}^{max} to less than 5% (cf. Subsec. B.1.5). Therefore, we can safely transfer our observations from the samples to the real population (cf. Subsec. 14.3.10). For some measurements at the border of feasibility, some samples caused out-of-memory (oom) or timeouts (cf. Subsec. 14.3.10). Since this influences our statistics, we plot the point for those measurements in **parantheses**.

14.3.4. Comparing MBT Approaches

This subsection compares on-the-fly MBT via JTorX and LazyOTF for all TOs defined in Subsec. 14.3.3. For fair comparison, we implemented for each set of TOs a corresponding exit criteria to JTorX. Many experiments in this subsection have been executed by Felix Kutzner [Kutzner, 2014].

Note. Experiments with offline MBT are not necessary since they are not feasible for this case study: the variables in the STSs quickly cause state space explosion during traversal. If we restrict the expansion of the STS, give a bound b on the depth of the generated TCs (cf. Subsec. 10.2.5), and apply on-the-fly MC algorithms for traversal, then the presence of uncontrollable nondeterminism still causes state space explosion and test case explosion; the exponential runtime of a LazyOTF phase plotted against b, shown in Subfig. 14.5a, can be transferred to offline MBT, so offline MBT is only feasible for small depths b.

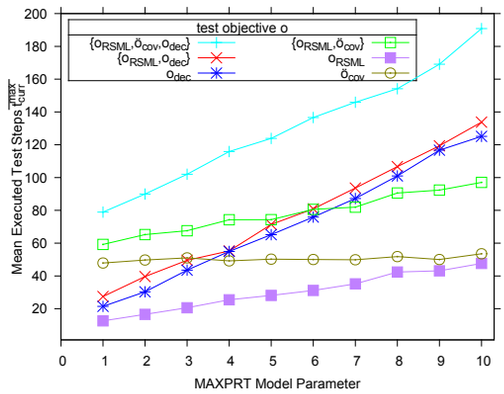
For the experiments in this subsection, we additionally fix the following parameters:

- the SUT $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$;
- the default $\underline{aggWTCs}_{PInputDefault}$;

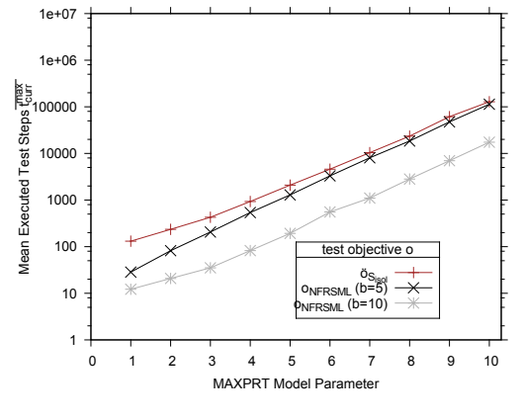
- the default bound heuristics: a fixed bound of 5 (and 10 as comparison for o_{NFRSML});
- the default $P = 1$, i.e., only sequential LazyOTF.

Therefore, the following parameters are varied: $\ddot{o}$, *MAXPRT*, using both LazyOTF and JTorX.

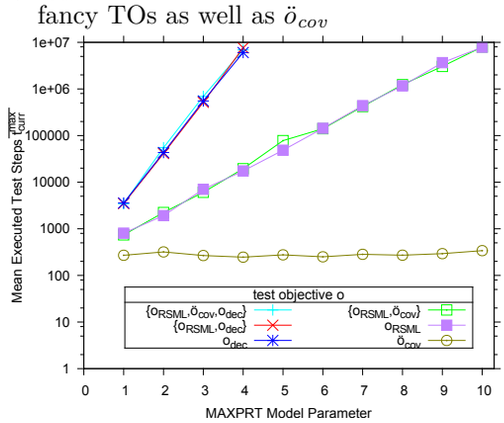
Fig. 14.3 depicts $\overline{t_{curr}^{max}}$, the expected value for t_{curr}^{max}, up to 10^7, plotted against *MAXPRT*. Subfig. 14.3a, resp. Subfig. 14.3b, shows all our fancy, resp. nonfancy, TOs for LazyOTF, Subfig. 14.3c, resp. Subfig. 14.3d, shows all our fancy, resp. nonfancy, TOs for JTorX.



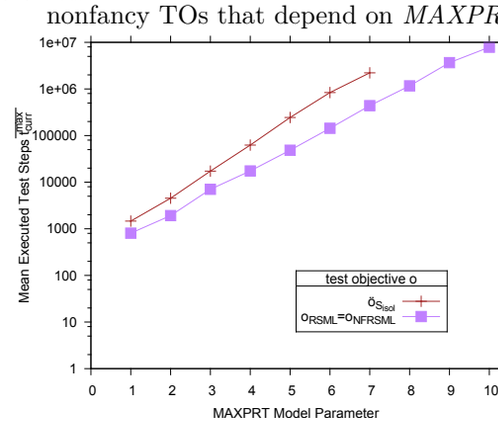
(a) $\overline{t_{curr}^{max}}$ that LazyOTF requires to discharge fancy TOs as well as $\ddot{o}_{cov}$



(b) $\overline{t_{curr}^{max}}$ that LazyOTF requires to discharge nonfancy TOs that depend on *MAXPRT*



(c) $\overline{t_{curr}^{max}}$ that JTorX requires to achieve fancy TOs as well as $\ddot{o}_{cov}$



(d) $\overline{t_{curr}^{max}}$ that JTorX requires to achieve nonfancy TOs that depend on *MAXPRT*

Figure 14.3.: Comparing the meaningfulness of the test cases generated by LazyOTF (Subfig. 14.3a and Subfig. 14.3b) and JTorX (Subfig. 14.3c and Subfig. 14.3d) for all our TOs, where Subfig. 14.3a is a lin-lin plot, all others are log-lin plots

JTorX requires roughly ten times more $\overline{t_{curr}^{max}}$ for $\ddot{o}_{cov}$ compared to LazyOTF; for all others TOs, JTorX's $\overline{t_{curr}^{max}}$ rises exponentially in *MAXPRT* and is multiple orders of magnitude larger than for LazyOTF. For instance, o_{RSML} is difficult for a random walk because the probability of reaching `showMoreLicenses` from `initLocation` with $n \in \mathbb{N}$ steps quickly decreases with increasing *MAXPRT* as some transitions are selected with

probability smaller 1. For fancy TOs, LazyOTF's $\overline{t_{curr}^{max}}$ rises linearly in $MAXPRT$, for nonfancy TOs exponentially. So for JTorX, all TOs except $\ddot{o}_{cov}$ confirm the theoretical analysis that for JTorX you have to expect an exponentially higher t_{curr}^{max} than the minimum number of required test steps for discharging the given TOs, i.e., for reaching some desired feature or coverage criterion via a shortest path (cf. Subsec. 10.2.5).

For LazyOTF with fancy TOs, the measurements also confirm the theoretical analysis (cf. Subsec. 11.3.3) that LazyOTF has strong guidance, i.e., requires t_{curr}^{max} that is linear in the minimum number of required test steps. For LazyOTF with nonfancy TOs, $\overline{t_{curr}^{max}}$ is exponential in $MAXPRT$, but at least one order of magnitude smaller than for JTorX, and the factor increases with $MAXPRT$. LazyOTF achieves this despite using random guidance when far away from TOs, with the help of strong, directed guidance in the final phase, where weak guidance that traverses transitions that nullify long paths traversed towards a TO would hurt the most. Since the length of those paths increases with rising $MAXPRT$, LazyOTF's advantage over JTorX increases with rising $MAXPRT$.

LazyOTF has synergetic TOs (cf. Subsec. 11.2.4), as Subfig. 14.3a shows: $\{o_{RSML}, o_{dec}\}$ is very synergetic since $\overline{t_{curr}^{max}(\{o_{RSML}, o_{dec}\})} << \overline{t_{curr}^{max}(o_{RSML})} + \overline{t_{curr}^{max}(o_{dec})}$ with $\overline{t_{curr}^{max}(\{o_{RSML}, o_{dec}\})} \in \overline{t_{curr}^{max}(o_{dec})} + O(1)$, but $\overline{t_{curr}^{max}(o_{RSML})} \notin O(1)$. The synergy is achieved since LazyOTF detects that both o_{RSML} and o_{dec} need to increase `licenseCount` multiple times at the beginning, and then o_{RSML} can cheaply be discharged before performing the decreases of `licenseCount` for o_{dec} [Kutzner, 2014]. $\{o_{RSML}, \ddot{o}_{cov}\}$ and $\{\{o_{RSML}, o_{dec}\}, \ddot{o}_{cov}\}$ are slightly synergetic since $\overline{t_{curr}^{max}(\{o_{RSML}, \ddot{o}_{cov}\})} < \overline{t_{curr}^{max}(o_{RSML})} + \overline{t_{curr}^{max}(\ddot{o}_{cov})}$ and $\overline{t_{curr}^{max}(\{\{o_{RSML}, o_{dec}\}, \ddot{o}_{cov}\})} < \overline{t_{curr}^{max}(\{o_{RSML}, o_{dec}\})} + \overline{t_{curr}^{max}(\ddot{o}_{cov})}$. However, $\overline{t_{curr}^{max}(\ddot{o}_{cov})}$ is in $O(1)$ anyways.

Notes. JTorX has even higher synergy, as Subfig. 14.3c shows. This is because the random walk to reach a hard TO requires a huge number of test steps, so it is likely that the random walk meanwhile also achieved weaker TOs [Duran and Ntafos, 1984; Denise et al., 2008]. Even though LazyOTF achieves our main goal of nearly optimal t_{curr}^{max}, which are orders of magnitude smaller than for JTorX, it still has synergetic TOs, but sometimes only with weak synergy.

We also investigated whether LazyOTF's linear results for fancy experiments scale for higher $MAXPRT$, up to 1024, which yields huge specifications. They do, except for $\ddot{o}$ with $o_{dec} \in \ddot{o}$ since the $\text{fancy}_{nonlinear}$ $\underline{path2W}_{o_{down}}$ was configured as $1000 + 2000/(\overline{lc} + 1)$, which yields strong guidance only for small average `licenseCount`; for $\overline{lc} \in [800, \ldots, 1333]$, it yields the same weight, which is why our experiments had a timeout for $MAXPRT$=1024. This shows that $\text{fancy}_{nonlinear}$ does not scale well when based on variable evaluations. But o_{dec} could easily be fixed by changing 2000 to a very large value or to using a fancy_{linear} TO instead. All other fancy TOs are slightly better for $MAXPRT$=1024 than the expected t_{curr}^{max} by extrapolating Subfig. 14.3a:

- for o_{RSML}: $\overline{t_{curr}^{max}} = 4092 < 5004 = MAXPRT * 4 + 8$;
- for $\ddot{o}_{cov}$: $\overline{t_{curr}^{max}} = 53 \approx 50$;
- for $\{o_{RSML}, \ddot{o}_{cov}\}$: $\overline{t_{curr}^{max}} = 4207 < 5050 = MAXPRT * 4 + 54$.

Even though meaningfulness is more important than time complexities (cf. Subsec. 11.3.3), we also compare our measurements to the overall worst case time complexities for t_{curr} test steps of both on-the-fly MBT and LazyOTF: Having the fixed

specification $\mathcal{S}_{LC}$ with small $branch_{\mathcal{S}_{\rightarrow^*}}$ and $branch_{\mathcal{S}_{det}}$ (cf. Appendix B.1.1), the theoretical worst case time complexity of both JTorX and LazyOTF for typical settings are in $O(t_{curr} \cdot |S_{\rightarrow^*}|)$ (cf. Chapter 10 and Chapter 11). For our measurements in Fig. 14.3, the average CPU time per test step approaches a constant for increasing $MAXPRT$ (cf. Appendix B.1.3). So up to $MAXPRT$=10, the worst case time complexities for t_{curr} test steps seem to be in $O(t_{curr})$, the factor $O(|S_{\rightarrow^*}|)$ for superstates and τ-closure in $O(1)$. Depending on the TO, however, the CPU time per test steps remains constant (for $\ddot{o}_{cov}$) or increases by a factor of up to 4 (for o_{RSML}) for our samples with $MAXPRT$=1024. Hence $O(|S_{\rightarrow^*}|)$ only causes a small factor, which was expected since the degree of nondeterminism is limited for our TOs: for fancy TOs due to strong guidance, for $\ddot{o}_{S_{isol}}$ since nondeterminism is fined, for o_{NFRSML} due to inducingness of `removeExpiredRequested`. So our performance measurements conform to the worst case time complexities. However, the measurements are rough due to the environment (hardware, operating system, JVM, cf. Subsec. 14.3.6 and Appendix B.1.2). This shows particularly in the memory measurements, which do not reflect the exact memory requirements due to the JVM's conservative memory allocation algorithm (cf. Subsec. 14.3.6).

14.3.5. Comparing $\underline{aggWTCs}$

This subsection compares the meaningfulness of test cases generated by LazyOTF using our pre-built $\underline{aggWTCs}$ (cf. Subsec. 14.3.3).

For the experiments in this subsection, we additionally fix the following parameters:

- the SUT $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$;
- the default bound heuristics: a fixed bound of 5 (and 10 as comparison for o_{NFRSML});
- the default $P = 1$, i.e., only sequential LazyOTF.

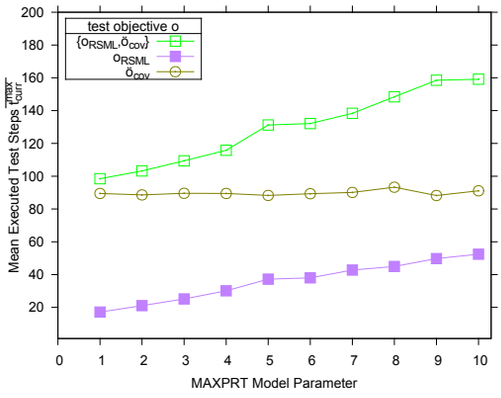
Therefore, the following parameters are varied:

- $MAXPRT$;
- $\ddot{o}$, using all TOs defined in Subsec. 14.3.3, excluding o_{dec}, which was programmatically designed only for $\underline{aggWTCs}_{PInputDefault}$;
- the pre-built $\underline{aggWTCs}$: The measurements for PInputDefaultare already shown in Fig. 14.3, so here we measure Max and MaxMax. These three $\underline{aggWTCs}$ cover all practical classes (cf. Subsec. 12.3.6).

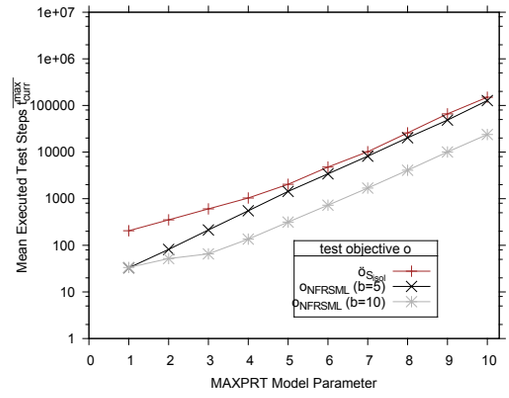
Fig. 14.4 depicts $\overline{t_{curr}^{max}}$, up to 10^7, plotted against $MAXPRT$. Subfig. 14.4a, resp. Subfig. 14.4b, shows all fancy, resp. nonfancy, TOs for Max, Subfig. 14.4c, resp. Subfig. 14.4d, shows all fancy, resp. nonfancy, TOs for MaxMax.

The measurements confirm the theoretical analysis of $\underline{aggWTCs}$ (cf. Subsec. 12.3.6): The balanced $\underline{aggWTCs}_{PInputDefault}$ yields the lowest $\overline{t_{curr}^{max}}$: considerably lower for fancy TOs, and lower by a small factor for nonfancy TOs. It is interesting that MaxMax has slightly lower $\overline{t_{curr}^{max}}$ than Max for o_{RSML} and $\{o_{RSML}, \ddot{o}_{cov}\}$, which can be explained by $\underline{path2W}_{o_{RSML}}$ steadily monotonically increasing towards o_{RSML}. For all other TOs, Max has significant lower $\overline{t_{curr}^{max}}$ than MaxMax, since Max takes all WTC nodes on a path into account, not just the heaviest like MaxMax. We plotted the measurements for MaxMax with o_{NFRSML} and $MAXPRT$=10 in parantheses since several oom occurred (cf. Subsec. 14.3.10).

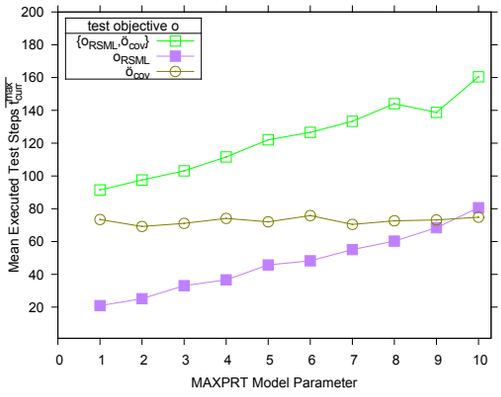
We expected Max and MaxMax to be faster computations than PInputDefault, but the difference in CPU time per test step was usually small (at most 25% difference,



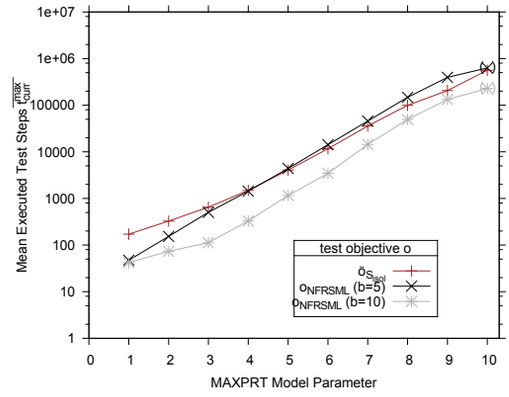
(a) $\overline{t_{curr}^{max}}$ for $\underline{aggWTCs}_{Max}$ to discharge fancy TOs



(b) $\overline{t_{curr}^{max}}$ for $\underline{aggWTCs}_{Max}$ to discharge non-fancy TOs



(c) $\overline{t_{curr}^{max}}$ for $\underline{aggWTCs}_{MaxMax}$ to discharge fancy TOs



(d) $\overline{t_{curr}^{max}}$ for $\underline{aggWTCs}_{MaxMax}$ to discharge nonfancy TOs

Figure 14.4.: Comparing the meaningfulness of the test cases generated by $\underline{aggWTCs}_{Max}$ (Subfig. 14.4a and Subfig. 14.4b) and $\underline{aggWTCs}_{MaxMax}$ (Subfig. 14.4c and Subfig. 14.4d) for our TOs (excluding o_{dec})

usually much less). Sometimes Max was faster, sometimes MaxMax. The insignificance of the aggregation computations on the runtime can be explained theoretically since the aggregation computation does not influence genWTS's worst case time complexity (cf. Subsec. 12.3.4). Due to the small difference in the CPU time per test step, most performance measurements conform to the worst case time complexities, as for PInputDefault. But for $\{o_{RSML}, \ddot{o}_{cov}\}$, the CPU time per test step is slightly increasing for both Max and MaxMax, which is probably due to larger degrees of nondeterminism: Max and MaxMax are not balanced aggregations, so only the most meaningful resolution of nondeterminism on output determines the weight. Hence nondeterminism is not fined but can increase meaningfulness, i.e., larger degrees of nondeterminism are rewarded.

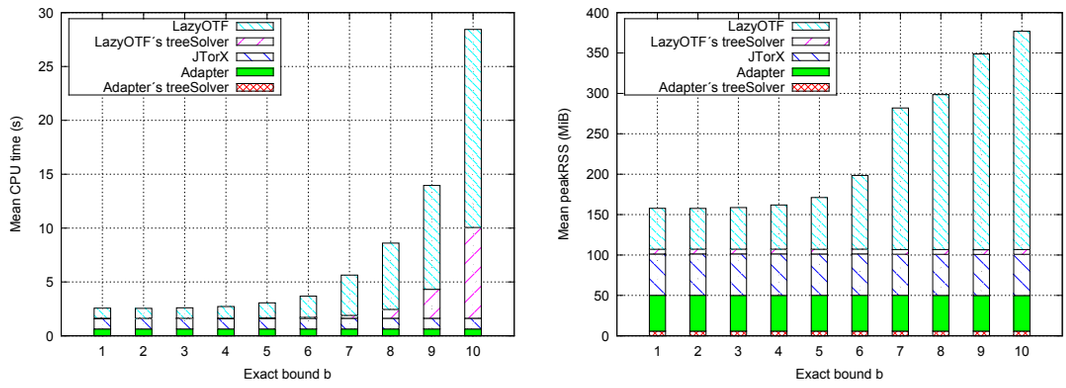
14.3.6. Comparing Bound Heuristics

This subsection compares the performance and meaningfulness of test case generation by LazyOTF depending on the bound heuristics. This shows the effectiveness of the bound heuristics since it must balance the trade-off between meaningfulness and performance for the set of active TOs (cf. Subsec. 12.4.4).

For the first experiments in this subsection, we show the measurements from [Kutzner, 2014] on the resource requirements for one traversalSubphase depending on a constant bound b. They fix all parameters but b:

- the SUT $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$;
- $MAXPRT = 10$;
- $P = 1$, i.e., only sequential LazyOTF;
- $\underline{aggWTCs} = \underline{aggWTCs}_{PInputDefault}$;
- $\ddot{o} = \varnothing$, such that full TCs of depth b are generated, i.e., without early pruning by inducing states.

Fig. 14.5 depicts the required CPU time (Subfig. 14.5a) and memory (Subfig. 14.5b), plottet against bound b.



(a) CPU time required for startup and first traversalSubphase (using no TOs)

(b) Memory required for startup and first traversalSubphase (using no TOs)

Figure 14.5.: Resource consumption for one traversalSubphase depending on a constant bound b

For one traversalSubphase, the worst case time, space, and test case complexities are exponential in $b = b_{max}$: with fixed $branch_{\mathcal{S}_{\rightarrow^*}}$, $branch_{\mathcal{S}_{det}}$, and $\ddot{o} = \varnothing$, both complexities are in $O(branch^b_{\mathcal{S}_{det}} \cdot |S_{\rightarrow^*}|)$ (cf. Subsec. 11.3.3). The measured CPU time confirms the worst case time complexity since it is exponential in b. The measured memory seems to be polynomial in b, i.e., too low to confirm the exponential worst case space complexity. But the measured memory requirement is probably distorted by the JVM's conservative memory allocation algorithm (cf. Subsec. 14.3.10) for smaller memory requirements, i.e., for smaller b. Since we have no TOs and no end states in $\mathcal{S}_{LC}$, complete test trees are constructed per traversalSubphase. If all verdicts are given explicitly, the test case complexity per traversalSubphase is $(L_U + 1)^b \cdot b$. If we allow implicit verdicts, the test case complexity per traversalSubphase is at least $2^{b/2}$ because at least every other state in $\mathcal{S}_{LC}$ exhibits nondeterminism on output.

As described in Subsec. 11.3.3, the number of required test steps t_{curr}^{max} to discharge all TOs is more critical than the above complexities. Thus Fig. 14.6 illustrates the overall effect that the bound heuristics has on meaningfulness and performance, i.e., the trade-off between $\overline{t_{curr}^{max}}$ and the expected CPU time. Many bounds and parameters for the bound heuristics are possible (cf. Subsec. 13.2.3), especially when also considering dynamic bound heuristics, which continuously balance this trade-off. So a huge number of experiments would be required if we considered all possible parameter combinations. Therefore, we fix the following parameters:

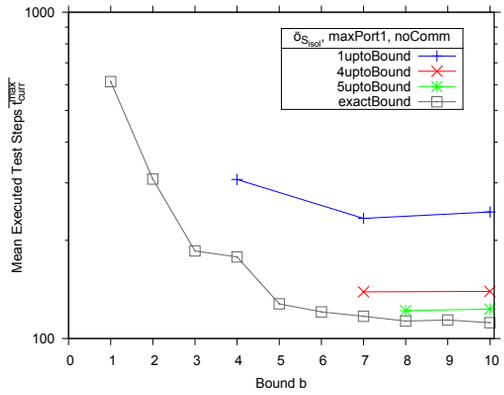
- $P = 1$, i.e., only sequential LazyOTF;
- $\underline{aggWTCs} = \underline{aggWTCs}_{PInputDefault}$;
- since storing communication caused an increase in CPU time for these experiments (cf. Appendix B.1.2), or even oom exceptions, we disabled it;
- $\ddot{o} = \ddot{o}_{S_{isol}}$, so this subsection focuses on one nonfancy TO. This is sufficient to show that the parameters of the bound heuristics strongly influence meaningfulness and performance, but are hard to predict;
- the sawtooth phase heuristic with p_+ being $x \mapsto x + 3$, which is likely a good heuristic for $\ddot{o}_{S_{isol}}$ since TGs of different TOs are clustered.

Having fixed the sawtooth phase heuristic, we vary $b_{max} \in [1, \ldots, 10]$. To also vary b_{min} for dynamic bound heuristics, we define the following: ***i*uptoBound** for $i \in \{1, 4, 5\}$, which uses $b_{min} = i$ and only $b_{max} \in [i + 3, i + 6, 10]$ (never more than 10); the bound heuristics **exactBound** use $b_{min} = b_{max}$ and are thus not dynamic. Furthermore, we choose $MAXPRT \in \{1, 10\}$.

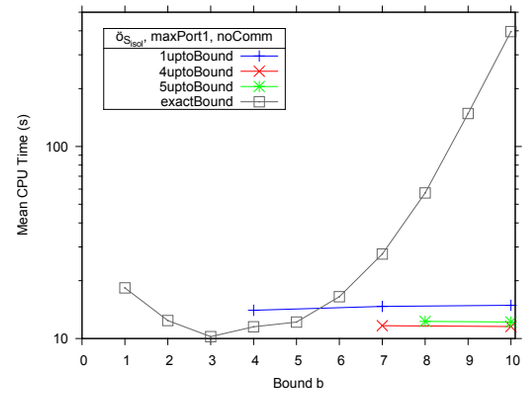
Fig. 14.6 depicts $\overline{t_{curr}^{max}}$ as well as the mean CPU time, plotted against the maximal bound b_{max}; Subfig. 14.6a and Subfig. 14.6b for $MAXPRT = 1$, Subfig. 14.6c and Subfig. 14.6d for $MAXPRT = 10$.

For $MAXPRT$=1, $\overline{t_{curr}^{max}}$ decreases exponentially until about $b = 5$, for larger b only weakly. The dynamic bound heuristics have $\overline{t_{curr}^{max}}$ that are only slightly higher than exactBound (for higher b_{min}) or about twice as high (for $b_{min} = 1$). The mean CPU time starts to increase exponentially at about $b = 5$ for exactBound, but remains low and constant for dynamic bound heuristics. So for $MAXPRT$=1, our dynamic bound heuristics have a good trade-off between $\overline{t_{curr}^{max}}$ and the mean CPU time, especially for higher b_{min}. Unfortunately, the situation is different for $MAXPRT$=10: $\overline{t_{curr}^{max}}$ decreases exponentially for all considered $b \in [1, \ldots, 10]$. Again, the dynamic bound heuristics have $\overline{t_{curr}^{max}}$ that are only slightly higher than exactBound (for higher b_{min}), but about four times as high (for $b_{min} = 1$). Again, the mean CPU time starts to increase exponentially at about $b = 5$ for exactBound, but unfortunately also for dynamic bound heuristics. Furthermore, the mean CPU time is about twice as high for $b_{min} = 1$ compared to exactBound, a bit higher for $b_{min} = 4$ and slightly lower for $b_{min} = 5$. These measurements for the dynamic bound heuristics are probably due to $\overline{t_{curr}^{max}}$ always decreasing exponentially with b, i.e., dynamically choosing a $b < b_{max}$ has a drastic effect on $\overline{t_{curr}^{max}}$. Consequently, for our experiment with $MAXPRT$=10, our dynamic bound heuristics have a bad trade-off between $\overline{t_{curr}^{max}}$ and the mean CPU time for small b_{min}, for high b_{min} the trade-off is only insignificantly better than for exactBound.

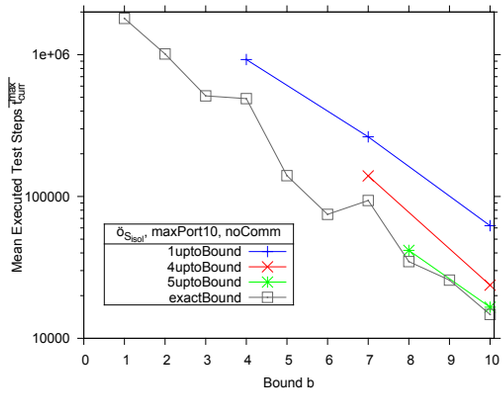
In summary, exactBound with $b = 5$ seems to be a good default because CPU time remains comparably low. But the exact bound for an optimal trade-off between $\overline{t_{curr}^{max}}$



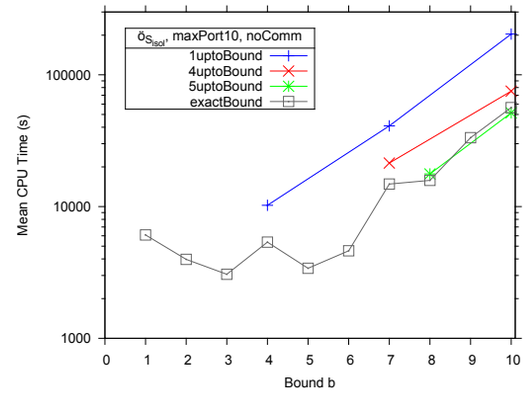
(a) $\overline{t_{curr}^{max}}$ for various bound heuristics with $MAXPRT = 1$



(b) Mean CPU time for various bounds heuristics with $MAXPRT = 1$



(c) $\overline{t_{curr}^{max}}$ for various bound heuristics with $MAXPRT = 10$



(d) Mean CPU time for various bounds heuristics with $MAXPRT = 10$

Figure 14.6.: Comparing the meaningfulness and runtime of various (dynamic) bound heuristics of LazyOTF with $MAXPRT = 1$ (Subfig. 14.6a and Subfig. 14.6b) and $MAXPRT = 10$ (Subfig. 14.6c and Subfig. 14.6d) for $\ddot{o}_{S_{isol}}$

and the mean CPU time is hard to predict. Dynamic bound heuristics may restricts CPU time if b_{min} is not chosen too small, depending on the situation, e.g., whether $\overline{t_{curr}^{max}}$ always decreasing exponentially with b. So our suggestion after these experiments is using exactBound with $b = 5$ or 5uptoBound with $b_{max} = 10$. However, in the case of dynamic bound heuristics being much better than exactBound, i.e., for $MAXPRT$=1, 4uptoBound has a slightly lower mean CPU time than 5uptoBound. Several alternative settings and improvements for dynamic bound heuristics are possible and should be investigated as future work (cf. Subsec. 14.4.3).

14.3.7. Distributed LazyOTF

This subsection investigates how distributed LazyOTF speeds up test generation and test execution. We consider strong scaling (cf. Subsec. 3.5.2) since LazyOTF is CPU

bound: previous experiments showed runtimes of several days (cf. Appendix B.1.5) for hard problems, but seldom oom exceptions (if storing communication was disabled, cf. Appendix B.1.2).

To distribute test execution in an efficient and fault tolerant way, we distribute LazyOTF on different machines in this subsection. We choose up to 16 nodes in our cluster (cf. Subsec. 14.3.3). But preliminary experiments have shown that distributed LazyOTF can also use multi-core multiprocessing (cf. Note 11.12).

We used our default communication via Hazelcast, since it is the most flexible, reliable (cf. Note 14.1), and offers the most features (synchronization, reliable communication, an object-based DSM system, elasticity, easy deployment to cloud services, and a monitor, cf. Subsec. 11.5.2).

Notes 14.1. We conducted the first experiments on distributed LazyOTF with lightweight message passing via broadcasts and multicasts. They had roughly the same results as with Hazelcast. But sometimes the cluster's scheduler (PBS) did not start all instances sufficiently synchronously, so the later instances did not receive all broadcasts or multicasts from earlier instances, i.e., did not detect all externally discharged TOs, in which case there was higher work duplication. For short running experiments (roughly one second), even the most extreme case occurred: the early instances finished the experiment and terminated before the later instances started receiving messages, so the later instances missed all external discharges from the early instances, causing full work duplication for them. Hence we chose Hazelcast as default communication in our implementation and our experiments, and did not investigate lightweight message passing further.

Since we used shell scripts and PBS for all other experiments, we also used them to schedule parallel jobs (cf. Subsec. 11.6.3). In hindsight, we should have used Hazelcast's distributed executor service due to the complexity of parallel scenarios, which is therefore future work.

For the experiments in this subsection, we additionally fix the following parameters:

- the SUT $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$;
- the default $\underline{aggWTCs}_{PInputDefault}$;
- the default bound heuristics: a fixed bound of 5;
- message passing via Hazelcast, version 3.1.2 (and version 3.4.2);
- $MAXPRT = 10$ to have a sufficiently large fixed problem size to consider strong scaling.

Therefore, the following parameters are varied: $P \in [1, \ldots, 16]$ and $\ddot{o}$, using $\ddot{o}_{S_{isol}}$ (as example for a hard problem and a nonfancy TO, and several fancy TOs (as examples for simple problems).

Fig. 14.7 depicts the following quantities for $\ddot{o}_{S_{isol}}$ (cf. Subsec. 14.3.3):

- t_{curr}^{max} to measure meaningfulness, counterexample length, and test execution performance, which is the bottleneck for slow SUTs;
- CPU time to measure how computationally intensive the problem is, which affects costs and WC time;
- all WC time to measure cost and how long the user has to wait when Hazelcast is not running beforehand;
- test WC time as alternative to all WC time when Hazelcast is already running;

- internal discharges to measure redundancy (i.e., multiple discharges) and distribution of work (i.e., of discharges amongst the nodes).

We always take the average over the set of samples n. For each sample, we have P runs, one on each distributed node, and pick the minimal, maximal and mean value:

- *min* is the optimistic estimate (since the shortest test execution might depend on results of the longest test execution);
- *max* is the pessimistic estimate;
- *avg* shows the mean, which takes each measure into account, and is also relevant for the overall costs.

So for instance $\text{min CPU time} = \underset{i\in[1,\ldots,n]}{avg}(\underset{p\in[1,\ldots,P]}{min}(\text{CPU time of experiment number } i \text{ and node } p))$. In total, we have $5 \cdot 3 = 15$ measurements, which also allow insightful comparisons, e.g., between *min* and *max* or between all WC time and test WC time.

Since LazyOTF performs very efficiently for fancy TOs (cf. Subsec. 14.3.4), with about 12 to 22 seconds sequential WC all time, the parallelization overhead (see Appendix B.1.4) outweighs the gains from parallelization, so the fancy TOs exhibit sublinear speedups that drop around $P = 10$ to $P = 14$. Furthermore, measurements have shown that Hazelcast's setup is usually about 4 seconds faster for the node that is elected the master in the Hazelcast cluster. Furthermore, there is a variation of about 1 second among the other instances. Hence the actual distributed LazyOTF algorithm does not start synchronously, but with a time difference of up to 5 seconds. Such delays can distort the speedups for fancy TOs. We did not force the master node to wait since this decreases the overall performance, and such delays are usual for parallel executions [Bader et al., 2000]. Since strong scaling analysis should only consider a sufficiently large problem size [Bader et al., 2000], we do not analyze the fancy TOs and only plot them in Appendix B.1.4.

The measured CPU times and test WC times in Fig. 14.7 show almost linear speedup. *min*, *avg* and *max* test WC times are almost identical, showing that communication of discharges are fast. *avg* all WC times and *max* all WC times show lower but still good speedups. Since testing the SUT for all TOs is finished with the first parallel instance finished, and we immediately store a LazyOTF result (i.e., its experiment archive) on our RAID, *min* all WC time represents the time the user has to wait and is the most relevant all WC time. The difference between *min* and *avg* all WC time, as well as the difference between *avg* and *max* all WC time, are proportional to P and show Hazelcast's overhead since all test WC times are almost identical. But Hazelcast's high overhead (cf. Appendix B.1.4) only occurs if Hazelcast must setup up and clean up the Hazelcast cluster for each MBT run. An important application of distributed MBT is, however, the processing of many MBT tasks on a varying number of processors (cf. Subsec. 11.5.1 and [Nupponen, 2014]). Hazelcast can handle this without repetitive setups and cleanups. In this case, test wc time is the important measure. The measured speedups for TO discharges give insights to how the fastest or luckiest parallel instance (*max* intern discharge) performs compared to the slowest or unluckiest (*min* intern discharge). These two plots are far apart, which shows that the fastest instance is much faster than the slowest, i.e., TOs are not evenly discharged. The fastest instance is the elected master in the Hazelcast cluster, which starts with the actual LazyOTF algorithm up to 5 second earlier than the other instances (see above). Furthermore, the CPU

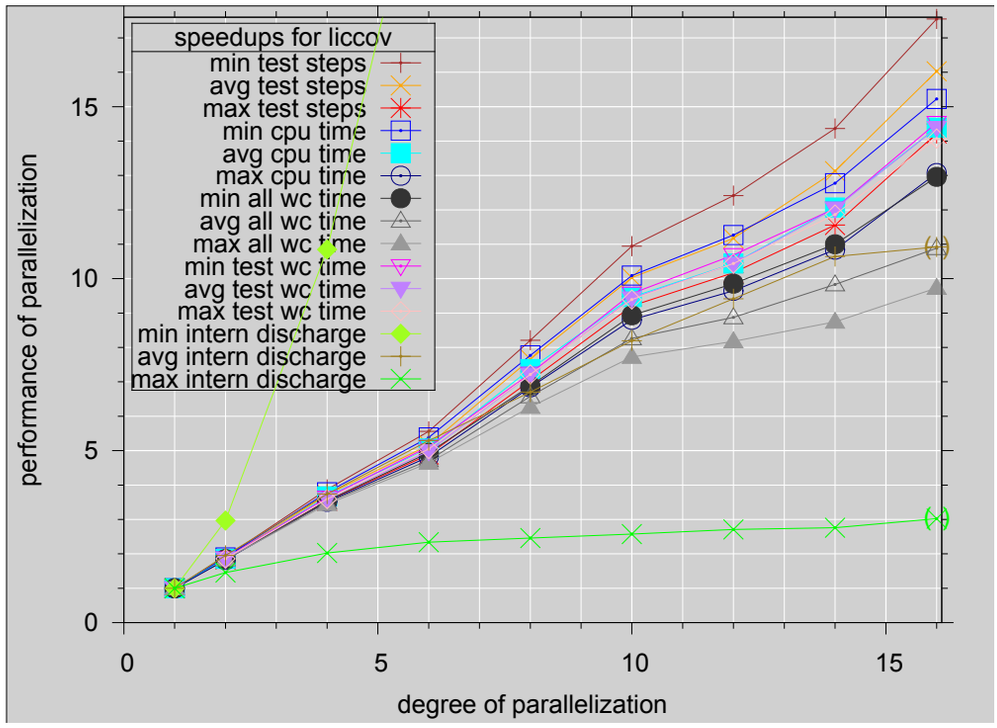


Figure 14.7.: Distributed LazyOTF for $\ddot{o}_{S_{isol}}$

performance of the parallel instances varies (cf. Appendix B.1.2). *avg* intern discharge shows that on average, less than 20% redundant TO discharges occur, very likely the simpler ones that are quickly discharged from the fastest parallel instance. Finally, we consider the measured test steps t^{max}_{curr}: Since there is a large difference between avg t^{max}_{curr} and max t^{max}_{curr}, there also has to be a corresponding difference between avg t^{max}_{curr} and min t^{max}_{curr}. Therefore, the super-linear speedup of min test steps can be explained by the high speedup of avg t^{max}_{curr}. But why does the practically relevant avg t^{max}_{curr} have speedup that is slightly super-linear? More precisely, the speedup of avg t^{max}_{curr} for $P = 16$ is 16.03 and has a factor of 2.1 from $P = 8$ to $P = 16$, so only slightly above linear and hence denoted as **(super-)linear**. Meaningful test execution can achieve super-linear speedup because parallel test execution cannot be simulated efficiently (cf. Note 6.34) on a sequential machine: In the parallel setting, simple TOs are discharged concurrently at the beginning, so that each instance can then execute the SUT directed towards more difficult TOs without being diverted by simple TOs. In the sequential case, simple TOs can divert test generation from more difficult TOs, but these simple TOs would be easily discharged after a restart during a directed test generation towards further difficult TOs. Simulating the concurrent discharges of simple TOs at the beginning and thereafter concurrently moving towards more difficult TOs can generally not be implemented in the sequential case: Running multiple SUTs on a single machine usually differs too much from the real environment (usually with a single SUT), leading to less stable SUTs that might influence each other, to very different performance behavior, and

to less fault-tolerant testing that might break at the first failing TC (cf. Subsec. 11.5.1). Simulating the concurrent discharges on a single machine with a single SUT is usually not possible during test execution since the SUT cannot jump between arbitrary states (without performance penalty).

Note 14.2. We had a bug in our logging that caused inconsistent evaluation of internal discharges. We fixed this bug by changing the Hazelcast version to 3.4.2 and its cleanup. Since this only has a potential effect on max intern discharge and avg intern discharge for P=16, we plotted these values in parantheses and did not repeat the experiment, which required 34 days of accumulated WC time.

Since Conformiq Grid (cf. Subsec. 11.5.1) also offers parallelization, we can compare speedups: For *max* or *avg* all WC time, both tools have a speedup of about 10 on 16 nodes. But it is reasonable to consider *min* all WC time or test WC time for distributed LazyOTF, in which case we have a speedup of 13 to 14.5 on 16 nodes. Furthermore, the faster the sequential algorithm that the speedups are based on, the more significant the scalability analysis is; since our sequential algorithm is usually exponentially faster than offline MBT to achieve the desired TOs, our speedups (being the absolute speedup, not only the relative) have even stronger meaning. These speedups were achieved by including meaningful test execution with (super-)linear speedup, which also allows to only communicate TO discharges (i.e., much lower communication than for transmitting TCs).

14.3.8. Real SUT

This subsection employs LazyOTF on a real, industrial SUT, and is a proof of concept that the aspects investigated on $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$ can be transferred to practical SUTs. For this, we measure LazyOTF's overall WC times and test execution WC times on $\mathbb{S}_{LC}$, and compare the test steps on $\mathbb{S}_{LC}$ with those on $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$.

For scalability analysis and comparisons with former experiments, we show these measurements depending on *MAXPRT*, for both a fancy and a nonfancy TO, and the other parameters as before. Hence we fix the following parameters:

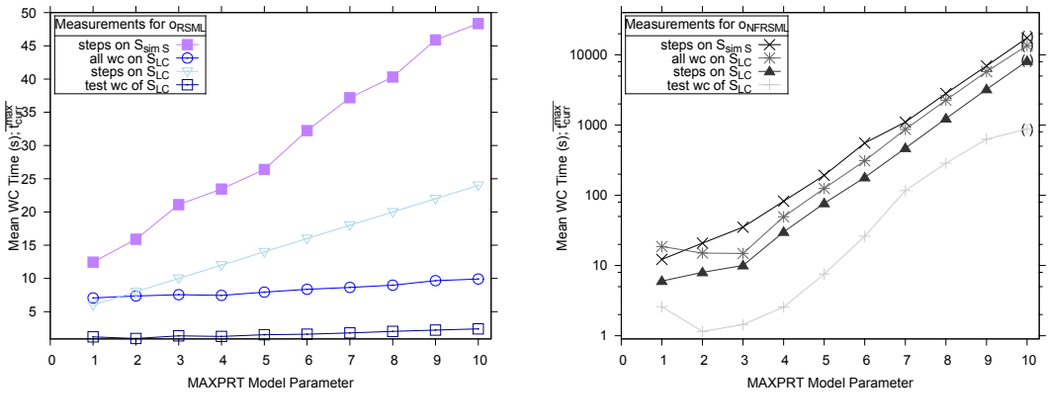
- the SUT $\mathbb{S}_{LC}$: the tests are executed on License Central version 2.0a, preliminary experiments also on version 2.01;
- the default $\underline{aggWTCs}_{PInputDefault}$;
- the default bound heuristics: a fixed bound of 5;
- the default $P = 1$, i.e., only sequential LazyOTF. Besides executing a single LazyOTF in our main experiments, we also executed up to 80 sequential LazyOTF in parallel on a single $\mathbb{S}_{LC}$ to investigate a different kind of parallelism: concurrent accesses on a single SUT.

Therefore, the following parameters are varied: *MAXPRT* and $\ddot{o}$, using the fancy TO o_{RSML} and the nonfancy TO o_{NFRSML} (cf. Subsec. 14.3.3). Due to the simple facade (see below), no coverage level of 100% is achievable, i.e., no $\ddot{o}_{cov}$ and concurrent accesses with o_{dec} would also not terminate.

Since we employ LazyOTF for testing $\mathbb{S}_{LC}$, we apply a general MBT tool that is not aware of web services, and move those technicalities into the test adapter (cf. Subsec. 10.3.4). The actual test execution on $\mathbb{S}_{LC}$ is performed via JTorX and its test

adapters (cf. Sec. 13.3). The main developer of $\mathbb{S}_{LC}$, Stefan Nikolaus, also implemented a simple facade to simplify the test adapter for $\mathcal{S}_{LC}$. The test adapter was mainly implemented by Felix Kutzner. For direct measurements and better comparison, the test adapter is kept as simple as possible. Since this subsection only performs a proof of concept, the facade is very limited, so that only the core functionality of $\mathbb{S}_{LC}$ is tested: the backup functionality is stubbed, and exception handling is omitted. No failures occurred during the experiments that would have been covered by the removed exception handling. Therefore, the runtime measurements in this subsection are a lower estimate for testing the full $\mathbb{S}_{LC}$.

Fig. 14.8 depicts the expected overall WC time and test execution WC times on $\mathbb{S}_{LC}$, furthermore $\overline{t_{curr}^{max}}$ and the corresponding $\overline{t_{curr}^{max}}$ on $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$ for comparison, plotted against $MAXPRT$, for o_{RSML} (cf. Subfig. 14.8a) and for o_{NFRSML} (cf. Subfig. 14.8b).



(a) $\overline{t_{curr}^{max}}$ and WC times to discharge fancy TO o_{RSML}

(b) $\overline{t_{curr}^{max}}$ and WC times to discharge nonfancy TO o_{NFRSML}

Figure 14.8.: One LazyOTF on the real SUT $\mathbb{S}_{LC}$, version 2.0a, with fancy TO o_{RSML} (Subfig. 14.4a) and nonfancy TO o_{NFRSML} (Subfig. 14.4c)

The number of test steps $\overline{t_{curr}^{max}}$ on $\mathbb{S}_{LC}$ is a constant factor f smaller than on $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$, for both o_{RSML} and o_{NFRSML}. Test WC time on our simple $\mathbb{S}_{LC}$ is always about 10% to 30% of all WC time on $\mathbb{S}_{LC}$, for both o_{RSML} and o_{NFRSML}. For o_{RSML}, the WC times increase linearly and very slowly with $MAXPRT$. The values for $MAXPRT = 1$ show that LazyOTF has a significant overhead (for setup, waiting and cleanup). For o_{NFRSML}, the WC times increase exponentially with $MAXPRT$, like $\overline{t_{curr}^{max}}$.

The WC times for $maxprt = 1$ were unexpectedly high for o_{NFRSML}, since these runs have a very low $\overline{t_{curr}^{max}}$. This is probably caused by the frequent restarts and inefficient resets in $\mathbb{S}_{LC}$. But this did not occur for o_{RSML}. Since it was not reproducible on $\mathbb{S}_{LC}$, version 2.01, we investigated no further. For o_{NFRSML} and $MAXPRT = 10$, quiescence errors occurred sporadically in 22 out of overall 2250 cases, all within 3 hours; hence the points for $MAXPRT = 10$ are plotted in parantheses. Quiescence errors occurred where the SUT was by mistake quiescent, even though the timeout was set to 10 seconds, i.e., latency increases significantly. The error logs showed messages similar to Listing 14.1. The reason is either a network error or a rare performance problem of $\mathbb{S}_{LC}$ or a rare failure in the test adapter. Since these errors did not re-occur on $\mathbb{S}_{LC}$, version 2.01, we

did not investigate further and the error is probably not in the test adapter. No other errors occurred. This was expected since $\mathcal{S}_{LC}$ is very abstract and License Central is a well tested and broadly applied product. Otherwise, $\mathbb{S}_{LC}$ scales well, as the test WC times in relation to $\overline{t_{curr}^{max}}$ for increasing $MAXPRT$ show.

```
....                                                                    1
Info: 406 output(out): (Quiescence)                                     2
Verdict: fail                                                           3
stopTestRun(): done myDriver.stopAuto();                                4
...                                                                     5
```

Listing 14.1: Exemplary quiescence error message

The measurements show that LazyOTF is applicable on a real SUT, also in an industrial context. Due to the constant factor f (see above), our conclusions from measurements on $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$ in previously subsections can be transferred to a real SUT. Furthermore, the constant factor f shows that LazyOTF scales well with the degree of nondeterminism on output since $\mathbb{S}_{LC}$ is very simple (e.g., exercises no exceptions), whereas $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$ uses equidistribution over all outgoing transitions in $\mathcal{S}_{det}$ (i.e., a lot of exceptions). Hence the increase in the number of test steps must be at least a constant factor. For on-the-fly MBT, it is higher since more random choices are made that can reduce `licenseCount`. Offline MBT does not know a priori how often exceptions occur, which leads to test case explosion or inconclusive testing.

Finally, we also performed experiments with up to 80 sequential LazyOTF instances running in parallel for concurrent accesses of the single $\mathbb{S}_{LC}$, with a single run for each instance. This caused only insignificant changes in the measurements. This is understandable since o_{RSML} and o_{NFRSML} only increases `licenseCount` and notices when it has increased sufficiently. Hence it does not matter whether the increases are caused by one or multiple instances. It does, however, show that the web services in $\mathbb{S}_{LC}$ handle concurrent accesses well, i.e., License Central scales well in the number of users.

14.3.9. Preliminary Further Experiments

Preliminary experiments investigating the cycle warnings (cf. Subsec. 12.3.7) showed that for a threshold of 10, cycle warnings usually did not occur for fancy TOs (mostly 0 times on average, but 0.8 times on average for $\ddot{o}_{cov}$ with $MAXPRT = 10$), but exponentially in $MAXPRT$ for o_{NFRSML} (0 times on average for $MAXPRT = 1$ and 1593 times on average for $MAXPRT = 10$) and for $\ddot{o}_{S_{isol}}$ (16 times on average for $MAXPRT = 1$ and 123311 times for $MAXPRT = 10$). For a threshold of 1000, cycle warnings usually did not occur for nonfancy TOs, too (mostly 0 times on average, but 3.75 times on average for o_{NFRSML} with $MAXPRT = 10$ and 14 times for S_{isol} with $MAXPRT = 10$). Since we have multiple parameters for these cycle warning experiments, thorough statistics are a lot of work and hence future work. The experiments might yield further interesting results since cycle warnings also help to detect weak guidance heuristics, or mean adversaries, or hot spots in $\mathcal{S}_{det}$ during LazyOTF's traversal and test execution sub-phases, which indicates hot spots in the application (cf. Subsec. 12.3.7).

Preliminary experiments show that an optimized implementation for the τ-closure (cf. Subsec. 13.5.2 and [Kutzner, 2014]) usually only yield small runtime improvements (as expected due to the (almost) constant CPU time per test step independent of $MAXPRT$). Hence further experiments were not conducted. For consistency, all other experiments described in this section were conducted without τ-closure optimization, which was implemented only after many experiments had already been conducted.

14.3.10. Conclusion

Summary

This section has investigated multiple aspects of LazyOTF: its capability compared to OTF, its guidance and bound heuristics, its parallelization, and its ability for testing a real SUT.

Since our experiments are fully automated and all scripts and plots available online [URL:experimentLOTFHP], they can be recomputed. Their validity is considered in the following subsection.

Threats to Validity

Threats to validity are potential risks in the design and execution of empirical studies that may limit the experiments' trustworthiness, i.e., reliable results or their generalization to a larger population [Barros and Dias-Neto, 2011; Wohlin et al., 2012; Creswell, 2013].

Since our case study is empirical research, this section mentions its threats to validity. Since we do not perform applied research, but perform experiments to test our theory, the biggest threats are for invalid relationships, resulting in the following priorities: internal over construct over conclusion over external threats [Wohlin et al., 2012].

Threats to Internal Validity.

We explained all parameters in Subsec. 14.3.3, and partitioned them into a set of parameters V that were varied in some experiment, and a set of parameters that were always fixed: V contains the SUT, bound heuristics, $\underline{aggWTCs}$, the number P of parallel instances, TOs, and $MAXPRT$. The following parameters were always fixed: the specification $\mathcal{S}_{LC}$, the implementation relation $ioco$, no user interaction (i.e., JTorX's text mode), the lazy traversal sub-phase optimization, the eager micro-traversal sub-phase optimization with $b_{future} = b_{exec} = 1$, no cycle warning (except in Subsec. 14.3.9), $\underline{aggPath2Ws}(\cdot) = max$, the software (except for Hazelcast and License Central) and hardware settings, storing communication between JTorX and LazyOTF (except for experiments where this impacted memory), and verbose logging. Furthermore, we listed for each experiment which of the parameters in V were additionally fixed and which one actually varied for that specific experiment.

We thoroughly considered our fixed parameters. However, our environment is complex (including hardware like our cluster, and software like the JVM), especially for our parallel experiments [Bader et al., 2000] environment (including heavy communication and Hazelcast). Consequently, there might still be some technical aspect that we have overlooked but that influenced our results. To investigate whether the hardware in our cluster threatens the validity of our measurements, we additionally investigated the

hardware performance (cf. Appendix B.1.2) and detected a potential influencing parameter: the processor load causes power management to vary the time measurements, but at most 20% and usually much smaller. Hence we set similar loads in our experiments for better comparisons. But the load could not be set precisely since the cluster's job scheduler (Portable Batch System [URL:PBSHP]) sometimes put jobs into idle mode or started some jobs with delay. But we only detected this infrequently, and the measurements usually vary only slightly, so this is only a small threat to internal validity. Another technical influence on performance was Hazelcast's varying parallel overhead for setup and cleanup (cf. Appendix B.1.2). We mitigated this threat to internal validity by discussing it at our parallel experiments and measuring both all WC time and test WC time.

Since our experiments have many attributes, we lowered the risk of invalid inferences by performing many experiments over multiple parameters. We believe there are no further influencing parameters (except for possibly network errors within 3 hours, cf. Subsec. 14.3.8) since we conducted many experiments, each one over multiple iterations spread over time.

A few bugs were found during our experiments, in which case we fixed the bug and restarted the experiment from scratch. τ-closure optimization in STSimulator (cf. Subsec. 13.5.2) was only implemented after many experiments had already been conducted. Hence we performed the remaining experiments without τ-closure optimization, except for the corresponding preliminary experiments (cf. Subsec. 14.3.9).

To get all possible information and measurements, we used verbose logging by default, which we wrote into experiment archives stored onto a high-performance hardware raid (cf. Subsec. 14.3.3). Since this never caused significant performance penalties for our experiments (cf. Appendix B.1.2), it did not threaten measurement accuracy (i.e., construct validity), but had the advantage of full information and measurements, i.e., lower threats to internal validity. Similarly, we stored the communication between JTorX and LazyOTF if this did not cause strong memory increases, which was rarely the case (and memory measurements were distorted anyway, cf. threats to construct validity below).

The strongest threats to internal validity were from experimental mortality (aka experimental attrition): For some measurements at the border of feasibility, some samples did not finish the experiment, i.e., did not terminate due to out-of-memory (oom) or timeouts. Since this can lead to distorted measurements, we dealt with these cases exceptionally: we mentioned them in the text and plotted the corresponding points for the possibly distorted measurements in **parantheses** (cf. Subsec. 14.3.3). In some cases, we adapted settings (e.g., deactivated verbose logging and set the maximum Java heap size to 16 GiB, cf. Subsec. 14.3.3) to avoid mortality; in all these cases, the expected values were achieved, i.e., we had no outliers.

Finally, we were not able to evaluate the internal discharges in a few logs for P=16 (cf. Note 14.2), which caused slight outliers in max intern discharge and avg intern discharge; we consequently plotted them in parantheses in Fig. 14.7. We did not repeat the experiment since these experiments for P=16 required about 34 days of accumulated WC time, since we updated Hazelcast to the newest version, and since intern discharge were only auxiliary values to explain t_{curr}^{max} and time quantities.

In summary, the inferences from our measurements and conclusions on the relationships between our parameters have high validity, i.e., we have mitigated all relevant threats to internal validity.

Threats to Construct Validity.

Our main evaluation metric was t_{curr}^{max} until all TOs are discharged because this metric is established [Nieminen et al., 2011; Anand et al., 2013; Gay et al., 2015] and has predictive validity for meaningfulness and test execution performance (cf. Subsec. 14.3.1). Furthermore, we measured the number of (internal) discharges, memory requirement and multiple time requirements: The CPU time, which is the definitive measure for serial executions, and various wallclock times, which are the definitive measure for parallel executions [Bader et al., 2000]. While all WC time measures the complete WC time elapsed, test WC time excludes the setup and cleanup time to measure the WC time elapsed for the actual distributed LazyOTF algorithm, which should be considered for runtime analysis of parallel algorithms [Bader et al., 2000] (cf. Subsec. 14.3.3). Since these performance measurements are difficult [Beyer et al., 2015], we used the **System Information Gatherer and Reporter** (**SIGAR**) framework, which is a cross-platform, cross-language API that provides accurate operating system and hardware level information, such as CPU and system memory statistics [URL:sigarHP; Reddy and Rajamani, 2014]. It is employed by projects like MySQL, Terracotta, and JBoss.

The only measurement that caused difficulties to relate to theory was the memory requirement, which was distorted by the JVM's conservative memory allocation algorithm (cf. Subsec. 14.3.6). We do not know how to avoid these technical distortions, and whether this is possible at all.

The limitations and practical effects of using LTS models to abstract from the SUT are specified precisely in the interface abstraction and testing hypothesis (cf. Sec. 8.1).

In summary, we established a strong relationship between theory and observation with low threats to construct validity for all metrics but memory requirement.

Threats to Statistical Conclusion Validity.

To derive valid relationship from our experiments, we need to use statistical computations since the SUT is nondeterministic (or probabilistic in case of simulation), and genWTS chooses randomly amongst heaviest weights.

For strong statistical randomness yet fast performance, we integrated the pseudorandom number generator Mersenne Twister [Matsumoto and Nishimura, 1998] into the simulation and into genWTS (option use−mt).

Since measurements (like output, the number of test steps, or runtime) of random algorithms can have very complex non-normal probability distributions with high standard deviation, a high sample size (e.g., $n = 1000$) or powerful non-parameterized statistical tests [Wilcox, 2010; Wohlin et al., 2012] are advisable for hypothesis testing [Arcuri and Briand, 2014], if time permits. Simply comparing average measurements, e.g., in a t-test, might be misleading for small n since this does not give any information on the probability distribution, and for randomized algorithms, normality and equal variance are usually not given. But for a sufficiently high number n of samples, the t-test has just as much statistical power and accuracy as other tests, including non-parameterized tests [Arcuri and Briand, 2014]. Since most of our measured expected values that we compare yield high differences (e.g., between JTorX and LazyOTF, between our heuristics, or between different parallelization degrees), we do compare expected values similar to a t-test. Furthermore, we often have a high number n of samples (cf. Appendix B.1.5). Hence our

lightweight statistical tests via RSEMs (as suggested by KIT's statistical consulting) are sufficient and yield more general results than testing for a few specific hypotheses. If WC time is a bottleneck, performing more experiments with smaller n usually yields more insights than fewer experiments with larger n [Arcuri and Briand, 2014]. Hence we allow smaller numbers n and focus on achieving an RSEM $< 5\%$ for t_{curr}^{max} (for 3 experiments we have an RSEM $< 10\%$ with an accumulated WC time of 40 days, each). Appendix B.1.5 summarizes the RSEM, required sample size, and accumulated WC time for all experiments, which totals to 180000 runs and 14 years of WC time.

Note. A standard in reliable statistics is set by health-related statistics, (e.g., from the U.S. National Center for Health Statistics), which demand an RSEM of at most 30%, and a sample size of at least 30 [Klein et al., 2002].

For our comparisons, we always chose a representative of the best known solution so far as baseline: on-the-fly MBT via JTorX, PInputDefault a *aggWTCs*, exact bounds as bound heuristics, the sequential LazyOTF for distributed LazyOTF, and $\mathbb{S}_{sim\,\mathcal{S}_{LC}}$ as SUT.

For descriptive statistics, we considered *mean*, *min*, *max*, *median*, corrected sample standard deviation, SEM, and $RSEM$ in our experiments (cf. Subsec. 14.3.3). To keep the plots simple, we summarize our measurements by demanding an RSEM <5% and only plotting the average measurements in most plots (not for Fig. 14.7 and several Figures in Appendix B.1). We summarize the RSEM and number n of sample in Appendix B.1.5, from which many of the above statistical numbers can be reconstructed. Furthermore, the detailed statistical measurements for each experiment can be found at [URL:experimentLOTFHP].

Since our statistical computations are slightly distorted by mortality, we mention this in our descriptions and plot the corresponding measurements in parantheses (cf. threats to internal validity).

Since we have no reason to consider extreme measurements to be outliers caused by erroneous settings introduced into our experiments, we did not truncate extremes [Kaner and Kabbani, 2012; Bader et al., 2000].

In summary, we derive statistical relationships with a RSEM $< 5\%$. For this, we often, but not always, measure $n \geq 1000$ samples because performing more experiments with smaller n usually yields more insights.

Threats to External Validity.

To allow the generalization from our samples to larger populations, i.e., to lower the threats to external validity, we carefully chose our settings: the specification, its size and complexity, the experiments and their parameters:

We only used one case study based on the abstract specification $\mathcal{S}_{LC}$ since this already caused 44 experiments with an accumulated WC time of 14 years (cf. Appendix B.1.5) to compare approaches, heuristics, and parallelization degrees. Since $\mathcal{S}_{LC}$ has high abstraction, we did not find any functional bug when testing the industrial SUT $\mathbb{S}_{LC}$ (except for quiescence errors). Hence our experiments only roughly investigated LazyOTF's behavior when it finds a bug, but this was not considered by our experiments anyway, so it does not increase the threats to external validity.

It would have been interesting to conduct experiments with a full benchmark to be able to use different specifications and settings to cover more cases and different complexity, to reveal performance limits and to find and test optimizations. Unfortunately, an established benchmark does not exist yet [Lackner and Schlingloff, 2012]. Hence we started building one [URL:MBTbenchmark] and included $\mathcal{S}_{LC}$, but had to mitigated the threats to external validity differently: We based our specification on an industrial example, which we abstracted in natural ways, in accord with architectural principles (cf. Subsec. 14.3.2). Furthermore, $\mathcal{S}_{LC}$ is of variable size and complexity, and incorporates all relevant aspects of an LTS (cf. enumeration in Subsec. 14.3.2), e.g., nondeterminism, τ-cycles, and irregular underspecification of input. We conducted many experiments, with many parameter variations, e.g., using various TOs for coverage, requirements, and more complex properties than reachability (cf. o_{dec} in Subsec. 14.3.3). Furthermore, they vary in difficulty of discharging and in strength of guidance (fancy vs. nonfancy TOs).

Our experiments of distributed LazyOTF were conducted on one cluster. Since each machine and environment is its own special case for parallel experiments [Bader et al., 2000], the measurements can differ on another parallel machine. But since the parallel overhead was low for the actual distributed LazyOTF, the measured test WC times should differ only slightly.

Our insights are applicable independent of the degree of nondeterminism on output and nondeterminism of the LTS (cf. Subsec. 14.3.8).

The TO o_{dec} was created manually, using two helper TOs and a programmatic *discharge* function, configured and implemented specifically for $\underline{aggWTCs}_p$ for our experiments in Subsec. 14.3.4. For other $\underline{aggWTCs}$, o_{dec} does not describe meaningfulness sufficiently well, such that guidance can become inefficient or even inexhaustive. But o_{dec} could easily be fixed by configuring it with higher values (cf. Subsec. 14.3.4). Furthermore, o_{dec} for our experiments in Subsec. 14.3.4 was sufficient to show that the advantages of LazyOTF can be transferred to TOs that are more complex than reachability properties if they describe meaningfulness sufficiently well by weights; then LazyOTF can handle them as well as reachability TOs.

A benchmark is also often used to compare results between various tools for low threats to external validity. Even though we have no benchmark yet, we did not conduct our experiments with further tools besides JTorX and LazyOTF either, since this is not required to achieve a low threat to external validity: Other MBT tools perform either on-the-fly or offline MBT (cf. Sec. 10.3), and we investigated on-the-fly and offline behavior that is inherent in the approach. Hence our findings are generalizable to the other tools. We tested our theoretical results of on-the-fly MBT via JTorX (cf. Subsec. 14.3.4), of offline MBT indirectly via traversalSubphase (cf. Subsec. 14.3.6).

Our theoretical result of t_{curr}^{max} to reach some goal being exponential in the minimal number of required steps for on-the-fly MBT was based on probabilistic computations of random walks, as described in Example 11.1. We additionally computed such a probabilistic value of a random walk for $\mathcal{S}_{LC}$: the expected value t_{curr}^{max} for OTF to reach o_{RSML}, using the tool PRISM (cf. Subsec. 5.5.4). We got similar results as JTorX (cf. Subsec. 14.3.4), so our measurements are due to the bad guidance of OTF and not caused by some technicality in JTorX.

Notes. There is one use case that has been used as case study for various MBT tools: the conference protocol; however, each tool was investigated with an own specification, with uneven degrees of complexity, aggravating comparisons [Belinfante et al., 2005]. Furthermore, most specifications did not exhibit all relevant aspects of an LTS (see above). Hence we used $\mathcal{S}_{LC}$ instead.

PRISM's results differed from the expected value t_{curr}^{max} we measured in JTorX for large $MAXPRT$, but this is probably due to instable computations in PRISM since variations in the applied numerical method resulted in different expected values t_{curr}^{max}.

The exponential t_{curr}^{max} are particularly severe for specifications $\mathcal{S}$ with complex guards, since these often cause TOs being deep in the state space, i.e., only reachable via long paths, which usually have low probability. Similarly, guards can cause the state space to contain a bottleneck, i.e., a large sub-graph in the state space that is reachable only through relatively few transitions. Such bottlenecks in state spaces are usually traversed only with low probability, too. Bottlenecks also exist in practice [Gay et al., 2015] More generally, there are specifications that have an irregular topology, i.e., for some $b \in \mathbb{N}$, some paths of length b have a much lower probability compared to the other paths of length b [Denise et al., 2008]).

$\mathcal{S}_{LC}$ is an abstract STS that is suitable for JTorX, i.e., the reduction heuristic of STSimulator is sufficient for $[[\mathcal{S}_{\mathcal{V}_{LC}^{d}}]]$. This is not the case in general, i.e., for arbitrarily complex STSs, which is a relevant threat to external validity for STSimulator. For example, more expressive STSs would have to be used to check concrete licenses (cf. Sec. 13.4). But alternatively, the test adapter could keep track of the licenses. Furthermore, this thesis does not focus on STSimulator and arbitrarily complex STSs, but on LazyOTF and on LTSs, for which this threat does not apply. Finally, the proof of concept in Sec. 13.4 shows how to handle complex STSs.

Since the experiments were conducted by the implementers of the LazyOTF tool, they do not reflect the average knowledge of an average tester to set up the heuristics. This threat was addressed by conducting experiments for various test objectives with various complexities in their heuristics setup and providing usable TOs and a provisos framework. So all threats to external validity were mitigated and we can generalize our observed results to larger populations. This can be individually examined since we clearly defined our samples, i.e., our specification, the parameters we set and the parameters we measured.

In conclusion, we have several minor threats to the construct, internal, conclusion and external validity, but were able to mitigate all but distorted memory measurements. Therefore, we can safely transfer our observations (except for memory requirements) from the samples to the real population.

14.4. Conclusion

14.4.1. Summary

Sec. 14.2 presented an application of MBT and LazyOTF in a prominent software development process: agile development with Scrum and XP. For optimal application, MBT

and AD should be integrated tightly. This requires flexible MBT without a BDUF and early feedback, e.g., via LazyOTF and *refines*.

Sec. 14.3 confirmed most theoretical analyses from Chapter 11 and 12: LazyOTF and its heuristics yield exponentially better meaningfulness and performance. Distributed LazyOTF achieves (super-)linear speedup of meaningful test execution and almost linear speedup overall. Unfortunately, the dynamic bound heuristics turned out to improve the trade-off between meaningfulness and CPU time only in some cases compared to a well chosen exact bound; in other cases, dynamic bound heuristics yield only slight improvements or even deterioration of the trade-off. But good exact bounds are not always known in advance. Many alternative settings and improvements for dynamic bound heuristics are possible and future work (see below).

14.4.2. Contributions

We described how MBT and agile development should be integrated for practical software engineering, and how LazyOTF and *refines* can be applied fruitfully for this.

Thorough experiments with 14 years of accumulated WC time were conducted, comparing different MBT approaches, different guidance heuristics settings, different bound heuristics settings, distributed LazyOTF, and finally MBT of a real SUT, an industrial web service. The experiments confirmed all theoretical analyses except for the dynamic bound heuristics: we expected significant improvements for bound heuristics by making them dynamic, too, but the experiments showed improved trade-off between meaningfulness and CPU time only in some cases.

14.4.3. Future

For future work on integrating LazyOTF and *refines* with agile development, an industrial case study is promising. For our LazyOTF experiment, interesting future work is performing further experiments on:

- cycle warnings;
- τ-closure optimization;
- bound heuristics, especially with triangle phase heuristics instead of sawtooth;
- bound heuristics using more dynamic information (cf. Subsec. 12.4.4);
- quantifying nondeterminism (Subsec. 12.4.5).

15. Conclusion

15.1. Summary

This thesis has been concerned with making formal methods more feasible in practice, i.e., faster, more scalable and more usable. The focus was on software, but some techniques can easily be applied to hardware as well (e.g., model-based testing [Utting et al., 2006] or bounded model checking [Biere et al., 1999]). Since automation strongly increases usability, we have focused on model checking and model-based testing. More precisely, the positioning of this thesis is depicted in Fig. 15.1.

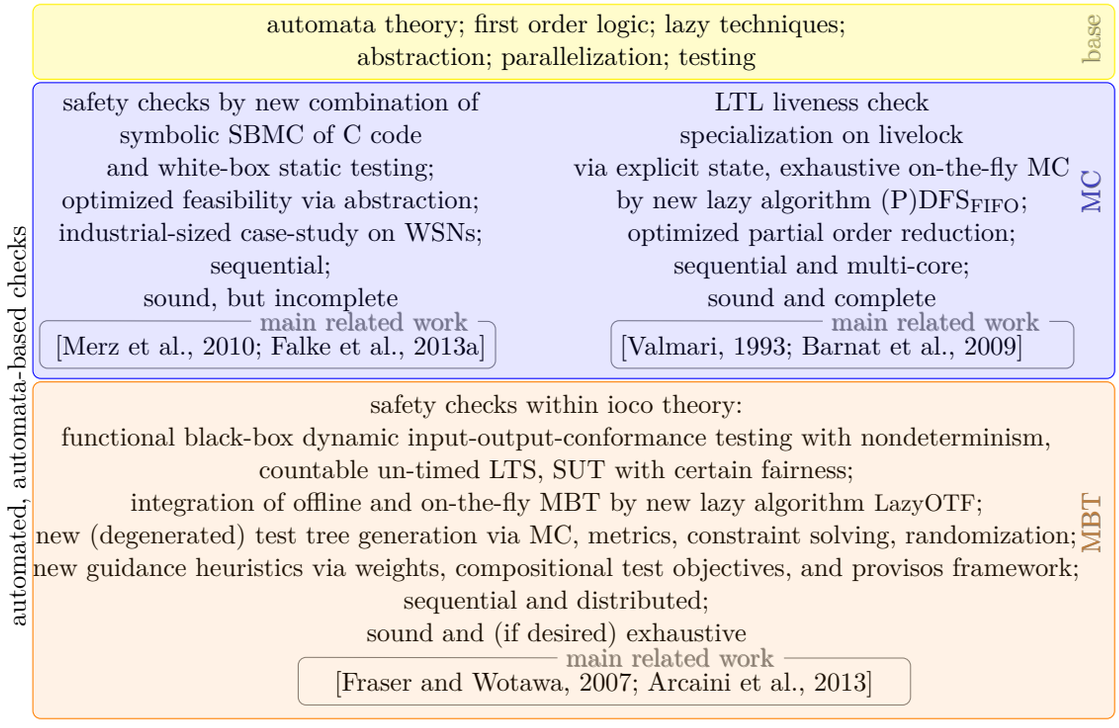


Figure 15.1.: Positioning of this thesis (according to our taxonomies)

15.1.1. Main Addressed Challenges

Sec. 1.2 already described how this thesis achieves increased feasibility for model checking and model-based testing, and listed the contributions of this thesis. Here, we consider the main challenges that the thesis addressed:

The major problem in MC is state space explosion, causing infeasible time and space requirements, especially for liveness properties. For the important liveness property of

livelock freedom, DFS_{FIFO} reduces both time and space requirements by simultaneously exploring and checking the specification in one pass and without a Büchi product, and by improving partial order reduction for DFS_{FIFO}. The time requirement is further reduced by parallelization with almost linear speedup. Usability is improved by simpler and more accurate modeling of progress and shortest counterexamples. Unfortunately, DFS_{FIFO} cannot be generalized to cover all LTL properties.

The major problem in MBT is state space explosion and test case explosion for offline MBT and low meaningfulness for on-the-fly MBT, both causing infeasible time, for offline MBT also infeasible space requirements, especially for SUTs with uncontrollable nondeterminism. LazyOTF reduces both time and space requirements by integrating both on-the-fly and offline MBT: It swaps between state space traversal and test execution phases, enabling strong guidance by traversing sub-graphs of the specification, employing heuristics that make use of dynamic information from previous test execution phases, for instance the SUT's resolution of uncontrollable nondeterminism. The time requirement is further reduced by parallelization with (super-)linear speedup of meaningful test execution and almost linear speedup overall. Usability is improved by higher meaningfulness and reproducibility. Furthermore, flexible heuristics via weights, composable TOs, and a provisos framework offer usable configuration and guarantees for exhaustiveness, coverage, and discharging. Unfortunately, LazyOTF tightly integrates MC algorithms with heuristics to make strong use of the available dynamic information; hence useful off-the-shelf tools (e.g., for MC or SMT solving) cannot be efficiently employed by LazyOTF without adaption; but our provisos framework helps the adaption. Furthermore, many aspects of MBT and possibilities for further heuristics are future work, with MBT of timed automata being a big challenge (cf. next section).

15.1.2. Measurements

For our experiments on four established protocols (cf. Subsec. 6.9.1) and within feasible problem size, DFS_{FIFO}'s space requirements are reduced by a factor of 3 to over 200 compared to related work, its time by a factor of 3.4 to 16. Furthermore, DFS_{FIFO} has over 150 times stronger on-the-flyness than related work. $\text{PDFS}_{\text{FIFO}}$ achieves almost linear parallel speedup, and $\text{PDFS}_{\text{FIFO}}$ with POR can handle 4 to 5 more orders of magnitude compared to related work. For relevant livelocks of our experiments, $\text{PDFS}_{\text{FIFO}}$'s counterexamples are up to 10 times shorter than those of related work. Unfortunately, the improvement of on-the-flyness drops to a factor of only 1.75 for our experiments with $\text{PDFS}_{\text{FIFO}}$ with $P = 48$, but this is mainly due to the strong improvement of the related work (CNDFS) for $P = 48$ compared to $P = 1$.

For our experiments with an accumulated WC time of 14 years (cf. Sec. 14.3 and Appendix B.1.5), LazyOTF's meaningfulness and CPU times are exponentially better than for related work, its space requirements are apparently only polynomial (probably due to JVM's conservative memory allocation for our problem sizes). Furthermore, distributed LazyOTF achieves (super-)linear speedup of meaningful test execution and almost linear speedup overall. Unfortunately, dynamic bound heuristics turned out to improve the trade-off between meaningfulness and CPU time only in some cases. In other cases, dynamic bound heuristics yield only slight improvements or even deterioration of the trade-off. The comparison shows, however, that further investigations and more dynamic information for bound heuristics is relevant future work (see next section).

15.2. Future

Since this thesis covered many fields and MC and MBT span wide domains, we have already summarized many future work in different directions at the end of most chapters. The most fundamental and important ones are:

- for DFS_{FIFO}, the extension from livelock properties to other properties is interesting future work, in spite of the extension to full LTL MC being impossible and livelocks being one of the most relevant liveness properties already;
- for LazyOTF, a clean reimplementation on top of an SMT solver instead of STSimulator can improve the expressiveness of LazyOTF and its specifications due to more powerful symbolic capabilities.

But since the goal of this thesis was improving the feasibility of formal methods in practice, and the main results have been integrated in the popular tools LTSmin and JTorX, the most relevant future work is applying DFS_{FIFO} and LazyOTF in industrial case studies. This is the next step (after sufficiently increasing FM's feasibility) towards the challenge of broad industrial application of FMs (cf. Chapter 1). Due to the trend in parallelization, likely examples for industrial case studies are

- in MC: the verification of livelock freedom for lock-free and wait-free algorithms and data structures, which are quickly growing in number and importance;
- in MBT: the verification of web services, as has already been started in Subsec. 14.3.8.

These case studies should also contain broader empirical experiments on usability, and implement or prioritize among the many smaller future work given throughout this thesis, e.g.,

- integrating our finite $\cup$ infinite trace semantics (e.g., into LTSmin) to improve the specification's expressivity and avoid workarounds;
- integrating further heuristics such as bound heuristics or i_δ caching if LazyOTF's performance must further improve, or a cycling heuristic via NPC checks (which brings DFS_{FIFO} and LazyOTF fully together), or integrating further features of distributed LazyOTF if it is employed heavily due to the boom in cloud computing.

A. Source Code

A.1. Excerpt for SPIN

Listing A.1 implements Example 6.19 by defining process1 and process2, with progress corresponding to the action d_step{process1globalPC = 0;printf("process1 made progress")}; at l.7.

```
byte process1globalPC = 0;                                                    1
                                                                              2
proctype process1()                                                           3
{                                                                             4
  start: process1globalPC = 1;                                                5
  progress:                                                                   6
    d_step{process1globalPC = 0;printf("process1 made progress")};            7
    goto start;                                                               8
}                                                                             9
                                                                              10
proctype process2()                                                           11
{                                                                             12
  start: process1globalPC==1                                                  13
    -> printf("process2 restarted without progress")}; goto start;            14
}                                                                             15
                                                                              16
init {run process1();run process2();}                                        17
```

Listing A.1: PROMELA description for process1 and process2 in Fig. 6.6

A.2. Excerpt for CBMC

Listing A.2 shows the autostart function used for most *STATUS* packets: The first block (l.2 – l.5) sets parameters to configure the network for the chosen scenario. The second block (l.7 – l.20) constructs a *SET* package and lets the node process it. The third block (l.22 – l.30) constructs n *SETAGG* packets to communicate the aggregation tree to the node.

Listing A.3 shows the autostart function used for most ESAWN packets: The first block (l.2 – l.8) sets parameters to configure the network for the chosen scenario. Here p is set nondeterministically. The second block (l.10 – l.21) creates an empty ESAWN packet and then fills it. Here the measurements are set nondeterministically. The third block (l.23 – l.26) sets the ESAWN packet's destination, i.e., the recipient in the aggregation tree. The fourth block (l.28 – l.29) moves the ESAWN packet into the node's queue and lets the node process it.

```
void autostart(void){
  uint16_t n = 5;
  TOS_NODE_ID = 2;
  uint16_t _parent[5] = { 0, 0, 1, 2, 3};
  uint32_t _fake_agg[5] = { 0, 0, 0, 0, 0};

  radio_status_msg_t *_msg = (radio_status_msg_t*)
    EsawnC$SerialPacketStatus$getPayload(
    &EsawnC$tmp_pkt, sizeof(radio_status_msg_t ));
  EsawnC$parents = malloc(sizeof(uint16_t )
    * EsawnC$num_nodes);
  _msg->status_id= STATUS_SET;
  _msg->data.set.num_nodes=n;
  _msg->data.set.k=2;
  _msg->data.set.p=65;
  _msg->data.set.fake_agg=_fake_agg[TOS_NODE_ID];
  EsawnC$SerialAMPacketStatus$setDestination(
    &EsawnC$tmp_pkt, TOS_NODE_ID);
  EsawnC$SerialReceiveStatus$receive(&EsawnC$tmp_pkt,
    _msg, sizeof(radio_status_msg_t ));

  for (i = 1; i < n; i++) {
      _msg->status_id=STATUS_SETAGG;
      _msg->data.setagg.node_id=i;
      EsawnC$SerialAMPacketStatus$setDestination(
        &EsawnC$tmp_pkt, TOS_NODE_ID);
      EsawnC$SerialReceiveStatus$receive(&EsawnC___msg
        ->data.setagg.parent_id=_parent[i];
      d__tmp_pkt, _msg, sizeof(radio_status_msg_t ));
  }
}
```

Listing A.2: Exemplary autostart function for the initialization without any nondeterminism

```
void autostart(void){
  uint16_t i;
  TOS_NODE_ID=2;
  EsawnC$param_k = 2;
  EsawnC$num_nodes = 5;
  EsawnC$param_p = nondet_int();
  EsawnC$parents[5] = { 0, 0, 1, 2, 3};
  EsawnC$fake_agg = _fake_agg[0];

  msg_queue_item mitem;
  radio_esawn_msg_t *_msg = (radio_esawn_msg_t *)
    EsawnC$AMSendEsawn$getPayload(
    &EsawnC$tmp_pkt, sizeof(radio_esawn_msg_t ));
  EsawnC$SerialAMPacketStatus$setDestination(
    &EsawnC$tmp_pkt, TOS_NODE_ID);
  _msg->part[0].from=0;
  _msg->part[0].data.dec.value=nondet_int();
  EsawnC$add_to_relay_list(&_msg->part[0]);
  _msg->part[1].from=1;
  _msg->part[1].data.dec.value=nondet_int();
  EsawnC$add_to_relay_list(&_msg->part[1]);

  mitem.to = TOS_NODE_ID;
  mitem.am_type = AM_RADIO_ESAWN_MSG;
  mitem.msg = EsawnC$tmp_pkt;
  mitem.output = WIFI;

  EsawnC$MessageQueue$enqueue(mitem);
  EsawnC$send_queue$postTask();
}
```

Listing A.3: Exemplary autostart function for the probabilistic concast with measurements and p being nondeterministic

B. Supplemental Figures And Tables

B.1. Figures for the Application of MBT

B.1.1. Abstract STS Specification for CodeMeter License Central

This subsection describes the system specification for $\mathbb{S}_{LC}$, the CodeMeter License Central [URL:LC] web services from WIBU SYSTEM AG for managing and distributing licenses. The specification is an STS that strongly abstracts $\mathbb{S}_{LC}$ and covers its main features: generating, removing, inspecting and backing up licenses.

We call the STS $\boldsymbol{\mathcal{S}_{LC}}$ and depict it in Fig. B.1 (plotted by Felix Kutzner [Kutzner, 2014] using yEd [URL:yEdHP]. $\mathcal{S}_{LC}$ contains a hard-coded parameter: the constant $MAXPRT \in \mathbb{N}_{>0}$ describing the maximal portion for pagination (see next paragraph).

As described in Subsec. 14.3.2, $\mathcal{S}_{LC}$ is designed for web services, using the request-response pattern [W3C, 2001] and fault handling via response messages and additional exceptional behaviors. Managing licenses is specified by the following features, all having appropriate exception handling:

- generateLicense generates a new license, returning its ID;
- removeLicense removes the license with the given ID;
- removeExpired removes all licenses that are currently expired. Since expiration is not modeled by $\mathcal{S}_{LC}$, the abstraction nondeterministically removes a subset of all present licenses;
- removeAll removes all licenses;
- showLicenses lists all present licenses using portions of maximal $MAXPRT \in \mathbb{N}_{>0}$ many licenses, to support pagination and packages for transmission, i.e., successively $MAXPRT$ many licenses can be requested;
- remoteBackup resp. localBackup to perform remote resp. local backups.

Formally, $\mathcal{S}_{LC} = (S_{LC}, \rightarrow_{LC}, L_{LC}, \mathcal{V}_{LC}, \mathcal{I}_{LC}, S^0_{LC}) \in \mathbb{S}_{STS}$, with

- $S_{LC} = \{$`initLocation`, `generateLicenseRequested`, `generateLicenseUnknownException`, `removeLicenseRequested`, `removeLicenseUnknownException`, `removeLicenseLicenseNotFoundException`, `removeAllRequested`, `removeAllUnknownException`, `removeExpiredRequested`, `removeExpiredUnknownException`, `showLicensesRequested`, `showLicensesUnknownException`, `showMoreLicenses`, `backupMenu`, `localBackupRequested`, `localBackupUnknownException`, `localBackupDiskFull`, `remoteBackupRequested`, `remoteBackupUnknownException`, `remoteBackupDiskFull`$\}$, i.e., 20 abstract states;
- $L_{LC} = L_I \dot\cup L_U \dot\cup \{(unobservable)\}$, with the 9 input gates $L_I = \{$`?generateLicenseInput`, `?removeLicenseInput`, `?removeAllLicensesInput`, `?removeExpiredLicensesInput`, `?showLicensesInput`, `?makeBackupOnlyInput`, `?localBackupInput`, `?remoteBackupInput`, `?abortOnlyInput`$\}$, the 7 output gates $L_U = \{$ `!generateLicensesOutput`, `!removeLicenseOutput`, `!removeAllLicensesOutput`,

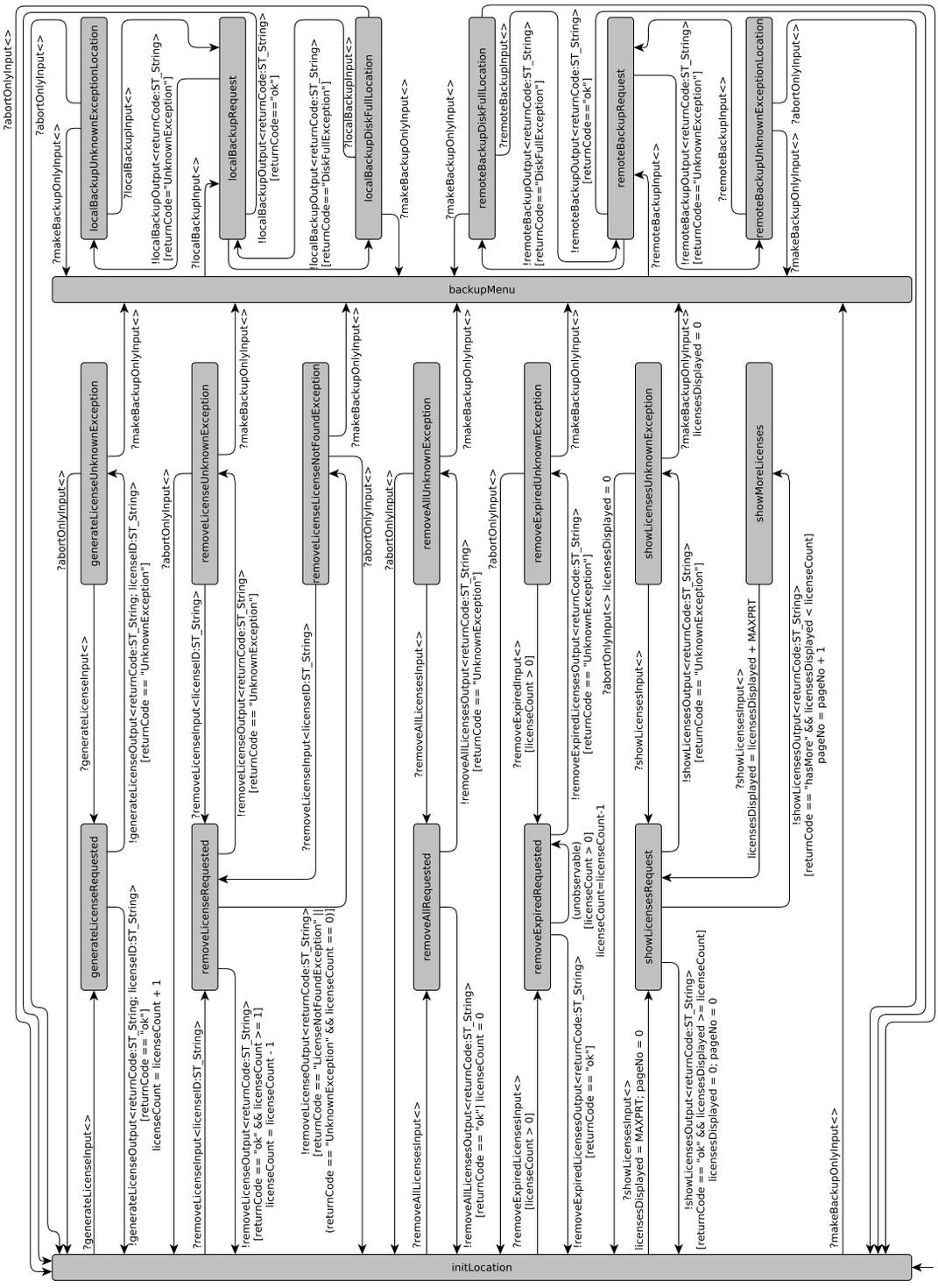


Figure B.1.: Abstract specification of the License Central, described as STS $\mathcal{S}_{LC}$, with maximal portion $MAXPRT \in \mathbb{N}_{>0}$

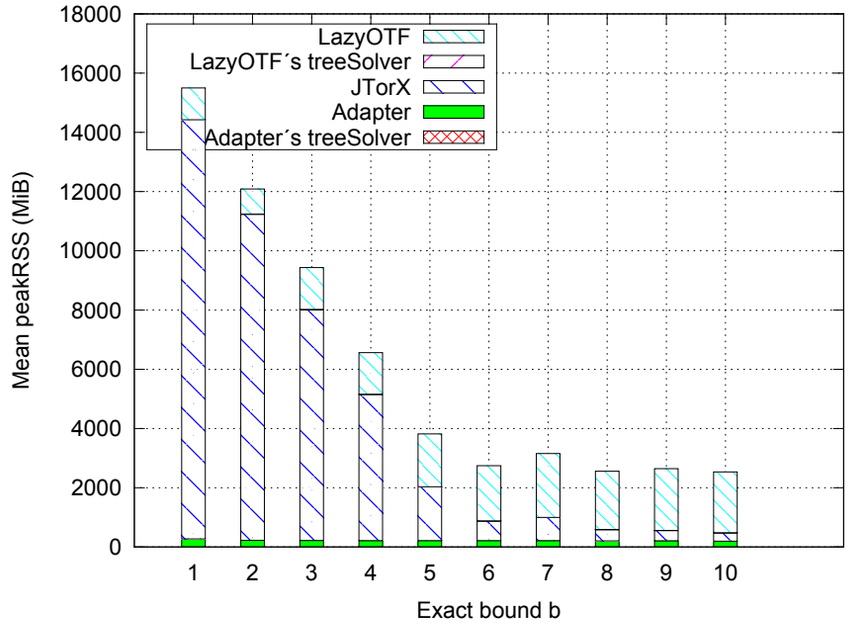


Figure B.2.: Memory requirements (of storing communication) for a long running experiment, depending on the bound (i.e., on the amount of communication)

`!removeExpiredLicensesOutput`, `!showLicensesOutput`, `!localBackupOutput`, `!remoteBackupOutput`}, and the τ transition labeled `(unobservable)`;

- $\mathcal{V}_{LC}$ = {`licenseCount`, `licensesDisplayed`, `pageNo`}, all of type bounded non-negative (32 bit) integer;
- $\mathcal{I}_{LC}$ = {`returnCode`, `licenseID`}, where `licenseID` is a bounded non-negative (32 bit) integer and `returnCode` is of type string;
- S^0_{LC} = {`initLocation`};
- $\rightarrow_{LC}$ consists of the 58 switches depicted in Fig. B.1, where 39 switches have input gates, 18 switches have output gates, and one transition is a τ-cycle in the abstract state `removeExpiredRequested`.

B.1.2. General Performance Plots

Storing Communication Between JTorX and LazyOTF

Fig. B.2 shows an example where storing communication between LazyOTF and JTorX causes a strong increase of JTorX's memory requirement: for the experiment Fig 14.6 with $MAXPRT$=10 and exactBound, the number of traversalSubphases is very high ($\approx 2E6$ resp. $5E5$ resp. $1.7E5$ for $b = 1$ resp. 2 resp. 3, cf. Subfig. 14.6c), causing a huge amount of communication for small b. This leads to many GiB of data in JTorX's memory. When b increases, the number of traversalSubphases and hence the communication and memory requirement strongly decreases.

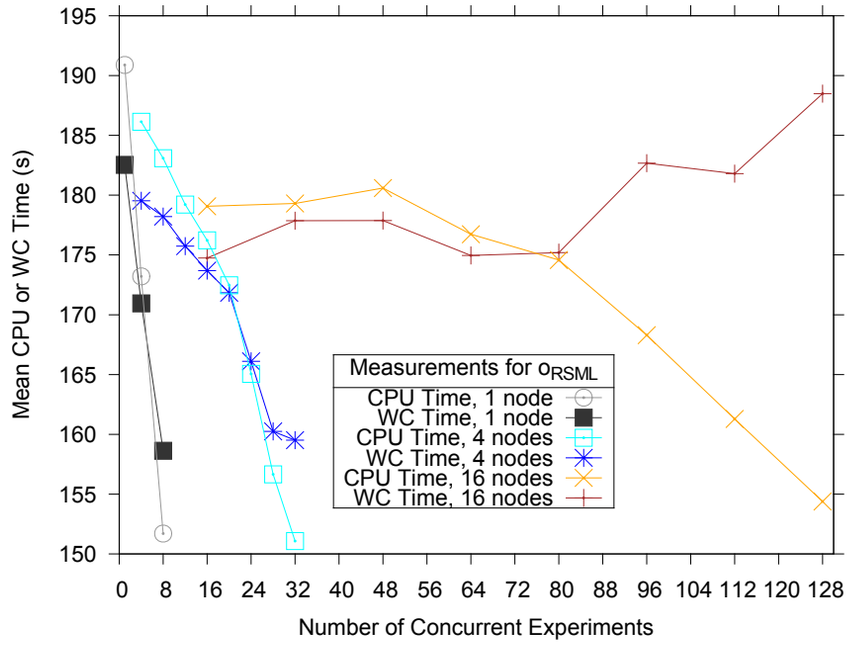


Figure B.3.: Hardware Performance (CPU and WC time) for Concurrent Experiments

Hardware Performance

To investigate how much our experiment results are influenced by the hardware and its contention for concurrent experiments, we measure the overall CPU time and WC time for increasingly higher loads for 1 resp. 4 resp. 16 nodes, using LazyOTF instances with o_{RSML} with $MAXPRT = b = 5$, whose verbose logging has a relatively high, but still representative throughput.

Fig. B.3 shows that for increasing load per node, the average CPU time per experiment decreases, independent of whether 1, 4, or 16 nodes are employed. The average WC time per experiment also decreases, but not as strongly and not above 80 concurrent experiments.

The consistent decrease in the average CPU time per experiment that occurs for higher load is probably due to the CPU's Enhanced Intel SpeedStep and Demand Based Switching, which are "power-management technologies in which the applied voltage and clock speed of a microprocessor are kept at the minimum necessary levels until more processing power is required", i.e., voltage and frequency are "switched in response to processor load" [URL:XeonHP]. Consequently, the average WC time per experiment also decreases for higher load (and at most 80 concurrent experiments), but slightly less than the CPU time. This shows that there is some small local contention, e.g., due to local disk storage, internal busses, memory, or the network. For 16 nodes, the average WC time and CPU time are almost identical for up to 80 concurrent experiments, but thereafter the average WC time increases. This shows that the RAID does not cause I/O contention for up to 80 concurrent experiments, and is also not the cause for the small local contention. But for 96 and more concurrent experiments, there is notable contention, likely I/O contention due to the RAID.

So the overall computation successively converts from a CPU-bound to a probably I/O-bound problem for 96 and more instances. Thus verbose logging, even with relatively high throughput, scales well up to at least 80 instances.

This decrease in CPU time and small local contention show that CPU time and WC time have to be measured and interpreted carefully.

Performance Scatter Plot

Fig. B.4 shows an exemplary scatter plot for the different numbers of test steps in each experiment. So the dependent variable is t_{curr}^{max}, the independent variable is the experiment id (a random 6 digit string and hence not displayed in the plot), sorted according to the maximal t_{curr}^{max} in each experiment. The measurements are from our experiment in Subsec. 14.3.7 for $P = 14$. The plots for other P look similarly.

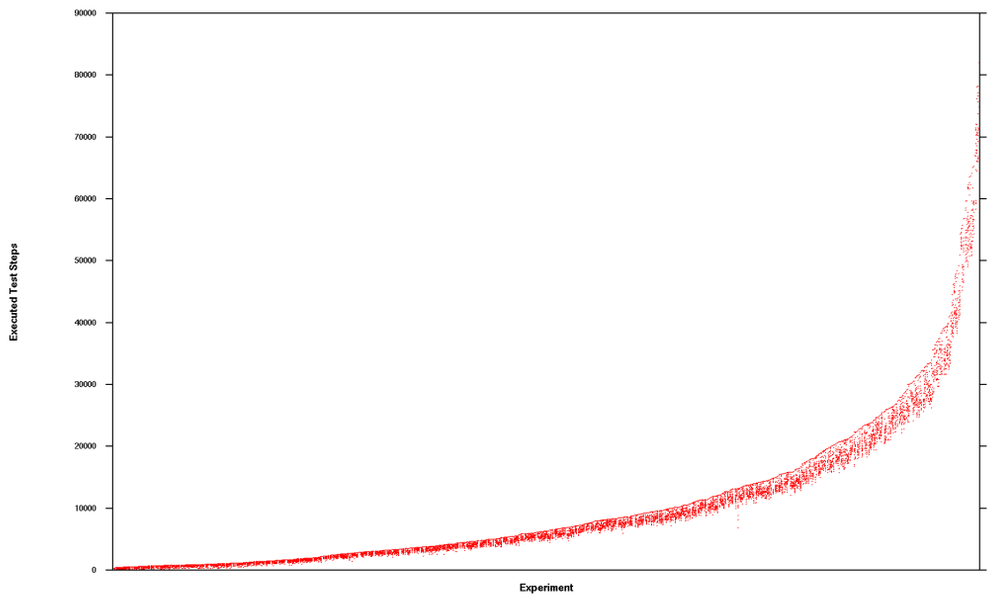


Figure B.4.: Scatter plot for parallel experiment with P=14

The plot shows that the variance in the number of test steps is proportional to the numbers of test steps each experiment requires, suggesting that the different nodes in the cluster have slightly different runtime, i.e., vary in their performance. This again shows that CPU time and WC time have to be measured and interpreted carefully.

B.1.3. Exemplary Plot for CPU Time per Test Step

Fig. B.5 shows an exemplary plot for the mean CPU time per test step, up to *MAXPRT* = 10. Due to LazyOTF's overhead for setup, waiting and cleanup (cf. Subsec. 14.3.8), the value decreases for increasing *MAXPRT*. Since the value approaches a constant for increasing *MAXPRT* up to 10, the mean CPU time per test step does not increase with *MAXPRT*. For other experiments, the plots look similarly (except for $\ddot{o} = \{o_{RSML}, \ddot{o}_{cov}\}$

for Max and MaxMax, cf. Subsec. 14.3.5). The mean CPU time per test step for $MAXPRT > 10$ are also considered in Subsec. 14.3.4.

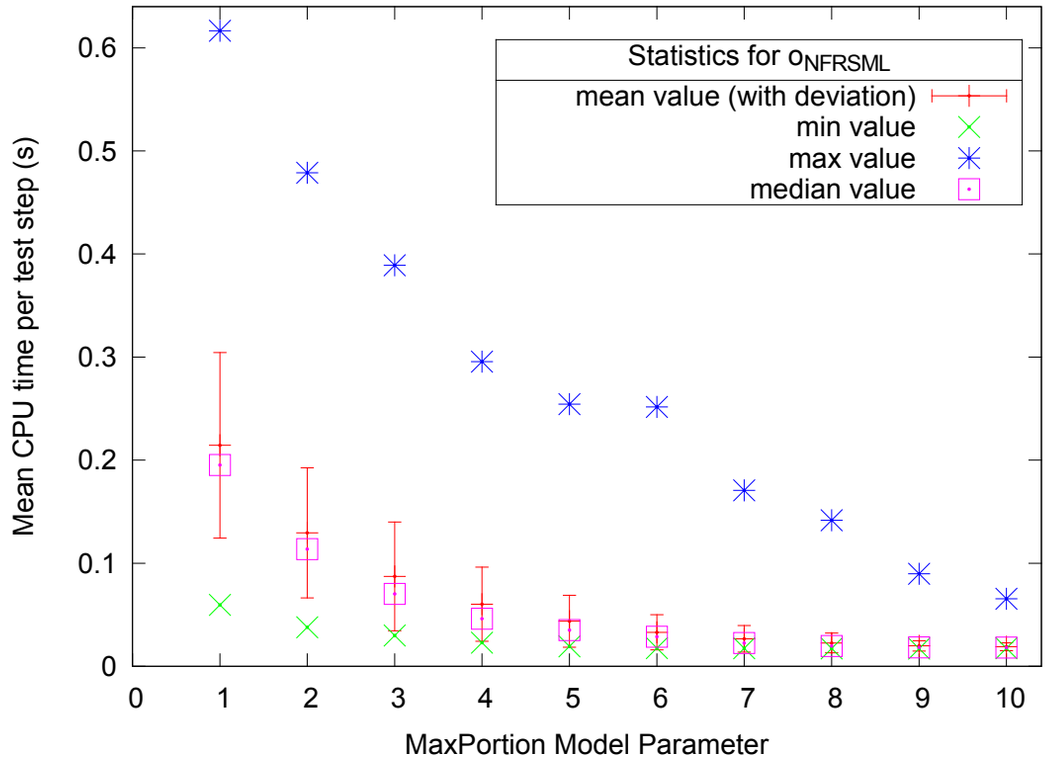


Figure B.5.: Mean CPU time per test step for o_{NFRSML} with bound $b = 5$

B.1.4. Plots for Distributed LazyOTF

Hazelcast overhead

Hazelcast's powerful features cost overhead for setup and cleanup (cf. Subsec. 3.5.2). Fig. B.6 depicts the expected WC time for Hazelcast's setup and cleanup, plotted against the parallelization degree $P \in [2, \ldots, 16]$. The measurements are from our experiment in Subsec. 14.3.7, and we again consider for each distributed sample on P nodes

- *max*, i.e., the setup and cleanup WC time of the node with the highest WC time;
- *avg*, i.e., the average setup and cleanup WC time over all P nodes;
- *min*, i.e., the setup and cleanup WC time of the node with the smallest WC time.

The WC time for setup and cleanup are about 25 seconds for *min*, and almost constant. For *max* resp. *avg*, the WC time also starts at about 25 seconds, but increases linearly with P, up to over 100 seconds resp. 70 seconds. The costs for communication are often high for parallel algorithms [Bader et al., 2000], for Hazelcast luckily only for setup and cleanup, not while running (cf. Subsec. 14.3.7). The WC time for cleanup was often higher in our experiments than for setup, but cannot be measured easily, since it

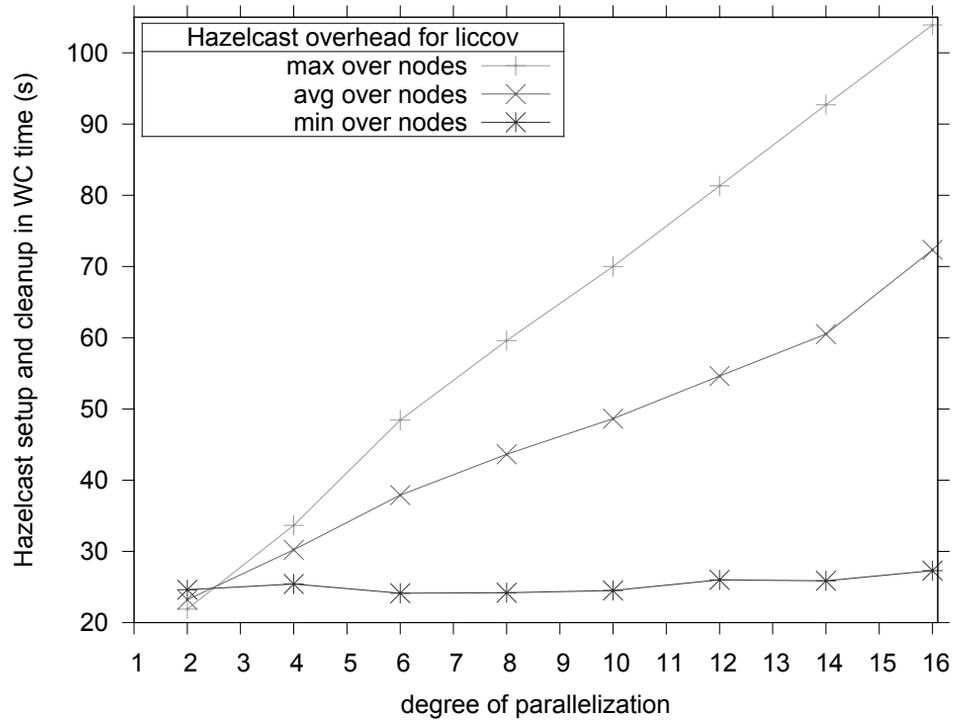


Figure B.6.: Hazelcast's WC time for setup and cleanup

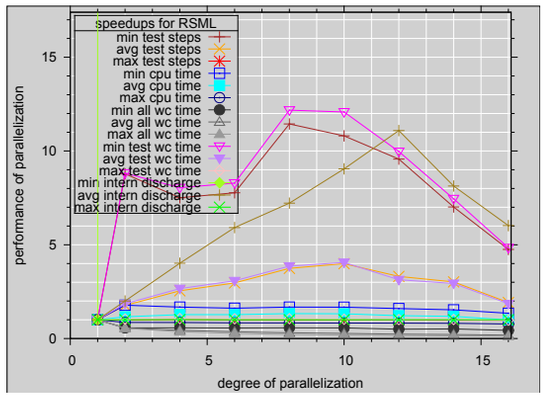
depends on which instance leaves the Hazelcast cluster when; the cleanup of one node performs repartitioning and new backups when another node leaves the Hazelcast cluster earlier (cf. Subsec. 3.5.2). This explains why *min* WC time over the nodes remains almost constant, whereas *max* WC time over the nodes increases to over 100 seconds for $P = 16$. To avoid these setup and cleanup costs, one Hazelcast cluster can be kept running for multiple experiments by using its distributed executor service and elasticity.

Simple TOs

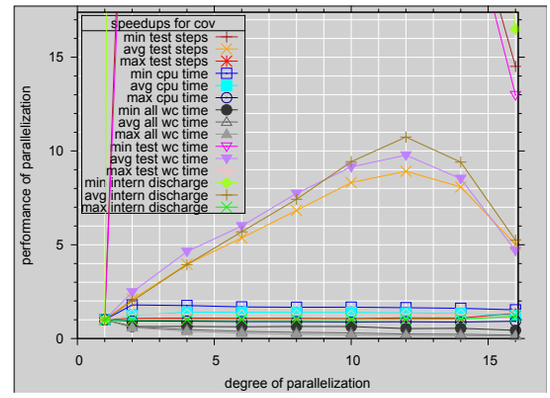
Fig. B.7 shows our parallel experiments for o_{RSML}, $\ddot{o}_{cov}$ and $\{o_{RSML}, \ddot{o}_{cov}\}$, analogously to Fig. 14.7. They have not been investigated in Subsec. 14.3.7 since strong scaling analysis should only consider sufficiently large problem sizes [Bader et al., 2000].

B.1.5. Table Describing the Amount of Experiments

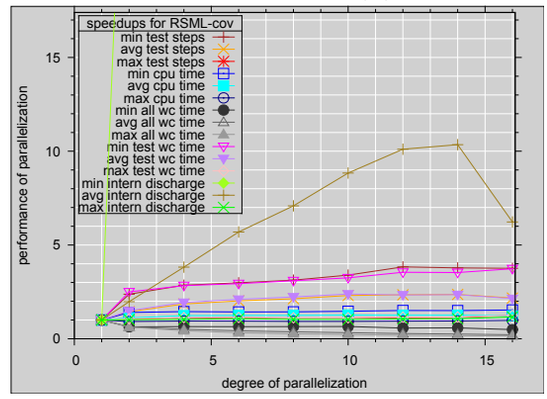
To consider the effort of our experiments in Sec. 14.3, we give measures for each experiment. An **experiment sample** is one single (concurrent) execution, an **experiment sample set** the set of all experiment samples for one experiment with one specific value for the independent variable (usually $MAXPRT \in [1, \ldots, 10]$), which usually leads to one point in a plot. An **experiment** is the set of all experiment sample sets taken over all possible values for the independent variable, which usually leads to one function in a plot. Table B.1 shows the RSEM of t_{curr}^{max}, the required sample size n, and accumulated



(a) Distributed LazyOTF for o_{RSML}



(b) Distributed LazyOTF for $\ddot{o}_{cov}$



(c) Distributed LazyOTF for $\{o_{RSML}, \ddot{o}_{cov}\}$

Figure B.7.: Distributed LazyOTF for o_{RSML}, $\ddot{o}_{cov}$, and their composition

WC time t (to reach the RSEM $< 5\%$). The **accumulated WC time** takes the all WC time (cf. Subsec. 14.3.3) for each experiment sample (despite concurrent execute), and sums up those values for the considered experiment sample set (since many experiment samples were executed in isolation on the cluster). In detail,

- the first column describes the considered experiment;
- the second column shows the minimal and maximal RSEM for that experiment, taken over all experiment sample sets;
- the third column shows the minimal and maximal accumulated WC time, taken over all experiment sample sets;
- the fourth column shows the sum of all accumulated WC times, taken over all experiment sample sets;
- the fifth column shows the minimal and maximal sample size, taken over all experiment sample sets;
- the sixth column shows the sum of all sample sizes, taken over all experiment sample sets.

In summary, we executed 32 of our LazyOTF experiments between 90 and 10444 times to push the RSEM of t_{curr}^{max} to less than 5%. For 3 LazyOTF experiments (Fig. 14.3b,

o_{NFRSML} with b=10, Fig. 14.6 with $MAXPRT$=10 and 4uptoBound resp. 5uptoBound), we have a sample size between 90 and 245 to push the RSEM of t_{curr}^{max} to less than 10%; for a lower RSEM, the accumulated WC time was too high for each (between 47 and 156 days).

For our LazyOTF experiment Fig. B.2, we have an RSEM of t_{curr}^{max} that is less than 15%, with an accumulated WC time over 90 days. But this experiment was not our focus since we can deactivate storing communication for experiments that run much longer than the average. Our hardware performance experiment Fig. B.3 were also not our focus, so we only executed between 5 and 640 runs, with an RSEM between 0.3% and 3.1%. For our preliminary experiments, we have RSEMs between 0% and 48.5%, with 1 to 30 runs. Our 7 JTorX experiments (first rows in Table B.1) were also not the focus of this thesis, and we have an RSEM of t_{curr}^{max} that is between 4.4% and 15.3%.

Automating these experiments required over 100 bash scripts, evaluating them over 50 Java classes.

Table B.1.: The (min-max or sum) values for RSEM of t_{curr}^{max} (in %), sample size n, and total WC time t (in minutes or days) per experiment

(Figure,) experiment	RSEM	t	sum t	n	sum n
Fig. 14.3c, $\ddot{o}_{cov}$	6.5% - 10.0%	10.7' - 17.8'	139.2'	70 - 70	700
Fig. 14.3c, o_{dec}	10.6% - 13.9%	20.6' - 7.7d	8.5d	70 - 70	280
Fig. 14.3c, $\{o_{RSML}, o_{dec}, \ddot{o}_{cov}\}$	10.7% - 15.3%	21.6' - 7.6d	8.7d	70 - 70	280
Fig. 14.3c, $\{o_{RSML}, \ddot{o}_{cov}\}$	10.3% - 14.8%	16.3' - 10.0d	16.5d	64 - 70	694
Fig. 14.3c, $\{o_{RSML}, o_{dec}\}$	10.1% - 13.2%	21.7' - 9.1d	9.9d	70 - 70	280
Fig. 14.3c, o_{RSML}	9.8% - 13.0%	11.5' - 10.4d	17.7d	70 - 70	700
Fig. 14.3d, $\ddot{o}_{S_{isol}}$	4.4% - 6.0%	35.6' - 4.8d	10.3d	91 - 152	962
Fig. 14.3a, o_{RSML}	2.1% - 4.2%	17.7' - 25.3'	210.0'	90 - 90	900
Fig. 14.3a, $\ddot{o}_{cov}$	2.1% - 2.8%	18.6' - 27.9'	214.5'	90 - 90	900
Fig. 14.3a, o_{dec}	1.7% - 4.8%	18.6' - 28.2'	222.6'	90 - 90	900
Fig. 14.3a, $\{o_{RSML}, \ddot{o}_{cov}\}$	1.4% - 2.3%	22.3' - 32.3'	249.7'	90 - 90	900
Fig. 14.3a, $\{o_{RSML}, o_{dec}\}$	1.7% - 3.4%	18.2' - 31.7'	217.0'	90 - 90	900
Fig. 14.3a, $\{o_{RSML}, o_{dec}, \ddot{o}_{cov}\}$	1.4% - 2.2%	21.4' - 37.3'	250.5'	90 - 90	900
Fig. 14.3b, o_{NFRSML},b=10	4.5% - 9.7%	44.0' - 26.8d	46.6d	90 - 173	1016
Fig. 14.3b, o_{NFRSML},b=5	1.7% - 4.9%	167.9' - 9.6d	22.0d	412 - 1524	12076
Fig. 14.3b, $\ddot{o}_{S_{isol}}$	1.4% - 4.9%	139.4' - 21.4d	39.4d	530 - 578	5400
Fig. 14.4a, o_{RSML}	2.3% - 4.7%	14.2' - 25.0'	183.4'	120 - 130	1210
Fig. 14.4a, $\ddot{o}_{cov}$	2.0% - 2.5%	14.8' - 16.2'	154.9'	100 - 100	1000
Fig. 14.4a, $\{o_{RSML}, \ddot{o}_{cov}\}$	2.2% - 2.9%	16.5' - 41.7'	283.4'	100 - 100	1000
Fig. 14.4b, $\ddot{o}_{S_{isol}}$	1.6% - 5.0%	57.0' - 18.1d	30.4d	220 - 413	2853
Fig. 14.4b, o_{NFRSML},b=5	2.3% - 4.6%	123.2' - 14.6d	26.5d	539 - 876	7718
Fig. 14.4b, o_{NFRSML},b=10	4.6% - 4.9%	152.5' - 147.4d	253.3d	212 - 428	3239
Fig. 14.4c, o_{RSML}	3.8% - 4.9%	38.7' - 125.7'	589.6'	240 - 280	2580
Fig. 14.4c, $\ddot{o}_{cov}$	2.1% - 2.8%	15.2' - 44.0'	234.9'	100 - 130	1180
Fig. 14.4c, $\{o_{RSML}, \ddot{o}_{cov}\}$	2.2% - 4.9%	25.0' - 56.8'	368.4'	110 - 130	1170
Fig. 14.4d, $\ddot{o}_{S_{isol}}$	1.1% - 5.0%	175.7' - 28.3d	83.0d	195 - 681	5901
Fig. 14.4d, o_{NFRSML},b=5	3.6% - 4.9%	52.3' - 26.8d	69.8d	214 - 597	4338
Fig. 14.4d, o_{NFRSML},b=10	3.8% - 5.0%	416.5' - 624.7d	1520.6d	231 - 443	3968
Fig. 14.6, $MAXPRT$=1,1uptoBound	2.7% - 3.1%	32.4' - 81.0'	146.1'	118 - 312	548
Fig. 14.6, $MAXPRT$=1,4uptoBound	3.2% - 3.3%	17.4' - 17.7'	35.1'	102 - 102	204
Fig. 14.6, $MAXPRT$=1,5uptoBound	2.6% - 2.8%	18.1' - 18.2'	36.4'	100 - 101	201
Fig. 14.6, $MAXPRT$=1,exactBound	1.4% - 3.3%	63.5' - 1.4d	2.5d	306 - 308	3069
Fig. 14.6, $MAXPRT$=10,1uptoBound	3.8% - 4.7%	42.1d - 430.8d	630.2d	187 - 338	854
Fig. 14.6, $MAXPRT$=10,4uptoBound	8.0% - 9.9%	33.3d - 95.5d	128.8d	117 - 134	251
Fig. 14.6, $MAXPRT$=10,5uptoBound	6.9% - 8.7%	50.2d - 105.4d	155.7d	180 - 245	425
Fig. 14.6, $MAXPRT$=10,exactBound	3.2% - 4.8%	13.9d - 650.4d	1497.3d	335 - 1014	6663
Fig. 14.7, $\ddot{o}_{S_{isol}}$	1.2% - 3.1%	24.4d - 46.1d	262.7d	1180 - 10444	45818
Fig. 14.8, o_{RSML}	0.0% - 0.0%	14.4' - 19.8'	166.6'	120 - 122	1206
Fig. 14.8, o_{NFRSML}	0.0% - 5.0%	117.1' - 72.9d	121.0d	427 - 474	4639
Subsec. 14.3.9	0% - 48.5%	1' - 1.9d	6.3d	1 - 30	1046
Fig. B.2, $\ddot{o}_{S_{isol}}$	10.4% - 14.1%	253.9' - 34.1d	93.9d	5 - 87	631
Fig. B.3	0.3% - 1.9%	15.2' - 1.4d	7.8d	5 - 640	3744
Fig. B.7a, o_{RSML}	1.4% - 3.1%	910.3' - 5.0d	18.1d	934 - 3990	17074
Fig. B.7b, $\ddot{o}_{cov}$	3.1% - 5.0%	885.4' - 4.9d	18.4d	946 - 3964	17512
Fig. B.7c, $\{o_{RSML}, \ddot{o}_{cov}\}$	0.7% - 1.7%	821.6' - 4.9d	17.9d	638 - 3504	15024
Total amount	about 14 years			about 180000	

Bibliography

IEEE 829WG. IEEE 829 Standard for Software and System Test Documentation. Technical report, IEEE, 2008. URL `https://standards.ieee.org/findstds/standard/829-2008.html`. (Cited on page 13.)

Aynur Abdurazik, Paul Ammann, Wei Ding 0003, and A. Jefferson Offutt. Evaluation of Three Specification-Based Testing Criteria. In **ICECCS**, pages 179–187. IEEE Computer Society, 2000. ISBN 0-7695-0583-X. (Cited on pages 22, 261, and 287.)

Harold Abelson, Gerald Jay Sussman, and with Julie Sussman. **Structure and Interpretation of Computer Programs**. MIT Press/McGraw-Hill, Cambridge, 2nd editon edition, 1996. ISBN 0-262-01153-0. `http://www.csie.ncnu.edu.tw/~klim/lisp-922/scheme.pdf` `http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-1.html` – last visited 26^{th} December 2007. (Cited on page 54.)

Harold Abelson et al. Revised report on the algorithmic language Scheme. **Higher-order and symbolic computation**, 11(1):7–105, 1998. (Cited on page 252.)

David Abrahams. Exception-Safety in Generic Components. In Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors, **Generic Programming**, volume 1766 of **Lecture Notes in Computer Science**, pages 69–79. Springer, 1998. ISBN 3-540-41090-2. URL `http://dblp.uni-trier.de/db/conf/dagstuhl/generic1998.html#Abrahams98`. (Cited on page 204.)

Alain Abran, James W. Moore, Pierre Bourque, Robert Dupuis, and Leonard L. Tripp. **Guide to the Software Engineering Body of Knowledge (SWEBOK)**. IEEE, 2004. (Cited on page 19.)

Jean-Raymond Abrial. **Modeling in Event-B - System and Software Engineering.** Cambridge University Press, 2010. ISBN 978-0-521-89556-9. (Cited on page 60.)

Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. **IEEE Computer**, 29(12):66–76, 1996. (Cited on page 51.)

Bernhard K Aichernig and Martin Tappler. Symbolic Input-Output Conformance Checking for Model-Based Mutation Testing. **USE**, 2015. (Cited on pages 261 and 339.)

Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Protocol Conformance Testing a SIP Registrar: an Industrial Application of Formal Methods. In **SEFM**, pages 215–226. IEEE Computer Society, 2007. ISBN 978-0-7695-2884-7. (Cited on page 261.)

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Efficient Mutation Killers in Action. In **ICST 11: International Conference on Software Testing, Verification and Validation: Proceedings of the Fourth IEEE International Conference on Software Testing, Verification, and Validation**, pages 120–129. IEEE Computer Society, 2011. ISBN 978-0-7695-4342-0. (Cited on pages 24 and 261.)

Bernhard K Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. **Software Testing, Verification and Reliability**, 2014. (Cited on pages 24 and 261.)

Samy Al-Bahra. Nonblocking algorithms and scalable multicore programming. **Commun. ACM**, 56(7):50–61, 2013. URL `http://dblp.uni-trier.de/db/journals/cacm/cacm56.html#Al-Bahra13`. (Cited on pages 50 and 51.)

Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. **IEEE Trans. Software Eng.**, 36(6):742–762, 2010. URL `http://dblp.uni-trier.de/db/journals/tse/tse36.html#AliBHP10`. (Cited on pages 23, 241, 259, 287, and 288.)

Teresa Alsinet, Felip Manya, and Jordi Planes. A Max-SAT Solver with Lazy Data Structures. In Christian Lemaitre, Carlos A. Reyes GarcÃa, and JesÃºs A. Gonzalez, editors, **Advances in Artificial Intelligence (IBERAMIA 2004)**, volume 3315 of **Lecture Notes in Computer Science**, pages 334–342. Springer, 2004. ISBN 3-540-23806-9. (Cited on page 55.)

Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, **ICALP 90: Automata, Languages, and Programming**, volume 443 of **LNCS**, pages 322–335. Springer-Verlag, 1990. (Cited on page 33.)

Rajeev Alur, Thomas Henzinger, Orna Kupferman, and Moshe Vardi. Alternating refinement relations. **CONCUR'98 Concurrency Theory**, pages 163–178, 1998. doi: 10.1007/BFb0055622. URL `http://dx.doi.org/10.1007/BFb0055622`. (Cited on pages 234 and 251.)

Scott Ambler. **Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process**. John Wiley & Sons, Inc., New York, NY, USA, 2002. ISBN 047127190X. (Cited on pages 55, 342, and 348.)

Scott Ambler. Has agile peaked? **Dr. Dobb's**, 2008. URL `http://www.ddj.com/architecture-and-design/207600615`. (Cited on page 341.)

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In **AFIPS Spring Joint Computing Conference**, volume 30 of **AFIPS Conference Proceedings**, pages 483–485. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967. URL `http://dblp.uni-trier.de/db/conf/afips/afips67s.html#Amdahl67`. (Cited on pages 49 and 151.)

Paul Ammann and Jeff Offutt. **Introduction to software testing**. Cambridge University Press, 2008. ISBN 978-0-521-88038-1. (Cited on pages 19, 21, 22, and 25.)

Paul Ammann, Paul E. Black, and William Majurski. Using Model Checking to Generate Tests from Specifications. In **ICFEM**, pages 46–54, 1998. (Cited on page 261.)

Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. **Journal of Systems and Software**, 86(8):1978–2001, 2013. (Cited on pages 184, 204, 241, 246, 251, 256, 270, 271, 287, 288, 289, 297, 348, 349, and 371.)

Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, Gabriele Palma, and Salvatore Sabina. Improving the automatic test generation process for coverage analysis using cbmc. In **Proceedings of the 16th International RCRA workshop, RCRA**, volume 2009, 2009a. (Cited on page 175.)

Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. Automatic test generation for coverage analysis using CBMC. In **Computer Aided Systems Theory-EUROCAST 2009**, pages 287–294. Springer, 2009b. (Cited on page 175.)

Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. Automatic Test Generation for Coverage Analysis of ERTMS Software. In **ICST**, pages 303–306. IEEE Computer Society, 2009c. ISBN 978-0-7695-3601-9. (Cited on page 175.)

Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. **Journal of Object Technology**, 8(5):49–84, 2009. URL `http://dblp.uni-trier.de/db/journals/jot/jot8.html#ApelK09`. (Cited on page 176.)

Sylwester Arabas, Michael R. Bareford, Ian P. Gent, Benjamin M. Gorman, Masih Hajiarabderkani, Tristan Henderson, Luke Hutton, Alexander Konovalov, Lars Kotthoff, Ciaran McCreesh, Ruma R. Paul, Karen E. Petrie, Abdul Razaq, and Daniël Reijsbergen. An Open and Reproducible Paper on Openness and Reproducibility of Papers in Computational Science. **CoRR**, abs/1408.2123, 2014. URL `https://github.com/larskotthoff/recomputation-ss-paper`. (Cited on page 354.)

Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Combining Model-Based Testing and Runtime Monitoring for Program Testing in the Presence of Nondeterminism. In **Proceedings of the Sixth International Conference on Software Testing, Verification and Validation (ICST), Workshops Proceedings, Luxembourg**, pages 178–187. IEEE, 2013. ISBN 978-1-4799-1324-4. URL `http://dblp.uni-trier.de/db/conf/icst/icstw2013.html#ArcainiGR13`. (Cited on pages 14, 242, 246, 256, 260, 271, 272, 310, and 377.)

Andrea Arcuri and Lionel C. Briand. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. **Softw. Test., Verif. Reliab.**, 24(3): 219–250, 2014. (Cited on pages 371 and 372.)

Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. **STTT**, 11(1):69–83, 2009. URL `http://dblp.uni-trier.de/db/journals/sttt/sttt11.html#ArmandoMP09`. (Cited on page 106.)

Andrea Asperti and Stefano Guerrini. **The optimal implementation of functional programming languages**, volume 45. Cambridge University Press, 1998. (Cited on page 54.)

Motor Industry Research Association. **MISRA-C 2004: Guidelines for the Use of the C Language in Critical Systems**. Motor Industry Research Association, September 2004. ISBN 0952415623. (Cited on page 157.)

Diyaa-Addein Atiya, Néstor Catano, and Gerald Lüttgen. Towards a benchmark for model checkers of asynchronous concurrent systems. In **University of Warwick, United Kingdom**, 2005. (Cited on page 116.)

Algirdas Avizienis. The methodology of n-version programming. **Software fault tolerance**, 3:23–46, 1995. (Cited on pages 17 and 43.)

Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using FindBugs on production software. In **OOPSLA Companion**, pages 805–806, 2007. (Cited on page 344.)

Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to Büchi Automata Translation: Fast and More Deterministic. In Cormac Flanagan and Barbara König, editors, **TACAS**, volume 7214 of **Lecture Notes in Computer Science**, pages 95–109. Springer, 2012. ISBN 978-3-642-28755-8. URL `http://dblp.uni-trier.de/db/conf/tacas/tacas2012.html#BabiakKRS12`. (Cited on pages 88 and 97.)

David A. Bader, Bernard M. E. Moret, and Peter Sanders. Algorithm Engineering for Parallel Computation. In Rudolf Fleischer, Bernard M. E. Moret, and Erik Meineche Schmidt, editors, **Experimental Algorithmics**, volume 2547 of **Lecture Notes in Computer Science**, pages 1–23. Springer, 2000. ISBN 3-540-00346-0. (Cited on pages 49, 151, 348, 364, 369, 371, 372, 373, 390, and 391.)

Xiaoying Bai, Muyang Li, Bin Chen, Wei-Tek Tsai, and Jerry Gao. Cloud testing tools. In Jerry Zeyu Gao, Xiaodong Lu, Muhammad Younas, and Hong Zhu, editors, **6th International Symposium on Service Oriented System Engineering, SOSE 2011**, pages 1–12. IEEE, December 2011. ISBN 978-1-4673-0411-5. URL `http://dblp.uni-trier.de/db/conf/sose/sose2011.html#BaiLCTG11`. (Cited on page 272.)

Christel Baier and Joost-Pieter Katoen. **Principles of model checking.** MIT Press, 2008. ISBN 978-0-262-02649-9. (Cited on pages 65, 67, 68, 72, 76, 77, 78, 97, 100, 101, 104, 107, 114, 129, and 133.)

Richard Baker and Ibrahim Habli. An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software. **IEEE Transactions on**

Software Engineering, 39(6):787–805, 2013. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2012.56. (Cited on pages 2, 23, and 24.)

Henry C Baker Jr and Carl Hewitt. The incremental garbage collection of processes. **ACM SIGART Bulletin**, 12(64):55–59, 1977. (Cited on pages 49 and 54.)

Adrian Balint, Anton Belov, M Heule, and Matti Järvisalo. SAT COMPETITION 2013, 2013. (Cited on page 27.)

Helmut Balzert. **Lehrbuch der Software-Technik, Bd. 2: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung, inkl. 1 CD-ROM**. Spektrum Akademischer Verlag, November 1997. ISBN 9783827400659. (Cited on pages 18 and 343.)

Mounika Banka and Madhuri Kolla. Merging Functional Requirements with Test Cases. 2015. (Cited on pages 17, 18, and 333.)

Jiri Barnat. **Platform-Dependent Verification**. Habilitation, Masaryk University, October 2010. (Cited on page 56.)

Jiri Barnat. Quo Vadis Explicit-State Model Checking. In Giuseppe F. Italiano, Tiziana Margaria-Steffen, Jaroslav Pokornv, Jean-Jacques Quisquater, and Roger Wattenhofer, editors, **SOFSEM**, volume 8939 of **Lecture Notes in Computer Science**, pages 46–57. Springer, 2015. ISBN 978-3-662-46077-1. (Cited on pages 92 and 102.)

Jiri Barnat, Lubos Brim, and Petr Rockai. Scalable Multi-core LTL Model-Checking. In Dragan Bosnacki and Stefan Edelkamp, editors, **Proceedings of the 14th International SPIN Workshop on Model Checking Software, Berlin, Germany**, volume 4595 of **LNCS**, pages 187–203. Springer, July 2007. ISBN 978-3-540-73369-0. URL `http://dblp.uni-trier.de/db/conf/spin/spin2007.html#BarnatBR07`. (Cited on page 112.)

Jiri Barnat, Lubos Brim, and Petr Rockai. A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In **ICFEM 2009**, volume 5885 of **LNCS**, pages 407–425. Springer, 2009. (Cited on pages 56, 113, 144, 145, and 377.)

Jiri Barnat, Lubos Brim, and Petr Rockai. Parallel Partial Order Reduction with Topological Sort Proviso. In **SEFM**, pages 222–231. IEEE Computer Society, 2010. ISBN 978-0-7695-4153-2. doi: 10.1109/SEFM.2010.35. URL `http://dx.doi.org/10.1109/SEFM.2010.35`. (Cited on pages 112 and 144.)

Jiri Barnat, Jan Havlíček, and Petr Rockai. Distributed LTL Model Checking with Hash Compaction. In **PASM/PDMC**, ENTCS. Elsevier, 2012. URL `http://eprints.eemcs.utwente.nl/22042/`. (Cited on page 147.)

Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlicek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimir Still, and Jiri Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In **To appear in Computer Aided Verification (CAV 2013)**, page 6. LNCS, 2013. (Cited on page 112.)

Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In **Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)**, volume 13, page 14, 2010. (Cited on pages 31 and 32.)

Clark Barrett, Morgan Deters, Leonardo Mendonca de Moura, Albert Oliveras, and Aaron Stump. 6 Years of SMT-COMP. **J. Autom. Reasoning**, 50(3):243–277, 2013. URL `http://dblp.uni-trier.de/db/journals/jar/jar50.html#BarrettDMOS13`. (Cited on pages 31 and 47.)

MdO Barros and AC Dias-Neto. Threats to validity in search-based software engineering empirical studies. **Relatórios Técnicos do DIA/UNIRIO**, (0006), 2011. (Cited on page 369.)

Andreas Bauer and Patrik Haslum. LTL Goal Specifications Revisited. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, **ECAI**, volume 215 of **Frontiers in Artificial Intelligence and Applications**, pages 881–886. IOS Press, 2010. ISBN 978-1-60750-605-8. URL `http://dblp.uni-trier.de/db/conf/ecai/ecai2010.html#BauerH10`. (Cited on page 72.)

Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL Semantics for Runtime Verification. **J. Log. Comput.**, 20(3):651–674, 2010. URL `http://dblp.uni-trier.de/db/journals/logcom/logcom20.html#BauerLS10`. (Cited on pages 14, 72, and 93.)

Kent Beck. **Test Driven Development: By Example**. Addison-Wesley Professional, 2002. (Cited on page 343.)

Kent Beck and Cynthia Andres. **Extreme Programming Explained: Embrace Change**. Addison-Wesley, Boston, 2 edition, 2004. ISBN 0321278658. URL `http://portal.acm.org/citation.cfm?id=1076267`. (Cited on page 1.)

Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. **Verification of object-oriented software: The KeY approach**. Springer-Verlag, 2007. (Cited on page 2.)

Bernhard Beckert, Christoph Gladisch, Shmuel S. Tyszberowicz, and Amiram Yehudai. KeYGenU: combining verification-based and capture and replay techniques for regression unit testing. **Int. J. Systems Assurance Engineering and Management**, 2(2):97–113, 2011. URL `http://dblp.uni-trier.de/db/journals/saem/saem2.html#BeckertGTY11`. (Cited on page 241.)

Boris Beizer. **Software Testing Techniques**. International Thomson Computer Press, 2 edition, June 1990. (Cited on page 19.)

Axel Belinfante. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In Javier Esparza and Rupak Majumdar, editors, **Tools and Algorithms for the Construction and Analysis of Systems, 2010**, volume 6015 of **Lecture Notes in Computer Science**, pages 266–270. Springer, 2010. ISBN 978-3-642-12001-5. URL `http://dblp.uni-trier.de/db/conf/tacas/tacas2010.html#Belinfante10`. (Cited on pages 233, 241, 249, and 251.)

Axel Belinfante. **JTorX: Exploring Model-Based Testing**. PhD thesis, University of Twente, 2014. (Cited on pages 193, 240, 241, 248, 249, 250, 251, 321, and 334.)

Axel Belinfante, Lars Frantzen, and Christian Schallhart. Tools for Test Case Generation. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, **Model-Based Testing of Reactive Systems**, volume 3472 of **Lecture Notes in Computer Science**, pages 391–438. Springer Berlin / Heidelberg, 2005. doi: 10.1007/11498490_18. (Cited on pages 247 and 374.)

Anton Belov, Daniel Diepold, Marijn JH Heule, and Matti Järvisalo. SAT COMPETITION 2014. 2014. (Cited on page 27.)

Jon Louis Bentley. Multidimensional Divide-and-Conquer. **Commun. ACM**, 23(4): 214–229, 1980. (Cited on page 54.)

B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie. **Systems and Software Verification: Model-Checking Techniques and Tools**. Springer, 2001. (Cited on page 240.)

Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. **Systems and software verification: model-checking techniques and tools**. Springer Science & Business Media, 2013. (Cited on pages 108 and 240.)

FI Berg. Model checking LLVM IR using LTSmin: using relaxed memory model semantics. 2014. (Cited on pages 51, 113, and 172.)

Donald J. Berndt and Alison Watkins. High Volume Software Testing using Genetic Algorithms. **Proceedings of the 38th Annual Hawaii International Conference on System Sciences**, 2005. URL `http://dblp.uni-trier.de/db/conf/hicss/hicss2005-9.html#BerndtW05`. (Cited on page 274.)

Antonia Bertolino, Lars Frantzen, Andrea Polini, and Jan Tretmans. Audition of Web Services for Testing Conformance to Open Specified Protocols. In Ralf H. Reussner, Judith A. Stafford, and Clemens A. Szyperski, editors, **Architecting Systems with Trustworthy Components**, volume 3938 of **Lecture Notes in Computer Science**, pages 1–25. Springer, 2004. ISBN 3-540-35800-5. URL `http://dblp.uni-trier.de/db/conf/dagstuhl/trust2004.html#BertolinoFPT04`. (Cited on page 252.)

Antonia Bertolino, Guglielmo De Angelis, Lars Frantzen, and Andrea Polini. The PLASTIC Framework and Tools for Testing Service-Oriented Applications. In Andrea De Lucia and Filomena Ferrucci, editors, **ISSSE**, volume 5413 of **Lecture Notes in Computer Science**, pages 106–139. Springer, 2008. ISBN 978-3-540-95887-1. URL `http://dblp.uni-trier.de/db/conf/issse/issse2008.html#BertolinoAFP08`. (Cited on page 252.)

Nathalie Bertrand, Thierry Jéron, Amélie Stainer, and MEZ Krichen. Off-line test selection with test purposes for non-deterministic timed automata. **Logical Methods in Computer Science**, 8(4), 2012. (Cited on page 240.)

Lorenzo Bettini. **Implementing Domain-Specific Languages with Xtext and Xtend**. Packt Publishing Ltd, 2013. ISBN 9781782160311. URL `http://books.google.de/books?id=ifaBCqGka2IC`. (Cited on page 233.)

Dirk Beyer. Second Competition on Software Verification - (Summary of SV-COMP 2013). In Nir Piterman and Scott A. Smolka, editors, **TACAS**, volume 7795 of **Lecture Notes in Computer Science**, pages 594–609. Springer, 2013. ISBN 978-3-642-36741-0. URL `http://dblp.uni-trier.de/db/conf/tacas/tacas2013.html#Beyer13`. (Cited on pages 172 and 178.)

Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: a technique to pass information between verifiers. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, **SIGSOFT FSE**, page 57. ACM, 2012. ISBN 978-1-4503-0443-6. URL `http://dblp.uni-trier.de/db/conf/sigsoft/fse2012.html#BeyerHKW12`. (Cited on page 91.)

Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and Resource Measurement. In Bernd Fischer and Jaco Geldenhuys, editors, **SPIN**, volume 9232 of **Lecture Notes in Computer Science**, pages 160–178. Springer, 2015. ISBN 978-3-319-23403-8. (Cited on page 371.)

Tom Bienmüller, Werner Damm, and Hartmut Wittke. The STATEMATE Verification Environment - Making It Real. In E. Allen Emerson and A. Prasad Sistla, editors, **CAV**, volume 1855 of **Lecture Notes in Computer Science**, pages 561–567. Springer, 2000. ISBN 3-540-67770-4. (Cited on pages vi, 4, and 60.)

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In Rance Cleaveland, editor, **TACAS**, volume 1579 of **Lecture Notes in Computer Science**, pages 193–207. Springer, 1999. ISBN 3-540-65703-7. URL `http://dblp.uni-trier.de/db/conf/tacas/tacas99.html#BiereCCZ99`. (Cited on pages 93, 94, 106, and 377.)

Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. **Handbook of Satisfiability**, volume 185 of **Frontiers in Artificial Intelligence and Applications**. IOS Press, February 2009. ISBN 978-1-58603-929-5. (Cited on pages 31 and 55.)

Stefan Biffl, Aybüke Aurum, Barry Boehm, Hakan Erdogmus, and Paul Grünbacher, editors. **Value-Based Software Engineering**. Springer, Berlin, 2006. (Cited on page 253.)

Robert V. Binder. 2011 Model-based Testing User Survey: Results and Analysis. Technical report, System Verification Associates, 2011. URL `http://robertvbinder.com/wp-content/uploads/rvb-pdf/arts/MBT-User-Survey.pdf`. (Cited on pages 247, 341, and 348.)

Richard S. Bird and Philip Wadler. **Introduction to functional programming.** Prentice Hall International series in computer science. Prentice Hall, 1988. ISBN 978-0-13-484197-7. (Cited on page 54.)

Rex Black, Dorothy Graham, and Evans Van Veenendaal. **Foundations of Software Testing: Istqb Certification**. Cengage Learning College, 2012. ISBN 9781408044056. (Cited on page 20.)

Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal Versus Agile: Survival of the Fittest. **Computer**, 42(9):37–45, 2009. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2009.284. (Cited on page 348.)

Erik-Oliver Blaß, Joachim Wilke, and Martina Zitterbart. Relaxed Authenticity for Data Aggregation in Wireless Sensor Networks. In **4th International Conference on Security and Privacy in Communication Networks (SecureComm 2008)**, Istanbul, Turkey, September 2008. 4th International Conference on Security and Privacy in Communication Networks (SecureComm 2008). (Cited on page 161.)

Joshua Bloch. How to design a good API and why it matters. In Peri L. Tarr and William R. Cook, editors, **OOPSLA Companion**, pages 506–507. ACM, 2006. ISBN 1-59593-491-X. (Cited on pages 58 and 233.)

Joshua Bloch. **Effective Java Programming Language Guide**. Addison-Wesley, Boston, MA, 2. edition, 2008. ISBN 978-0-321-35668-0. (Cited on page 54.)

Stefan Blom, Jaco van de Pol, and Michael Weber 0002. LTSmin: Distributed and Symbolic Reachability. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, **CAV**, volume 6174 of **Lecture Notes in Computer Science**, pages 354–359. Springer, 2010. ISBN 978-3-642-14294-9. URL `http://dblp.uni-trier.de/db/conf/cav/cav2010.html#BlomPW10`. (Cited on pages 40, 110, 111, and 177.)

ISTQB (International Software Testing Qualifications Board). Standard glossary of terms used in Software Testing, October 2012. (Cited on pages 13, 16, and 19.)

Barry Boehm. Verifying and validating software requirements and design specifications. **IEEE Software 1 (1)**, pages pp 75–88, 1984. (Cited on pages 17, 18, and 343.)

Hans Boehm, Justin Gottschlich, Victor Luchangco, Maged Michael, Mark Moir, Clark Nelson, Torvald Riegel, Tatiana Shpeisman, and Michael Wong. Transactional Language Constructs for C++. **ISO/IEC JTC1/SC22 (Programming languages and operating systems) WG21 (C++)**, 2012. (Cited on page 51.)

David Bohm. **Causality and chance in modern physics**. Routledge & Kegan Paul, London, 1971. Foreword by Louis de Broglie. (Cited on page 194.)

Prasad Bokil, Priyanka Darke, Ulka Shrotri, and R. Venkatesh. Automatic Test Data Generation for C Programs. In **SSIRI**, pages 359–368. IEEE Computer Society, 2009. ISBN 978-0-7695-3758-0. (Cited on page 175.)

Beate Bollig and Ingo Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. **IEEE Trans. Computers**, 45(9):993–1002, 1996. URL `http://dblp.uni-trier.de/db/journals/tc/tc45.html#BolligW96`. (Cited on page 28.)

Filippo Bonchi, Marcello M. Bonsangue, Georgiana Caltais, Jan J. M. M. Rutten, and Alexandra Silva. Final Semantics for Decorated Traces. **Electr. Notes Theor. Comput. Sci.**, 286:73–86, 2012. URL `http://dblp.uni-trier.de/db/journals/entcs/entcs286.html#BonchiBCRS12`. (Cited on page 65.)

Alexander Borgida, John Mylopoulos, and Harry K. T. Wong. Generalization/Specialization as a Basis for Software Specification. In **On Conceptual Modelling (Intervale)**, pages 87–117, 1982. (Cited on page 61.)

Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. Symmetric Spin. **International Journal on Software Tools for Technology Transfer**, 4(1):92–106, 2002. URL `http://netlib.bell-labs.com/who/dennis/Papers/bdh02a.pdf`. (Cited on pages 102, 109, and 134.)

Patricia Bouyer and François Laroussinie. Model checking timed automata. **Modeling and Verification of Real-Time Systems: Formalisms and Software Tools**, pages 111–140, 2010. (Cited on page 33.)

Howard Bowman and Rodolfo Gómez. **Concurrency theory - calculi and automata for modelling untimed and timed concurrent systems**. Springer, 2006. ISBN 978-1-85233-895-4. (Cited on pages 39, 43, and 60.)

Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In Ed Brinksma and Kim Guldstrand Larsen, editors, **Proceedings of the 14th International Conference on Computer Aided Verification (CAV), 2002**, volume 2404 of **Lecture Notes in Computer Science**, pages 343–348. Springer, 2002. ISBN 3-540-43997-8. URL `http://dblp.uni-trier.de/db/conf/cav/cav2002.html#BozgaGM02`. (Cited on pages 247 and 248.)

Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. Testing web services: A survey. **Department of Computer Science, King's College London, Tech. Rep.**, TR-10-01:1–49, 2010. (Cited on page 53.)

Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In Kousha Etessami and Sriram K. Rajamani, editors, **Proceedings of the 17th International Conference of Computer Aided Verification (CAV), Edinburgh, Scotland, UK**, volume 3576 of **LNCS**, pages 335–349. Springer, July 2005. ISBN 3-540-27231-3. URL `http://dblp.uni-trier.de/db/conf/cav/cav2005.html#BozzanoBCJRRS05`. (Cited on page 31.)

Christian Brandes, Benedikt Eberhardinger, David Faragó, Mario Friske, Baris Güldali, and Andrej Pietschker. Drei Methoden, ein Ziel: Testautomatisierung mit BDD, MBT und KDT im Vergleich. **38. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV), Softwaretechnik-Trends**, 2015. (Cited on pages 8, 60, and 208.)

Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leissa, Christoph Mallon, and Andreas Zwinkau. Simple and Efficient Construction of Static Single Assignment Form. In Ranjit Jhala and Koen De Bosschere, editors, **CC**, volume 7791 of **Lecture Notes in Computer Science**, pages 102–122. Springer, 2013. ISBN 978-3-642-37050-2. URL `http://dblp.uni-trier.de/db/conf/cc/cc2013.html#BraunBHLMZ13`. (Cited on page 157.)

Lubos Brim, Ivana Cerná, Pavel Moravec, and Jirí Simsa. Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In Alan J. Hu and Andrew K. Martin, editors, **FMCAD**, volume 3312 of **Lecture Notes in Computer Science**, pages 352–366. Springer, 2004. (Cited on page 112.)

Eckard Bringmann and Andreas Krämer. Model-Based Testing of Automotive Systems. In **ICST**, pages 485–493. IEEE Computer Society, 2008. URL `http://dblp.uni-trier.de/db/conf/icst/icst2008.html#BringmannK08`. (Cited on page 237.)

Ed Brinksma and Jan Tretmans. Testing Transition Systems: An Annotated Bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, **MOVEP**, volume 2067 of **Lecture Notes in Computer Science**, pages 187–195. Springer, 2000. ISBN 3-540-42787-2. URL `http://dblp.uni-trier.de/db/conf/movep/movep2000.html#BrinksmaT00`. (Cited on pages 183 and 186.)

Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible Debugging Software - Quantify the time and cost saved using reversible debuggers. **Citeseer**, 2013. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.370.9611&rep=rep1&type=pdf`. (Cited on page 1.)

D. Brook. Improving Embedded Software Test Effectiveness in Automotive Applications. **Embedded Systems Europe**, 8(55):16–17, February 2004. (Cited on page 2.)

Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. **Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]**, volume 3472 of **Lecture Notes in Computer Science**, 2005. Springer. ISBN 3-540-26278-4. (Cited on page 241.)

Achim D. Brucker and Burkhart Wolff. On theorem prover-based testing. **Formal Aspects of Computing**, 25(5):683–721, 2013. URL `http://dblp.uni-trier.de/db/journals/fac/fac25.html#BruckerW13`. (Cited on page 241.)

Robert Brummayer and Armin Biere. Lemmas on Demand for the Extensional Theory of Arrays. **JSAT**, 6(1-3):165–201, 2009. URL `http://dblp.uni-trier.de/db/journals/jsat/jsat6.html#BrummayerB09`. (Cited on page 55.)

Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. **Computers, IEEE Transactions on**, C-35(8):677–691, 1986. ISSN 0018-9340. doi: 10.1109/TC.1986.1676819. (Cited on page 28.)

Doina Bucur and Marta Kwiatkowska. **Ambient Intelligence**, volume Volume 5859/2009 of **Lecture Notes in Computer Science**, chapter Bug-Free Sensors:

The Automatic Verification of Context-Aware TinyOS Applications, pages 101–105. Springer Berlin / Heidelberg, 2009. doi: 10.1007/978-3-642-05408-2. URL `http://www.springerlink.com/content/x314qr7k163xl781`. (Cited on page 177.)

Doina Bucur and Marta Kwiatkowska. Towards Software Verification for TinyOS Applications. In **Proc. 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2010)**, pages 400–401, Computing Laboratory, Oxford University, UK, April 2010. ACM. (Cited on page 177.)

BusinessWire. Conformiq Releases Automated Test Design Solution Utilizing Multi-Core, Parallel and Distributed Computing. July 2009. URL `http://www.businesswire.com/news/home/20090713005432/en/Conformiq-Releases-Automated-Test-Design-Solution-Utilizing#.VgP5Mem5Ogw`. (Cited on page 274.)

Christian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Richard Draves and Robbert van Renesse, editors, **8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings**, pages 209–224. USENIX Association, 2008a. (Cited on pages 61 and 174.)

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. **ACM Trans. Inf. Syst. Secur.**, 12(2), 2008b. (Cited on pages 23, 173, 175, and 287.)

Kenneth L. Calvert, James Griffioen, Amit Sehgal, and Su Wen. Concast: Design and implementation of a new network service. In **Proceedings of 1999 International Conference on Network Protocols**, 1999. (Cited on page 156.)

Georg Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. II. **Mathematische Annalen**, 49(2), 1897. (Cited on page 10.)

Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. FocalTest: A Constraint Programming Approach for Property-Based Testing. In **Software and Data Technologies**, pages 140–155. Springer, 2013. (Cited on page 241.)

Nicholas Carreiro and David Gelernter. How to Write Parallel Programs: a guide to the perplexed. **ACM Computing Surveys**, September 1989. (Cited on page 273.)

Ivana Cerná and Radek Pelánek. Distributed Explicit Fair Cycle Detection (Set Based Approach). In Thomas Ball and Sriram K. Rajamani, editors, **SPIN**, volume 2648 of **Lecture Notes in Computer Science**, pages 49–73. Springer, 2003. ISBN 3-540-40117-2. (Cited on page 144.)

W. K. Chan, Lijun Mei, and Zhenyu Zhang. Modeling and testing of cloud applications. In Markus Kirchberg, Patrick C. K. Hung, Barbara Carminati, Chi-Hung Chi, Rajaraman Kanagasabai, Emanuele Della Valle, Kun-Chan Lan, and Ling-Jyh Chen, editors, **Proceedings of the Asia-Pacific Services Computing Conference (APSCC), 2012**, pages 111–118. IEEE, 2009. ISBN 978-1-4244-5336-8. URL

http://dblp.uni-trier.de/db/conf/apscc/apscc2009.html#ChanMZ09. (Cited on page 279.)

John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. **Software Engineering Journal**, 9(5), 1994. (Cited on page 21.)

Martin James Chorley. Performance Comparison of Message Passing and Shared Memory Programming with HPC Benchmarks. Master's thesis, University of Edinburgh, August 2007. (Cited on page 50.)

Sheldon J. Chow. What's the Problem with the Frame Problem? **Review of Philosophy and Psychology**, 4(2):309–331, June 2013. (Cited on page 184.)

Alonzo Church. An Unsolvable Problem of Elementary Number Theory. **American Journal of Mathematics**, 58(2):345–363, April 1936. ISSN 00029327. doi: 10.2307/2371045. (Cited on pages 3, 30, and 325.)

Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. **Operating Systems Review**, 43(4):5–10, 2009. URL http://dblp.uni-trier.de/db/journals/sigops/sigops43.html#CiorteaZBCC09. (Cited on page 275.)

Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. On the number and nature of faults found by random testing. **Softw. Test., Verif. Reliab.**, 21(1):3–28, 2011. URL http://dblp.uni-trier.de/db/journals/stvr/stvr21.html#CiupaPOLM11. (Cited on page 241.)

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In **ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming**, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: 10.1145/351240.351266. URL http://dx.doi.org/10.1145/351240.351266. (Cited on pages 3 and 274.)

Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A Symbolic Test Generation Tool. In Joost-Pieter Katoen and Perdita Stevens, editors, **TACAS**, volume 2280 of **Lecture Notes in Computer Science**, pages 470–475. Springer, 2002. (Cited on page 251.)

Edmund M. Clarke and I. Anca Draghicescu. Expressibility results for linear-time and branching-time logics. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, **REX Workshop**, volume 354 of **Lecture Notes in Computer Science**, pages 428–437. Springer, 1988. ISBN 3-540-51080-X. URL http://dblp.uni-trier.de/db/conf/rex/rex88.html#ClarkeD88. (Cited on pages 70, 83, and 85.)

Edmund M. Clarke, Daniel Kroening, Natashaand Sharygina, and Karen Yorav. Predicate Abstraction of ANSI-C Programs Using SAT. **Formal Methods in System Design**, 25:105–127, 1986. URL http://www.kroening.com/papers/fmsd2004.pdf. (Cited on page 102.)

Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another Look at LTL Model Checking. **Formal Methods in System Design**, 10(1):47–71, 1997. (Cited on pages 97 and 101.)

Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State Space Reduction using Partial Order Techniques. **International Journal on Software Tools for Technology Transfer (STTT)**, 2:279–287, 1999a. URL `http://www.dcs.warwick.ac.uk/~doron/ps/sttt98.ps`. (Cited on page 103.)

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. **Model Checking**. The MIT Press, Cambridge, Massachusetts, third printing, 2001 edition, 1999b. (Cited on pages 28, 38, 65, 67, 70, 71, 76, 77, 92, 97, 101, 103, 104, 105, 114, and 157.)

Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, **TACAS**, volume 2988 of **Lecture Notes in Computer Science**, pages 168–176. Springer, 2004a. ISBN 3-540-21299-X. (Cited on page 158.)

Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and Complexity of Bounded Model Checking. In Bernhard Steffen and Giorgio Levi, editors, **VMCAI**, volume 2937 of **Lecture Notes in Computer Science**, pages 85–96. Springer, 2004b. ISBN 3-540-20803-8. (Cited on pages 93, 106, and 107.)

Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Computational challenges in bounded model checking. **STTT**, 7(2):174–183, 2005. (Cited on pages 106 and 107.)

Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. **Commun. ACM**, 52(11):74–84, 2009. (Cited on page 27.)

Cliff Click. A lock-free wait-free hash table, February 2007. URL `http://www.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf`. (Cited on page 143.)

Alistair Cockburn. **Writing Effective Use Cases**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201702258. (Cited on page 346.)

Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. A Practical Verification Methodology for Concurrent Programs. Technical Report MSR-TR-2009-15, Microsoft Research, February 2009. (Cited on pages 173 and 177.)

Stephen A. Cook. The complexity of theorem proving procedures. In **Proceedings of the Third Annual ACM Symposium**, pages 151–158, New York, 1971. ACM. (Cited on page 27.)

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. **Introduction to Algorithms**. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511. (Cited on page 119.)

Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. **Formal Methods**

in System Design, 1(2/3):275–288, 1992. URL http://dblp.uni-trier.de/db/journals/fmsd/fmsd1.html#CourcoubetisVWY92. (Cited on page 99.)

Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. An Overview of the mCRL2 Toolset and Its Recent Advances. In Nir Piterman and Scott A. Smolka, editors, **Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2013**, volume 7795 of **Lecture Notes in Computer Science**, pages 199–213. Springer, 2013. ISBN 978-3-642-36741-0. URL http://dblp.uni-trier.de/db/conf/tacas/tacas2013.html#CranenGKSVWW13. (Cited on pages 43, 110, and 249.)

John W. Creswell. **Research Design: Qualitative, Quantitative, and Mixed Methods Approaches**. Sage Publications Ltd., 4 edition, 2013. (Cited on page 369.)

Lisa Crispin and Janet Gregory. **Agile Testing: A Practical Guide for Testers and Agile Teams**. Addison-Wesley Professional, 2009. ISBN 0321534468, 9780321534460. (Cited on page 346.)

Flaviu Cristian. Understanding Fault-Tolerant Distributed Systems. **Communications of the ACM**, 34(2):56–78, 1993. URL http://dblp.uni-trier.de/db/journals/cacm/cacm34.html#Cristian91. (Cited on pages 43, 142, and 277.)

David Culler, J.P. Singh, and Anoop Gupta. **Parallel Computer Architecture: A Hardware/Software Approach**. Morgan Kaufmann, 1st edition, 1998. ISBN 1558603433. The Morgan Kaufmann Series in Computer Architecture and Design. (Cited on page 49.)

Leonardo Dagum and Rameshm Enon. OpenMP: an industry standard API for shared-memory programming. **Computational Science & Engineering, IEEE**, 5(1): 46–55, 1998. (Cited on page 48.)

Andreas Engelbredt Dalsgaard, Alfons Laarman, Kim G. Larsen, Mads Chr. Olesen, and Jaco van de Pol. Multi-core Reachability for Timed Automata. In Marcin Jurdzinski and Dejan Nickovic, editors, **FORMATS**, volume 7595 of **Lecture Notes in Computer Science**, pages 91–106. Springer, 2012. ISBN 978-3-642-33364-4. URL http://dblp.uni-trier.de/db/conf/formats/formats2012.html#DalsgaardLLOP12. (Cited on pages 110 and 143.)

Mads Dam. CTL* and ECTL* as Fragments of the Modal mu-Calculus. **Theor. Comput. Sci.**, 126(1):77–96, 1994. URL http://dblp.uni-trier.de/db/journals/tcs/tcs126.html#Dam94. (Cited on pages 72, 85, and 86.)

D. Loveland M. Davis. A machine program for theorem proving. In **Communications of the ACM**, volume 5. ACM, 1962. (Cited on page 27.)

Eduardo Cunha de Almeida, Joao Eugenio Marynowski, Gerson Sunyé, Yves Le Traon, and Patrick Valduriez. Efficient Distributed Test Architectures for Large-Scale Systems. In Alexandre Petrenko, Adenilso da Silva Simao, and José Carlos Maldonado,

editors, **Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems (ICTSS'10)**, volume 6435 of **Lecture Notes in Computer Science**, pages 174–187. Springer, 2010. ISBN 978-3-642-16572-6. URL `http://dblp.uni-trier.de/db/conf/pts/ictss2010.html#AlmeidaMSTV10`. (Cited on pages 273 and 279.)

Leonardo Mendonca de Moura and Nikolaj Bjorner. Model-based Theory Combination. **Electr. Notes Theor. Comput. Sci.**, 198(2):37–49, 2008a. URL `http://dblp.uni-trier.de/db/journals/entcs/entcs198.html#MouraB08`. (Cited on page 31.)

Leonardo Mendonca de Moura and Nikolaj Bjorner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, **Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary**, volume 4963 of **Lecture Notes in Computer Science**, pages 337–340. Springer, March 2008b. ISBN 978-3-540-78799-0. URL `http://dblp.uni-trier.de/db/conf/tacas/tacas2008.html#MouraB08`. (Cited on page 32.)

Frederico de Oliveira Jr, Ricardo Lima, Márcio Cornélio, Sérgio Soares, Paulo Maciel, Raimundo Barreto, Meuse Oliveira Jr, and Eduardo Tavares. CML: C modeling language. **Journal of Universal Computer Science**, 13(6):682–700, 2007. (Cited on page 39.)

René G de Vries and Jan Tretmans. Towards formal test purposes. **Formal Approaches to Testing of Software (Fates), 2001**, 2001. (Cited on pages 248 and 287.)

René G. de Vries, Axel Belinfante, and Jan Feenstra. Automated Testing in Practice: The Highway Tolling System. In Ina Schieferdecker, Hartmut König, and Adam Wolisz, editors, **TestCom**, volume 210 of **IFIP Conference Proceedings**, pages 219–234. Kluwer, 2002. ISBN 0-7923-7695-1. (Cited on pages 245, 321, and 345.)

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. **Proceedings of the 6th conference on Operating Systems Design and Implementation (OSDI 04)**, 2004. (Cited on page 52.)

S. Deering and R. Hinden. **RFC 2460 Internet Protocol, Version 6 (IPv6) Specification**. Internet Engineering Task Force, December 1998. URL `http://tools.ietf.org/html/rfc2460`. (Cited on pages 52 and 53.)

Stephen E. Deering. Host Extensions for IP Multicasting. Internet RFC 1112, August 1989. URL `http://www.ietf.org/rfc/rfc1112`. (Cited on pages 52 and 53.)

Stéphane Demri and Paul Gastin. Specification and Verification using Temporal Logics. In **Modern Applications of Automata Theory**, pages 457–494. 2012. (Cited on page 99.)

Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyronnet. Coverage-Biased Random Exploration of Large Models and Application to Testing. Technical Report 1494, LRI, Université Paris-Sud XI, June 2008. 26 pages, submitted. (Cited on pages 321, 357, and 374.)

D Dennett. Cognitive Wheels: The Frame Problem in Artificial Intelligence. **Minds, machines and evolution**, 1984. (Cited on page 184.)

Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. Automated Unique Input Output Sequence Generation for Conformance Testing of FSMs. **Comput. J.**, 49(3):331–344, 2006. URL `http://dblp.uni-trier.de/db/journals/cj/cj49.html#DerderianHHG06`. (Cited on pages 23 and 287.)

Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanc Muslu, and Todd W. Schiller. Building and using pluggable type-checkers. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, **ICSE**, pages 681–690. ACM, 2011. ISBN 978-1-4503-0445-0. URL `http://dblp.uni-trier.de/db/conf/icse/icse2011.html#DietlDEMS11`. (Cited on page 54.)

Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. **Communications of the ACM**, 18(8):453–457, 1975. (Cited on page 43.)

Edsger Wybe Dijkstra. Notes on Structured Programming. EWD249 – circulated privately – `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF` – geprüft: 12. August 2002, April 1970. (Cited on page 13.)

Committee: SC-190 DO-278 Plenary. DO-278 Guidelines For Communication, Navigation, Surveillance,and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance. Technical report, RTCA, 2002. (Cited on page 4.)

Committee: SC-205 DO-330 Plenary. DO-330 Software Tool Qualification Considerations. Technical report, RTCA, 2011. (Cited on pages 4 and 237.)

SC-205/WG-71 DO-333 Plenary. DO-333: Formal Methods Supplement to DO-178C and DO-278A. Technical report, December 2011. (Cited on pages 2 and 4.)

Special C. of RTCA DO178C Plenary. DO-178C, Software Considerations in Airborne Systems and Equipment Certification. Technical report, RTCA, 2011. (Cited on pages vi, 3, 4, 17, 21, 22, and 237.)

Danny Dolev, Maria M. Klawe, and Michael Rodeh. An O(n log n) Unidirectional Distributed Algorithm for Extrema Finding in a Circle. **J. Algorithms**, 3(3):245–260, 1982. URL `http://dblp.uni-trier.de/db/journals/jal/jal3.html#DolevKR82`. (Cited on page 116.)

Yifei Dong, Xiaoqun Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky, Eugene W. Stark, and David S. Warren. Fighting livelock in the i-protocol: a comparative study of verification tools. In **TACAS'99, LNCS**, pages 74–88. Springer, 1999. (Cited on pages 116 and 130.)

Claire Dross, Jean-Christophe Filliâtre, and Yannick Moy. Correct Code Containing Containers. In Martin Gogolla and Burkhart Wolff, editors, **Tests And Proofs**, volume 6706 of **Lecture Notes in Computer Science**, pages 102–118. Springer, 2011. ISBN 978-3-642-21767-8. URL `http://dblp.uni-trier.de/db/conf/tap/tap2011.html#DrossFM11`. (Cited on pages vi, 2, 4, and 237.)

Alexandre Duarte, Walfredo Cirne, Francisco Vilar Brasileiro, and Patrícia Machado. GridUnit: software testing on the grid. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, **Proceedings of the 28th International Conference on Software Engineering**, pages 779–782. ACM, 2006. ISBN 1-59593-375-1. URL `http://dblp.uni-trier.de/db/conf/icse/icse2006.html#DuarteCBM06`. (Cited on pages 52 and 273.)

Alexandre Nóbrega Duarte, Walfredo Cirne, Francisco Brasileiro, Patricia Duarte, and L Machado. Using the computational grid to speed up software testing. In **Proceedings of 19th Brazilian symposium on software engineering**. Citeseer, 2005. (Cited on page 273.)

Elena Dubrova. **Fault-tolerant design**. Springer, 2013. (Cited on pages 19, 43, and 204.)

Arnaud Dupuy and Nancy Leveson. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In **Proceedings of the 19th Digital Avionics Systems Conference (DASC)**, volume 1, pages 1B6–1. IEEE, 2000. (Cited on pages 261 and 287.)

Joe W. Duran and Simeon C. Ntafos. An Evaluation of Random Testing. **IEEE Trans. Software Eng.**, 10(4):438–444, 1984. (Cited on pages 287 and 357.)

Paul Duvall, Stephen M. Matyas, and Andrew Glover. **Continuous Integration: Improving Software Quality and Reducing Risk**. Addison-Wesley, Upper Saddle River, NJ, 2007. ISBN 978-0-321-33638-5. (Cited on page 343.)

SC-205/WG-71 ED-218 Plenary. ED-218: Formal Methods Supplement to ED-12C and ED-109A. January 2012. (Cited on pages 2 and 4.)

Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Partial-order reduction and trail improvement in directed model checking. **STTT**, 6(4):277–301, 2004. URL `http://dblp.uni-trier.de/db/journals/sttt/sttt6.html#EdelkampLL04a`. (Cited on page 131.)

Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, **SAT**, volume 2919 of **Lecture Notes in Computer Science**, pages 502–518. Springer, 2003. URL `http://dblp.uni-trier.de/db/conf/sat/sat2003.html#EenS03`. (Cited on pages 27 and 158.)

Ruediger Ehlers. Small witnesses, accepting lassos and winning strategies in omega-automata and games. **CoRR**, abs/1108.0315, 2011. (Cited on page 99.)

Petter L. H. Eide. Quantification and Traceability of Requirements. Technical Report TDT4735 Software Engineering Depth Study, 2005. URL `http://www.idi.ntnu.no/grupper/su/fordypningsprosjekt-2005/eide-fordyp05.pdf`. (Cited on page 17.)

Victor Eijkhout. **Introduction to High Performance Scientific Computing**. Lulu. com, 2014. URL `http://pages.tacc.utexas.edu/~eijkhout/Articles/EijkhoutIntroToHPC.pdf`. (Cited on page 49.)

Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with Temporal Logic on Truncated Paths. In Warren A. Hunt Jr. and Fabio Somenzi, editors, **CAV**, volume 2725 of **Lecture Notes in Computer Science**, pages 27–39. Springer, 2003. ISBN 3-540-40524-0. (Cited on pages 73 and 93.)

Hesham El-Rewini and Mostafa Abd-El-Barr. **Advanced Computer Architecture and Parallel Processing**. Wiley Series on Parallel and Distributed Computing, 2005. (Cited on pages 41, 48, and 49.)

E. Allen Emerson and Edmund M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In J. W. de Bakker and Jan van Leeuwen, editors, **ICALP**, volume 85 of **Lecture Notes in Computer Science**, pages 169–181. Springer, 1980. ISBN 3-540-10003-2. URL `http://dblp.uni-trier.de/db/conf/icalp/icalp80.html#EmersonC80`. (Cited on page 105.)

Christian Engel, Christoph Gladisch, Vladimir Klebanov, and Philipp Rümmer. Integrating Verification and Testing of Object-Oriented Software. In Bernhard Beckert and Reiner Hähnle, editors, **Proceedings of the Second International Conference on Tests and Proofs (TAP), 2008**, volume 4966 of **Lecture Notes in Computer Science**, pages 182–191. Springer, 2008. ISBN 978-3-540-79123-2. URL `http://dblp.uni-trier.de/db/conf/tap/tap2008.html#EngelGKR08`. (Cited on page 241.)

Christian Engel, Eric Jenn, Peter H Schmitt, Rodrigo Coutinho, Tobias Schoofs, Thales Avionics, and GMV Portugal. Enhanced dispatchability of aircrafts using multi-static configurations. **Proceedings of the Embedded Real Time Software and Systems (ERTS2 2010), Toulouse, France**, 2010. (Cited on page 43.)

André Engels, Loe M. G. Feijs, and Sjouke Mauw. Test Generation for Intelligent Networks Using Model Checking. In Ed Brinksma, editor, **Proceedings of the third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 1997**, volume 1217 of **Lecture Notes in Computer Science**, pages 384–398. Springer, 1997. ISBN 3-540-62790-1. (Cited on page 240.)

EduardP. Enoiu, Adnan Causevic, Thomas J. Ostrand, Elaine J. Weyuker, Daniel Sundmark, and Paul Pettersson. Automated test generation using model checking: an industrial evaluation. **International Journal on Software Tools for Technology Transfer**, pages 1–19, 2014. ISSN 1433-2779. doi: 10.1007/s10009-014-0355-9. (Cited on page 270.)

Michael D. Ernst, Matthew M. Papi, Mahmood Ali, Telmo Correa, and Werner M. Dietl. **The Checker Framework: Custom pluggable types for Java**, 2011. (Cited on pages 4 and 328.)

ETSI, European Telecommunications Standards Institute. ES 202 951. Publication 1.1.1, European Telecommunications Standards Institute, July 2011.

URL http://www.etsi.org/deliver/etsi_es/202900_202999/202951/01.01.01_60/es_202951v010101p.pdf. (Cited on pages 38, 39, 195, and 237.)

S. Evangelista and C. Pajault. Solving the Ignoring Problem for Partial Order Reduction. **STTF**, 12:155–170, 2010. ISSN 1433-2779. doi: 10.1007/s10009-010-0137-y. URL http://dx.doi.org/10.1007/s10009-010-0137-y. (Cited on pages 104, 137, and 153.)

Sami Evangelista, Alfons Laarman, Laure Petrucci, and Jaco van de Pol. Improved Multi-Core Nested Depth-First Search. In Supratik Chakraborty and Madhavan Mukund, editors, **ATVA**, volume 7561 of **Lecture Notes in Computer Science**, pages 269–283. Springer, 2012. ISBN 978-3-642-33385-9. URL http://dblp.uni-trier.de/db/conf/atva/atva2012.html#EvangelistaLPP12. (Cited on pages 99, 111, 138, 144, 145, and 147.)

Vance Faber, Olaf M. Lubeck, and Andrew B. White Jr. Superlinear speedup of an efficient sequential algorithm is not possible. **Parallel Computing**, 3(3):259–260, 1986. (Cited on page 151.)

Stephan Falke, Florian Merz, and Carsten Sinz. The bounded model checker LLBMC. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, **ASE**, pages 706–709. IEEE, 2013a. (Cited on pages 172, 177, 179, and 377.)

Stephan Falke, Florian Merz, and Carsten Sinz. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM - (Competition Contribution). In Nir Piterman and Scott A. Smolka, editors, **TACAS**, volume 7795 of **Lecture Notes in Computer Science**, pages 623–626. Springer, 2013b. ISBN 978-3-642-36741-0. URL http://dblp.uni-trier.de/db/conf/tacas/tacas2013.html#FalkeMS13. (Cited on page 172.)

Stephan Falke, Florian Merz, and Carsten Sinz. Extending the Theory of Arrays: memset, memcpy, and Beyond. In Ernie Cohen and Andrey Rybalchenko, editors, **VSTTE**, volume 8164 of **Lecture Notes in Computer Science**, pages 108–128. Springer, 2013c. ISBN 978-3-642-54107-0. (Cited on page 55.)

David Faragó. Model checking of randomized leader election algorithms. Master's thesis, Universität Karlsruhe, 2007. (Cited on pages 44, 99, 102, 115, 116, 134, 135, 153, and 168.)

David Faragó. Model-based Testing in Agile Software Development. In **30. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV)**, Softwaretechnik-Trends, 2010a. (Cited on pages 7 and 341.)

David Faragó. Coverage Criteria for Nondeterministic Systems. **testing experience, The Magazine for Professional Testers**, pages 104–106, September 2010b. ISSN 1866-5705. (Cited on page 7.)

David Faragó. Improved Underspecification for Model-based Testing in Agile Development. In Stefan Gruner and Bernhard Rumpe, editors, **FM+AM 2010 - Second International Workshop on Formal Methods and Agile Methods, 17**

September 2010, Pisa (Italy), volume 179 of **LNI**, pages 63–78. GI, 2010. ISBN 978-3-88579-273-4. (Cited on pages 7, 60, 341, 342, and 347.)

David Faragó. Nondeterministic Coverage Metrics as Key Performance Indicator for Model- and Value-based Testing. In **31. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV)**, Softwaretechnik-Trends, 2011. (Cited on pages 7, 24, 256, 290, and 303.)

David Faragó and Peter H. Schmitt. Improving Non-Progress Cycle Checks. In **Model Checking Software, 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings**, LNCS, pages 50–67. Springer, June 2009. (Cited on pages 7, 115, 122, 137, and 153.)

David Faragó, Stephan Weißleder, Baris Güldali, Michael Mlynarski, Arne-Michael Törsel, and Christian Brandes. Wirtschaftlichkeitsberechnung für MBT: Wann sich modellbasiertes Testen lohnt. **OBJEKTspektrum**, 4:32–39, 2013. (Cited on pages v, 1, 8, 17, 208, 263, and 341.)

David Faragó, Florian Merz, and Carsten Sinz. Automatic Heavy-weight Static Analysis Tools for Finding Bugs in Safety-critical Embedded C/C++ Code. **36. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV), Softwaretechnik-Trends**, 2014. (Cited on pages 2, 8, and 172.)

Berndt Farwer. omega-Automata. In Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors, **Automata, Logics, and Infinite Games**, volume 2500 of **Lecture Notes in Computer Science**, pages 3–20. Springer, 2001. ISBN 3-540-00388-6. URL `http://dblp.uni-trier.de/db/conf/dagstuhl/automata2001.html#Farwer01`. (Cited on page 77.)

Karl-Filip Faxén. Wool-A work stealing library. **SIGARCH Computer Architecture News**, 36(5):93–100, 2008. (Cited on page 57.)

Loe M. G. Feijs, Nicolae Goga, Sjouke Mauw, and Jan Tretmans. Test Selection, Trace Distance and Heuristics. In Ina Schieferdecker, Hartmut König, and Adam Wolisz, editors, **TestCom**, volume 210 of **IFIP Conference Proceedings**, pages 267–282. Kluwer, 2002. ISBN 0-7923-7695-1. (Cited on pages 24, 288, and 317.)

Norman E. Fenton. **Software metrics - a rigorous approach.** Chapman and Hall, 1991. ISBN 978-0-412-40440-5. (Cited on page 19.)

Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-Free Algorithms Can Be Practically Wait-Free. In Pierre Fraigniaud, editor, **DISC**, volume 3724 of **Lecture Notes in Computer Science**, pages 78–92. Springer, 2005. ISBN 3-540-29163-6. URL `http://dblp.uni-trier.de/db/conf/wdag/disc2005.html#FichLMS05`. (Cited on page 142.)

George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. **ACM SIGSOFT Software Engineering Notes**, 22(4):74–80, 1997. (Cited on pages 3 and 274.)

Marcus S. Fisher. **Software verification and validation - an engineering and scientific approach.** Springer, 2007. ISBN 978-0-387-32725-9. (Cited on page 19.)

Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, and Zijiang Yang. Is There a Best Symbolic Cycle-Detection Algorithm? In Tiziana Margaria and Wang Yi, editors, **TACAS**, volume 2031 of **Lecture Notes in Computer Science**, pages 420–434. Springer, 2001. ISBN 3-540-41865-2. (Cited on pages 99 and 112.)

Melvin Fitting and Richard L. Mendelsohn. **First-Order Modal Logic**. Springer, 1999. (Cited on page 72.)

Cormac Flanagan and Shaz Qadeer. Thread-Modular Model Checking. In Thomas Ball and Sriram K. Rajamani, editors, **SPIN**, volume 2648 of **Lecture Notes in Computer Science**, pages 213–224. Springer, 2003. ISBN 3-540-40117-2. URL `http://dblp.uni-trier.de/db/conf/spin/spin2003.html#FlanaganQ03`. (Cited on page 96.)

R. W. Floyd. Assigning meanings to programs. **Mathematical aspects of computer science**, 19(19-32):1, 1967. (Cited on page 14.)

Michael J. Flynn. Some Computer Organizations and Their Effectiveness. **IEEE Transactions on Computers**, C-21(9):948–960, September 1972. (Cited on page 48.)

J. A. Fodor. Modules, Frames, Fridgeons, Sleeping Dogs and the Music of the Spheres. In Z. W. Pylyshyn, editor, **The Robot's Dilemma**, pages 139–150. Ablex, Norwood, NJ., 1987. (Cited on page 184.)

M. Fowler and J. Highsmith. The Agile Manifesto. In Software Development, Issue on Agile Methodologies, `http://www.sdmagazine.com`, last accessed on March 8th, 2006, August 2001. (Cited on page 342.)

Martin Fowler. **UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)**. The Addison-Wesley Object Technology Series. Addison-Wesley Professional, 3 edition, 2003. ISBN 0321193687. (Cited on page 252.)

Martin Fowler. Mocks aren't Stubs. 2007. URL `http://martinfowler.com/articles/mocksArentStubs.html`. (Cited on page 18.)

Martin Fowler. **Domain-Specific Languages**. Addison-Wesley Professional, 2010. ISBN 0321712943. (Cited on page 233.)

Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. **Journal of Systems and Software**, 38 (3):235–253, 1997. (Cited on pages 22, 23, 24, 179, and 287.)

Lars Frantzen. STSimulator - A Library to Simulate Symbolic Transition Systems, October 2007. (Cited on pages 233 and 325.)

Lars Frantzen. **Theory and Tools for Symbolic Model-Based Testing**. PhD thesis, Technische Universiteit Eindhoven, 2016. URL `mailto:lars@frantzen.info`. (Cited on pages 40, 183, 185, 186, 193, 208, 225, 227, 228, 233, 234, 235, 251, 252, 279, and 325.)

Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A Symbolic Framework for Model-Based Testing. In Klaus Havelund, Manuel Nunez, Grigore Rosu, and Burkhart Wolff, editors, **FATES/RV**, volume 4262 of **Lecture Notes in Computer Science**, pages 40–54. Springer, 2006. ISBN 3-540-49699-8. URL `http://dblp.uni-trier.de/db/conf/fates/fates2006.html#FrantzenTW06`. (Cited on pages 40 and 234.)

Lars Frantzen, Maria Las Nieves Huerta, Zsolt G. Kiss, and Thomas Wallet. On-The-Fly Model-Based Testing of Web Services with Jambition. **WS-FM 2008**, pages 143–157, 2009. doi: http://dx.doi.org/10.1007/978-3-642-01364-5_9. URL `http://dx.doi.org/http://dx.doi.org/10.1007/978-3-642-01364-5_9`. (Cited on page 252.)

Marc Frappier, Benoit Fraikin, Romain Chossart, Raphael Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of Model Checking Tools for Information Systems. In Jin Song Dong and Huibiao Zhu, editors, **ICFEM**, volume 6447 of **Lecture Notes in Computer Science**, pages 581–596. Springer, 2010. ISBN 978-3-642-16900-7. (Cited on pages 38 and 108.)

Gordon Fraser. **Automated Software Testing with Model Checkers**. PhD thesis, 2007. (Cited on page 270.)

Gordon Fraser and Andrea Arcuri. Evolutionary Generation of Whole Test Suites. In Manuel Nú nez, Robert M. Hierons, and Mercedes G. Merayo, editors, **11th International Conference on Quality Software (QSIC), 2011**, pages 31–40. IEEE Computer Society, 2011. URL `http://dblp.uni-trier.de/db/conf/qsic/qsic2011.html#FraserA11`. (Cited on page 288.)

Gordon Fraser and Franz Wotawa. Property relevant software testing with model-checkers. **ACM SIGSOFT Software Engineering Notes**, 31(6):1–10, 2006. (Cited on pages 14, 22, 23, 73, 93, 240, 253, 261, 270, 271, 287, and 333.)

Gordon Fraser and Franz Wotawa. Test-Case Generation and Coverage Analysis for Nondeterministic Systems Using Model-Checkers. In **Proceedings of the Second International Conference on Software Engineering Advances (ICSEA), 2007**, page 45. IEEE Computer Society, 2007. URL `http://dblp.uni-trier.de/db/conf/icsea/icsea2007.html#FraserW07`. (Cited on pages 240, 270, 271, 272, 290, and 377.)

Gordon Fraser, Franz Wotawa, and Paul Ammann. Issues in using model checkers for test case generation. **Journal of Systems and Software**, 82(9):1403–1418, 2009. (Cited on pages 13, 23, 59, 195, 208, 228, 240, 253, 259, 262, 270, 271, 272, 287, and 289.)

Mario Friske, Bernd-Holger Schlingloff, and Stephan Weißleder. Composition of Model-based Test Coverage Criteria. In Holger Giese, Michaela Huhn, Ulrich Nickel, and Bernhard Schätz, editors, **Tagungsband Modellbasierte Entwicklung eingebetteter Systeme IV, Schloss Dagstuhl, Germany (Dagstuhl-Workshop MBEES)**, volume 2008-2 of **Informatik-Bericht**, pages 87–94. TU Braunschweig, Institut für Software Systems Engineering, April 2008. URL `http:`

//dblp.uni-trier.de/db/conf/mbees/mbees2008.html#FriskeSW08. (Cited on page 22.)

M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. **Formal Methods in System Design**, 10(2-3):149–169, 1997. URL http://www.springerlink.com/index/PN0805U9102624R2.pdf. (Cited on page 113.)

Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, Reading, MA, 1995. ISBN 978-0-201-63361-0. (Cited on page 329.)

Hubert Garavel. OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, **Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), 1998**, volume 1384 of **Lecture Notes in Computer Science**, pages 68–84. Springer, 1998. ISBN 3-540-64356-7. (Cited on pages 39, 110, and 248.)

Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, **TACAS**, volume 6605 of **Lecture Notes in Computer Science**, pages 372–387. Springer, 2011. ISBN 978-3-642-19834-2. (Cited on pages 39 and 110.)

Saul I. Gass and Michael C. Fu, editors. **Encyclopedia of operations research and management science**, volume 1 and 2. Springer, 3rd edition, 2013. ISBN 978-1-4419-1137-7. (Cited on page 337.)

Paul Gastin and Pierre Moro. Minimal Counter-example Generation for SPIN. In Dragan Bosnacki and Stefan Edelkamp, editors, **SPIN**, volume 4595 of **Lecture Notes in Computer Science**, pages 24–38. Springer, 2007. ISBN 978-3-540-73369-0. URL http://dblp.uni-trier.de/db/conf/spin/spin2007.html#GastinM07. (Cited on page 131.)

Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, **CAV**, volume 2102 of **Lecture Notes in Computer Science**, pages 53–65. Springer, 2001. ISBN 3-540-42345-1. URL http://dblp.uni-trier.de/db/conf/cav/cav2001.html#GastinO01. (Cited on page 109.)

Marie-Claude Gaudel. Testing Can Be Formal, Too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, **TAPSOFT**, volume 915 of **Lecture Notes in Computer Science**, pages 82–96. Springer, 1995. ISBN 3-540-59293-8. URL http://dblp.uni-trier.de/db/conf/tapsoft/tapsoft95.html#Gaudel95. (Cited on page 185.)

Vibha Gaur and Anuja Soni. A fuzzy traceability vector model for requirements validation. **IJCAT**, 47(2/3):172–188, 2013. (Cited on page 333.)

Gregory Gay, Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. The Risks of Coverage-Directed Test Case Generation. **Software Engineering, IEEE Transactions on**, PP(99), 2015. ISSN 0098-5589. doi: 10.1109/TSE.2015.2421011. (Cited on pages 20, 21, 23, 256, 259, 260, 261, 270, 287, 348, 349, 371, and 374.)

Ian P. Gent. The Recomputation Manifesto. **CoRR**, abs/1304.3674, 2013. (Cited on page 354.)

Linda Di Geronimo, Filomena Ferrucci, Alfonso Murolo, and Federica Sarro. A Parallel Genetic Algorithm Based on Hadoop MapReduce for the Automatic Generation of JUnit Test Suites. **Fifth International Conference on Software Testing, Verification and Validation (ICST), 2012**, pages 785–793, 2012. URL `http://dblp.uni-trier.de/db/conf/icst/icst2012.html#GeronimoFMS12`. (Cited on pages 272, 275, and 288.)

Jason Ghidella and Pieter J. Mosterman. Requirements-based testing in aircraft control design. In **in Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit 2005**, pages 2005–5886, 2005. (Cited on pages 14 and 22.)

Dimitra Giannakopoulou and Klaus Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In **ASE**, pages 412–416. IEEE Computer Society, 2001. ISBN 0-7695-1426-X. URL `http://dblp.uni-trier.de/db/conf/kbse/ase2001.html#GiannakopoulouH01`. (Cited on pages 14 and 81.)

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. **ACM SIGACT News**, 33(2):51–59, 2002. (Cited on pages 277 and 278.)

Fred Glover. Tabu search – part I. **ORSA Journal on Computing**, 1(3):190 – 206, 1989. (Cited on page 337.)

S. Gnesi and T. Margaria. **Formal Methods for Industrial Critical Systems: A Survey of Applications**. Wiley, 2012. ISBN 9781118459874. URL `http://books.google.de/books?id=OI-RVMr_6ssC`. (Cited on pages vi, 3, and 4.)

Patrice Godefroid. **Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem**, volume 1032 of **Lecture Notes in Computer Science**. Springer, 1996. ISBN 3-540-60761-7. (Cited on page 103.)

Patrice Godefroid. Model Checking for Programming Languages using Verisoft. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, **POPL**, pages 174–186. ACM Press, 1997. ISBN 0-89791-853-3. URL `http://dblp.uni-trier.de/db/conf/popl/popl97.html#Godefroid97`. (Cited on page 173.)

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In **PLDI**, pages 213–223. ACM, 2005. (Cited on pages 23, 61, 173, 174, and 287.)

Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. **Java Concurrency in Practice**. Addison-Wesley Longman, Amsterdam, 2006. ISBN 0321349601. URL `http://www.amazon.de/Java-Concurrency-Practice-Brian-Goetz/dp/0321349601`. (Cited on page 48.)

Nicolae Goga. Comparing TorX, Autolink, TGV and UIO Test Algorithms. In Rick Reed and Jeanne Reed, editors, **SDL Forum**, volume 2078 of **Lecture Notes in Computer Science**, pages 379–402. Springer, 2001. ISBN 3-540-42281-1. URL `http://dblp.uni-trier.de/db/conf/sdl/sdl2001.html#Goga01`. (Cited on page 247.)

Nicolae Goga. Implementing heuristic testing selection. **WSEAS Transactions on Communications**, 2(4):398–403, 2003. URL `http://repository.tue.nl/624026`. (Cited on pages 248, 288, and 317.)

Seth Copen Goldstein, Klaus E. Schauser, and David E. Culler. Lazy Threads: Implementing a Fast Parallel Call. **J. Parallel Distrib. Comput.**, 37(1):5–20, 1996. (Cited on page 57.)

Daniele Gorla. Comparing communication primitives via their relative expressive power. **Inf. Comput.**, 206(8):931–952, 2008. (Cited on pages 41 and 42.)

Daniele Gorla. A taxonomy of process calculi for distribution and mobility. **Distributed Computing**, 23(4):273–299, 2010. (Cited on pages 41 and 42.)

Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan I. Maletic, and Patrick Mäder. Traceability Fundamentals. In Jane Cleland-Huang, Olly Gotel, and Andrea Zisman, editors, **Software and Systems Traceability**, pages 3–22. Springer, 2012. ISBN 978-1-4471-2238-8. (Cited on pages 17 and 19.)

Jens Grabowski, Victor V. Kuliamin, Alain-Georges Vouffo Feudjio, Antal Wu-Hen-Chang, and Milan Zoric. Towards the Usage of MBT at ETSI. In Alexander K. Petrenko and Holger Schlingloff, editors, **Proceedings of the 8th Workshop on Model-Based Testing (MBT), 2013**, volume 111 of **EPTCS**, pages 30–34, 2013. URL `http://dblp.uni-trier.de/db/series/eptcs/eptcs111.html#abs-1303-1007`. (Cited on page 251.)

Dorothy Graham and Mark Fewster. **Experiences of Test Automation: Case Studies of Software Test Automation**. Addison-Wesley Professional, 2012. (Cited on page 16.)

Jim Gray. Why Do Computers Stop and What Can Be Done About It? In **Symposium on Reliability in Distributed Software and Database Systems**, pages 3–12. Los Angeles, CA, USA, IEEE Computer Society, 1986. (Cited on pages 58 and 233.)

R. Graziani. **IPv6 Fundamentals: A Straightforward Approach to Understanding IPv6**. Pearson Education, 2012. ISBN 9780133033472. URL `http://books.google.de/books?id=FbYjJjZNA5gC`. (Cited on page 53.)

Carlos Gregorio-Rodriguez, Luis Llana, and Rafael Martinez-Torres. Extending mCRL2 with ready simulation and iocos input-output conformance simulation. In Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong, editors, **SAC**, pages 1781–1788. ACM, 2015. ISBN 978-1-4503-3196-8. (Cited on page 249.)

Wolfgang Grieskamp, Xiao Qu, Xiangjun Wei, Nicolas Kicillof, and Myra B. Cohen. Interaction Coverage Meets Path Coverage by SMT Constraint Solving. In Manuel Núnez, Paul Baker, and Mercedes G. Merayo, editors, **TestCom/FATES**, volume 5826 of **Lecture Notes in Computer Science**, pages 97–112. Springer, 2009. ISBN 978-3-642-05030-5. (Cited on pages 241 and 334.)

Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and VÃctor A. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. **Softw. Test., Verif. Reliab.**, 21(1):55–71, 2011. URL `http://dblp.uni-trier.de/db/journals/stvr/stvr21.html#GrieskampKSB11`. (Cited on pages 208 and 251.)

Jan Friso Groote and Mohammad Reza Mousavi. **Modeling and Analysis of Communicating Systems**. MIT Press, 2014. (Cited on pages 2, 41, 43, 89, 110, and 249.)

Martin Große-Rhode. On a Reference Model for the Formalization and Integration of Software Specification Languages. In **Current Trends in Theoretical Computer Science**, pages 215–225. 2001. (Cited on page 39.)

IEEE Life Cycle Processes Working Group. ANSI/IEEE 1008-1987 - IEEE Standard for Software Unit Testing. Technical report, 1987. URL `http://s3.amazonaws.com/akitaonrails/files/std1008-1987.pdf`. (Cited on page 13.)

Bernhard J. M. Grün, David Schuler, and Andreas Zeller. The Impact of Equivalent Mutants. In **ICST Workshops**, pages 192–199. IEEE Computer Society, 2009. ISBN 978-0-7695-3671-2. (Cited on pages 23, 24, 179, and 261.)

Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian Partial-Order Reduction. In Dragan Bosnacki and Stefan Edelkamp, editors, **SPIN**, volume 4595 of **Lecture Notes in Computer Science**, pages 95–112. Springer, 2007. ISBN 978-3-540-73369-0. URL `http://dblp.uni-trier.de/db/conf/spin/spin2007.html#GuetaFYS07`. (Cited on page 103.)

Aarti Gupta. From Hardware Verification to Software Verification: Re-use and Re-learn. In Karen Yorav, editor, **Hardware and Software: Verification and Testing, Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23-25, 2007, Proceedings**, volume 4899 of **Lecture Notes in Computer Science**, pages 14–15. Springer, 2008. ISBN 978-3-540-77964-3. (Cited on page 178.)

Yuri Gurevich. Evolving Algebras. In **IFIP Congress (1)**, pages 423–427, 1994. URL `http://dblp.uni-trier.de/db/conf/ifip/ifip94-1.html#Gurevich94`. (Cited on page 251.)

Walter J. Gutjahr. Partition Testing vs. Random Testing: The Influence of Uncertainty. **IEEE Trans. Software Eng.**, 25(5):661–674, 1999. (Cited on pages 23 and 287.)

Georg Hager and Gerhard Wellein. **Introduction to High Performance Computing for Scientists and Engineers.** Chapman and Hall / CRC computational science series. CRC Press, 2011. ISBN 978-1-439-81192-4. (Cited on page 49.)

Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly On-the-Fly LTL Model Checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, **TACAS**, volume 3440 of **Lecture Notes in Computer Science**, pages 191–205. Springer, 2005. ISBN 3-540-25333-5. (Cited on page 56.)

Grégoire Hamon, Leonardo Mendonca de Moura, and John M. Rushby. Generating Efficient Test Sets with a Model Checker. In **Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM), 2004**, pages 261–270. IEEE Computer Society, 2004. ISBN 0-7695-2222-X. URL `http://dblp.uni-trier.de/db/conf/sefm/sefm2004.html#HamonMR04`. (Cited on page 241.)

Grégoire Hamon, Leonardo De Moura, and John Rushby. Automated test generation with SAL. **CSL Technical Note**, page 15, 2005. (Cited on pages 20, 241, and 270.)

Youssef Hanna. SLEDE: lightweight verification of sensor network security protocol implementations. In **ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering**, pages 591–594, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-812-1. doi: http://doi.acm.org/10.1145/1295014.1295050. (Cited on page 176.)

Henri Hansen and Antti Kervinen. Minimal Counterexamples in $O(nlogn)$ Memory and $O(n^2)$ Time. In **ACSD**, pages 133–142. IEEE Computer Society, 2006. ISBN 0-7695-2556-3. URL `http://dblp.uni-trier.de/db/conf/acsd/acsd2006.html#HansenK06`. (Cited on page 131.)

Henri Hansen, Wojciech Penczek, and Antti Valmari. Stuttering-Insensitive Automata for On-the-fly Detection of Livelock Properties. **Electr. Notes Theor. Comput. Sci.**, 66(2):178–193, 2002. (Cited on pages 130 and 131.)

Mark Harman, S.Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. **Department of Computer Science, King's College London, Tech. Rep. TR-09-03**, 2009. (Cited on page 305.)

A. Hartman. Agedis - Model Based Test Generation Tools, Final Project Report. 9Seiten-Version: www.agileconnection.com/article/model-based-test-generation-tools, 2004. (Cited on page 247.)

Kelly J Hayhurst, Dan S Veerhusen, John J Chilenski, and Leanna K Rierson. **A practical tutorial on modified condition/decision coverage**. National Aeronautics and Space Administration, Langley Research Center, 2001. (Cited on pages 18, 21, and 22.)

Mats P.E. Heimdahl and George Devaraj. Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In **Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE), Linz, Austria**, pages 176–185. IEEE Computer Society, September 2004. ISBN 0-7695-2131-2. URL `http://dblp.uni-trier.de/db/conf/kbse/ase2004.html#HeimdahlG04`. (Cited on page 23.)

Mats P.E. Heimdahl, George Devaraj, and Robert Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In **Proceedings of the 8th IEEE High Assurance in Systems Engineering Workshop**, pages 178–186. IEEE Computer Society, 2004. ISBN 0-7695-2094-4. (Cited on pages 23 and 287.)

Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, **OPODIS**, volume 3974 of **Lecture Notes in Computer Science**, pages 3–16. Springer, 2005. ISBN 3-540-36321-1. (Cited on page 57.)

Maurice Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. In Danny Dolev, editor, **PODC**, pages 276–290. ACM, 1988. ISBN 0-89791-277-2. URL `http://dblp.uni-trier.de/db/conf/podc/podc88.html#Herlihy88`. (Cited on page 142.)

Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In David A. Padua, editor, **PPOPP**, pages 197–206. ACM, 1990. ISBN 0-89791-350-7. URL `http://dblp.uni-trier.de/db/conf/ppopp/ppopp90.html#Herlihy90`. (Cited on page 50.)

Maurice Herlihy. Wait-Free Synchronization. **ACM Trans. Program. Lang. Syst.**, 13(1):124–149, 1991. URL `http://dblp.uni-trier.de/db/journals/toplas/toplas13.html#Herlihy91`. (Cited on page 50.)

Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In Alan Jay Smith, editor, **ISCA**, pages 289–300. ACM, 1993. ISBN 0-8186-3810-9. URL `http://dblp.uni-trier.de/db/conf/isca/isca93.html#HerlihyM93`. (Cited on page 50.)

Maurice Herlihy and Nir Shavit. **The art of multiprocessor programming.** Morgan Kaufmann, 2008. ISBN 978-0-12-370591-4. (Cited on pages 48, 49, 50, 51, 57, 138, and 142.)

Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In **ICDCS**, pages 522–529. IEEE Computer Society, 2003. ISBN 0-7695-1920-2. URL `http://dblp.uni-trier.de/db/conf/icdcs/icdcs2003.html#HerlihyLM03`. (Cited on page 142.)

Carl Hewitt. Actor Model of Computation: Scalable Robust Information Systems. Technical Report v24, arXiv:1008.1459, July 2012. URL `http://arxiv.org/abs/1008.1459v24`. cite arxiv:1008.1459Comment: improved syntax. (Cited on page 48.)

Robert M. Hierons, Mercedes G. Merayo, and Manuel Núñez. Implementation relations and test generation for systems with distributed interfaces. **Distributed Computing**, 25(1):35–62, 2012. URL `http://dblp.uni-trier.de/db/journals/dc/dc25.html#HieronsMN12`. (Cited on page 279.)

Martin Hillenbrand. **Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen**. PhD thesis, Karlsruhe Institute of Technology, 2011. (Cited on pages 4 and 19.)

Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and Francois Pennaneac'h. UMLAUT: An Extendible UML Transformation Framework. In **Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE), 1999**, pages 275–278. IEEE Computer Society„ 1999. URL `http://dblp.uni-trier.de/db/conf/kbse/ase1999.html#HoJGP99`. (Cited on page 248.)

C. A. R. Hoare. **Communicating sequential processes**. Prentice-Hall, 1985. (Cited on page 43.)

C. A. R. Hoare. Assertions: A Personal Perspective. **IEEE Annals of the History of Computing**, 25(2):14–25, 2003. URL `http://dblp.uni-trier.de/db/journals/annals/annals25.html#Hoare03`. (Cited on page 15.)

Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. **Comm. ACM**, 12(10):576–580, 583, October 1969. (Cited on pages 3 and 39.)

S. Hölldobler, V. H. Nguyen, J. Stocklina, and P. Steinke. A short overview on modern parallel SAT-solvers. **Proceedings of the International Conference on Advanced Computer Science and Information Systems,**, pages 201–206, 2011. (Cited on page 47.)

Axel Hollmann. **Model-based mutation testing for test generation and adequacy analysis**. PhD thesis, University of Paderborn, 2011. (Cited on page 261.)

C Michael Holloway. Why engineers should consider formal methods. In **Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE**, volume 1, pages 1–3. IEEE, 1997. (Cited on page 2.)

Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In Aarti Gupta and Sharad Malik, editors, **CAV**, volume 5123 of **Lecture Notes in Computer Science**, pages 209–213. Springer, 2008. ISBN 978-3-540-70543-7. (Cited on pages 173 and 175.)

Gerard J. Holzmann. **Design and Validation of Computer Protocols**. Prentice Hall Software Series, 1992. URL `http://spinroot.com/spin/Doc/Book91.html`. (Cited on pages 131, 137, and 240.)

Gerard J. Holzmann. The Engineering of a Model Checker: The Gnu i-Protocol Case Study Revisited. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink,

editors, **SPIN**, volume 1680 of **Lecture Notes in Computer Science**, pages 232–244. Springer, 1999. ISBN 3-540-66499-8. URL http://dblp.uni-trier.de/db/conf/spin/spin1999.html#Holzmann99. (Cited on page 116.)

Gerard J. Holzmann. Economics of software verification. In John Field and Gregor Snelting, editors, **PASTE**, pages 80–85. ACM, 2001. ISBN 1-58113-413-4. (Cited on page 2.)

Gerard J. Holzmann. **The Spin Model Checker: primer and reference manual**. Addison Wesley, first edition, 2004. ISBN 0-321-22862-6. (Cited on pages 44, 45, 46, 71, 76, 95, 100, 102, 107, 114, 118, and 133.)

Gerard J. Holzmann and Doron A. Peled. An improvement in formal verification. In **Proceedings of the Formal Description Techniques 1994, Bern, Switzerland**, pages 197–211. Chapman & Hall, 1994. URL http://www.spinroot.com/spin/Doc/forte94a.pdf. (Cited on pages 100 and 136.)

Gerard J. Holzmann and Margaret H. Schmith. Automating Software Feature Verification. Technical report, Bell Labs Technical Journal, 2000. (Cited on page 176.)

Gerard J. Holzmann, Doron A. Peled, and M. Yannakakis. On nested depth-first search. In **Proceedings of the Second SPIN Workshop. Aug. 1996. Rutgers Univ., New Brunswick, NJ.**, pages 23–32. American Mathematical Society. DIMACS/32, 1996. URL http://www.spinroot.com/spin/Workshops/ws96/Ho.pdf. (Cited on pages 99, 100, 103, 104, 135, and 137.)

Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm Verification Techniques. **IEEE Trans. Software Eng.**, 37(6):845–857, 2011. URL http://dblp.uni-trier.de/db/journals/tse/tse37.html#HolzmannJG11. (Cited on page 111.)

G.J. Holzmann. Parallelizing the Spin Model Checker. In **SPIN**, volume 7385 of **LNCS**, pages 155–171. Springer, 2012. ISBN 978-3-642-31758-3. URL http://dx.doi.org/10.1007/978-3-642-31759-0_12. (Cited on pages 145, 146, and 147.)

J.E. Hopcroft and J.D. Ullman. **Introduction to Automata Theory, Languages and Computation**. Addison-Wesley, Cambridge, 1979. (Cited on pages 10 and 79.)

Joseph Robert Horgan, Saul London, and Michael R. Lyu. Achieving Software Quality with Testing Coverage Measures. **IEEE Computer**, 27(9):60–69, 1994. URL http://dblp.uni-trier.de/db/journals/computer/computer27.html#HorganLL94. (Cited on pages 23 and 287.)

David Hovemeyer and Wiliam Pugh. Finding Bugs is Easy. **ACM SIGPLAN Notices**, 39(12):92–106, 2004. (Cited on page 4.)

Falk Howar, Malte Isberner, Maik Merten, Bernhard Steffen, and Dirk Beyer. The RERS Grey-Box Challenge 2012: Analysis of Event-Condition-Action Systems. In Tiziana Margaria and Bernhard Steffen, editors, **ISoLA (1)**, volume 7609 of **Lecture Notes in Computer Science**, pages 608–614. Springer, 2012. ISBN 978-3-642-34025-3. URL http://dblp.uni-trier.de/db/conf/isola/isola2012-1.html#HowarIMSB12. (Cited on page 109.)

Wen-Ling Huang, Jan Peleska, and Uwe Schulze. COMPASS - Test Automation Support. Technical Report D34.1, FP7 compass-research.eu, 2013. (Cited on page 237.)

Daniel Hughes, Phil Greenwood, and Geoff Coulson. A Framework for Testing Distributed Systems. In Germano Caronni, Nathalie Weiler, and Nahid Shahmehri, editors, **Peer-to-Peer Computing**, pages 262–263. IEEE Computer Society, 2004. ISBN 0-7695-2156-8. URL `http://dblp.uni-trier.de/db/conf/p2p/p2p2004.html#HughesGC04`. (Cited on page 279.)

John Hughes. Why Functional Programming Matters. **Comput. J.**, 32(2):98–107, 1989. (Cited on page 54.)

Antti Huima. Implementing Conformiq Qtronic. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, **TestCom/FATES**, volume 4581 of **Lecture Notes in Computer Science**, pages 1–12. Springer, 2007. ISBN 978-3-540-73065-1. URL `http://dblp.uni-trier.de/db/conf/pts/testcom2007.html#Huima07`. (Cited on pages 186, 246, 251, 252, 253, and 280.)

Andrew Hunt and David Thomas. **The Pragmatic Programmer: From Journeyman to Master**. Addison-Wesley, Harlow, England, 1999. ISBN 978-0-201-61622-4. (Cited on page 214.)

James J. Hunt. The Practical Application of Formal Methods: Where Is the Benefit for Industry? In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, **FoVeOOS**, volume 7421 of **Lecture Notes in Computer Science**, pages 22–32. Springer, 2011. ISBN 978-3-642-31761-3. URL `http://dblp.uni-trier.de/db/conf/foveoos/foveoos2011.html#Hunt11`. (Cited on pages v, vi, 1, 2, 4, 13, and 237.)

Dieter Hutter. Deduction as an Engineering Science. **Electr. Notes Theor. Comput. Sci.**, 86(1):1–8, 2003. URL `http://dblp.uni-trier.de/db/journals/entcs/entcs86.html#Hutter03`. (Cited on page 3.)

65A System aspects IEC 61508 Plenary. IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems (S+ IEC 61508 ed2.0). Technical report, 2010. (Cited on page 4.)

IEEE. **Standard 610.12-1990: Glossary of Software Engineering Terminology**, volume 1, chapter Glossary of Software Engineering Terminology. IEEE Press, 1999. (Cited on page 19.)

Hazelcast Inc. **Hazelcast Documentation**. Hazelcast Inc., 2015. URL `http://hazelcast.org/documentation/`. (Cited on page 278.)

Koray Incki, Ismail Ari, and Hasan Sözer. A Survey of Software Testing in the Cloud. In **SERE (Companion)**, pages 18–23. IEEE, 2012. ISBN 978-0-7695-4743-5. (Cited on page 272.)

Intel. Intel Architecture Instruction Set Extensions Programming Reference. Technical Report 319433-014, Intel, August 2012. URL `http://download-software.intel.com/sites/default/files/319433-014.pdf`. (Cited on page 51.)

Krontiris Ioannis, Tassos Dimitriou, and Felix C. Freiling. Towards intrusion detection in wireless sensor networks. In **Proceedings of the 13th European Wireless Conference**, 2007. (Cited on page 161.)

Syed M. S. Islam, Mohammed H. Sqalli, and Sohel Khan. Modeling and Formal Verification of DHCP Using SPIN. **IJCSA**, 3(2):145–159, 2006. (Cited on page 116.)

ISO Information Technology. ISO/IEC 9646 - Open Systems Interconnection Conformance Testing Methodology and Framework. 1992. (Cited on pages 13 and 183.)

ISO/IEC/IEEE. Systems and software engineering – Life cycle processes –Requirements engineering. **ISO/IEC/IEEE 29148:2011(E)**, pages 1–94, 2011. (Cited on page 18.)

ISTQB. Syllabus Certified Model-based Tester. Technical Report 2.3, ISTQB, May 2013. URL `https://www.isqi.org/tl_files/data/documents/en/BoE_CMBT-FL_Syllabus_V2%203_ISQI_public.pdf`. (Cited on page 237.)

Alon Itai and Michael Rodeh. Symmetry Breaking in Distributive Networks. In **Proceedings of the 22nd Annual IEEE Symposium on Foundations of Computer Science, FOCS'81 (Nashville, TN, October 28-30, 1981)**, pages 150–158, Los Alamitos, Ca., USA, October 1981. IEEE Computer Society Press. URL `http://theory.lcs.mit.edu/~dmjones/FOCS/focs81.html`. (Cited on page 116.)

ITU-T Study Group 10. Z.500 - Framework on formal methods in conformance testing. **INTERNATIONAL TELECOMMUNICATION UNION**, 1997. (Cited on pages 183 and 186.)

Daniel Jackson. **Software Abstractions - Logic, Language, and Analysis.** MIT Press, 2006. ISBN 978-0-262-10114-1. (Cited on page 93.)

R. Janßen. A note on superlinear speedup. **Parallel Computing**, 4(2):211–213, 1987. (Cited on page 151.)

Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. **STTT**, 7(4): 297–315, 2005. URL `http://dblp.uni-trier.de/db/journals/sttt/sttt7.html#JardJ05`. (Cited on pages 187, 193, 196, 240, 246, 247, 253, 286, 287, 289, and 336.)

Claude Jard, Thierry Jéron, and Pierre Morel. Verification of Test Suites. In Hasan Ural, Robert L. Probert, and Gregor von Bochmann, editors, **TestCom**, volume 176 of **IFIP Conference Proceedings**, pages 3–18. Kluwer, 2000. ISBN 0-7923-7921-7. URL `http://dblp.uni-trier.de/db/conf/pts/testcom2000.html#JardJM00`. (Cited on page 15.)

Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The International SAT Solver Competitions. **AI Magazine**, 33(1), 2012. URL `http://dblp.uni-trier.de/db/journals/aim/aim33.html#JarvisaloBRS12`. (Cited on page 27.)

D. Ross Jeffery, Mark Staples, June Andronick, Gerwin Klein, and Toby C. Murray. An empirical research agenda for understanding formal methods productivity. **Information & Software Technology**, 60:102–112, 2015. (Cited on page 2.)

Ronald E. Jeffries, Ann Anderson, Chet Hendrickson, and Chapter Circle Of Life. Extreme Programming Installed, 2000. (Cited on page 342.)

Thierry Jéron, Margus Veanes, and Burkhart Wolff. Symbolic Methods in Testing (Dagstuhl Seminar 13021). **Dagstuhl Reports**, 3(1):1–29, 2013. URL `http://dblp.uni-trier.de/db/journals/dagstuhl-reports/dagstuhl-reports3.html#JeronVW13`. (Cited on pages 248, 250, 251, and 334.)

Yue Jia and Mark Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In **Practice and Research Techniques, 2008. TAIC PART'08. Testing: Academic & Industrial Conference**, pages 94–98. IEEE, 2008. (Cited on pages 23 and 179.)

Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. **IEEE Trans. Software Eng.**, 37(5):649–678, 2011. (Cited on pages 23, 24, and 179.)

Bo Jiang, T. H. Tse, Wolfgang Grieskamp, Nicolas Kicillof, Yiming Cao, Xiang Li, and W. K. Chan. Assuring the model evolution of protocol software specifications by regression testing process improvement. **Softw., Pract. Exper.**, 41 (10):1073–1103, 2011. URL `http://dblp.uni-trier.de/db/journals/spe/spe41.html#JiangTGKCLC11`. (Cited on page 287.)

Mat Johns. **Getting Started with Hazelcast**. Packt Publishing Ltd, 2013. ISBN 9781782167310. URL `http://books.google.de/books?id=tzAa_MxjNrsC`. (Cited on pages 53 and 278.)

Donald B. Johnson. Finding All the Elementary Circuits of a Directed Graph. **SIAM J. Comput.**, 4(1):77–84, 1975. URL `http://dblp.uni-trier.de/db/journals/siamcomp/siamcomp4.html#Johnson75`. (Cited on page 288.)

Capers Jones and Olivier Bonsignour. **The economics of software quality**. Addison-Wesley Professional, 2012. (Cited on pages 1 and 156.)

Paul C Jorgensen. **Software testing: a craftsman's approach**. CRC press, 4 edition, 2013. (Cited on page 21.)

Philip J. Koopman Jr., John Sung, Christopher P. Dingman, Daniel P. Siewiorek, and Ted Marz. Comparing Operating Systems Using Robustness Benchmarks. In **Symposium on Reliable Distributed Systems**, pages 72–79, 1997. URL `http://dblp.uni-trier.de/db/conf/srds/srds97.html#KoopmanSDSM97`. (Cited on page 232.)

ISO/IEC JTC 1/SC 7. ISO/IEC 12207:2008 (Systems and software engineering - Software life cycle processes). Technical report, International Organization for Standardization, 2008. (Cited on pages 17 and 19.)

Natalia Juristo Juzgado, Ana MarÃa Moreno, and Sira Vegas. Reviewing 25 Years of Testing Technique Experiments. **Empirical Software Engineering**, 9(1-2):7–44, 2004. (Cited on pages 19, 22, 23, 24, and 287.)

Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In Ahmed Bouajjani and Oded Maler, editors, **CAV**, volume 5643 of **Lecture Notes in Computer Science**, pages 398–413. Springer, 2009. ISBN 978-3-642-02657-7. URL `http://dblp.uni-trier.de/db/conf/cav/cav2009.html#KahlonWG09`. (Cited on page 103.)

M. Kamel and S. Leue. Validation of a remote object invocation and object migration in CORBA GIOO using Promela/SPIN. **SPIN workshop, Paris, France**, 1998. (Cited on page 116.)

Moataz Kamel and Stefan Leue. Formalization and validation of the general inter-orb protocol (GIOP) using PROMELA and SPIN. In **Software Tools for Technology Transfer**, pages 394–409. Springer-Verlag, 2000. (Cited on page 116.)

Prof. Alan Kaminsky. **Big CPU, Big Data: Solving the World's Toughest Computational Problems with Parallel Computing**. Creative Commons, 2015. URL `http://www.cs.rit.edu/~ark/bcbd/`. (Cited on page 49.)

Hans W. Kamp. **Tense Logic and the Theory of Linear Order**. Phd thesis, Computer Science Department, University of California at Los Angeles, USA, 1968. (Cited on page 72.)

Susanne Kandl and Raimund Kirner. Error detection rate of MC/DC for a case study from the automotive domain. In **Software Technologies for Embedded and Ubiquitous Systems**, Lecture Notes in Computer Science, pages 131–142. Springer, 2010. (Cited on pages 21 and 23.)

Cem Kaner and Nawwar Kabbani. Software Metrics Threats to Validity. **Black Box Software Testing series**, 2012. URL `http://testingeducation.org/BBST/metrics/MetricsValidityLecture2012.pdf`. (Cited on page 372.)

Cem Kaner, Jack Falk, and Hung Quoc Nguyen. **Testing computer software (2. ed.).** Van Nostrand Reinhold, 1993. ISBN 978-0-442-01361-5. (Cited on pages 13 and 14.)

Gijs Kant and Jaco van de Pol. Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games. In Anton Wijs, Dragan Bosnacki, and Stefan Edelkamp, editors, **GRAPHITE**, volume 99 of **EPTCS**, pages 50–65, 2012. URL `http://dblp.uni-trier.de/db/series/eptcs/eptcs99.html#abs-1210-6414`. (Cited on page 110.)

Gregory M Kapfhammer. Automatically and transparently distributing the execution of regression test suites. In **Proceedings of the 18th International Conference on Testing Computer Software**, 2001. (Cited on pages 273 and 277.)

Kalpesh Kapoor and Jonathan P. Bowen. Experimental Evaluation of the Variation in Effectiveness for DC, FPC and MC/DC Test Criteria. In **ISESE**, pages 185–194. IEEE Computer Society, 2003. ISBN 0-7695-2002-2. (Cited on page 21.)

Kalpesh Kapoor and Jonathan P. Bowen. A formal analysis of MCDC and RCDC test criteria. **Softw. Test., Verif. Reliab.**, 15(1):21–40, 2005. reinforced condition decision coverage (RCDC), six kinds of faults. (Cited on page 21.)

Harmen Kastenberg and Arend Rensink. Dynamic Partial Order Reduction Using Probe Sets. In Franck van Breugel and Marsha Chechik, editors, **CONCUR**, volume 5201 of **Lecture Notes in Computer Science**, pages 233–247. Springer, 2008. ISBN 978-3-540-85360-2. URL `http://dblp.uni-trier.de/db/conf/concur/concur2008.html#KastenbergR08`. (Cited on page 103.)

Mika Katara and Antti Kervinen. Making Model-Based Testing More Agile: A Use Case Driven Approach. In **Haifa Verification Conference**, 2006. (Cited on page 348.)

Shmuel Katz and Doron A. Peled. An Efficient Verification Method for Parallel and Distributed Programs. In **REX workshop**, volume 398 of **LNCS**, pages 489–507. Springer, 1988. ISBN 978-3-540-51080-2. doi: 10.1007/BFb0013032. URL `http://dx.doi.org/10.1007/BFb0013032`. (Cited on page 104.)

Dirk O. Keck and Paul J. Kühn. The Feature and Service Interaction Problem in Telecommunications Systems. A Survey. **IEEE Trans. Software Eng.**, 24(10): 779–796, 1998. URL `http://dblp.uni-trier.de/db/journals/tse/tse24.html#KeckK98`. (Cited on page 176.)

Richard Kelsey. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In Michael D. Ernst, editor, **Intermediate Representations Workshop**, pages 13–23. ACM, 1995. ISBN 0-89791-754-5. URL `http://dblp.uni-trier.de/db/conf/irep/irep95.html#Kelsey95`. (Cited on page 157.)

Alain Kerbrat, Thierry JÃ©ron, and Roland Groz. Automated test generation from SDL specifications. In **SDL Forum**, pages 135–152, 1999. URL `http://dblp.uni-trier.de/db/conf/sdl/sdl1999.html#KerbratJG99`. (Cited on page 248.)

Sean Michael Kerner. Agile Development Method Growing in Popularity. **Forrester**, 2009. URL `http://www.internetnews.com/dev-news/print.php/3841571`. (Cited on page 341.)

Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, and Mika Katara. Model-Based Testing Through a GUI. In Wolfgang Grieskamp and Carsten Weise, editors, **FATES**, volume 3997 of **Lecture Notes in Computer Science**, pages 16–31. Springer, 2005. ISBN 3-540-34454-3. URL `http://dblp.uni-trier.de/db/conf/fates/fates2005.html#KervinenMPK05`. (Cited on page 204.)

Charles Killian, Jamnes W. Anderson, Ranjit Jhala, and Amnin Vahdat. Life, death, and the critical transition: Detecting liveness bugs in systems code. In **Proc. of the 4th Symposium on Networked Systems Design and Implementation (NSDI)**, Cambridge, MA, USA, 2007. (Cited on page 176.)

SC Kleene. Mathematical logic. 1967. (Cited on pages 3, 10, 25, 30, and 325.)

Richard Klein, National Center for Health Statistics (US), et al. **Healthy People 2010 criteria for data suppression**. Department of Health and Human Services, Centers for Disease Control and Prevention, National Center for Health Statistics, 2002. (Cited on page 372.)

A. V. Klepinin and A. A. Melentyev. Integration of semantic verifiers into Java language compilers. **Automatic Control and Computer Sciences**, 45(7):408–412, 2011. URL `http://dblp.uni-trier.de/db/journals/accs/accs45.html#KlepininM11`. (Cited on page 54.)

Ralf Kneuper. Limits of Formal Methods. **Formal Asp. Comput.**, 9(4):379–394, 1997. (Cited on page 2.)

Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In **Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming**, PPoPP '12, pages 141–150, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145835. URL `http://doi.acm.org/10.1145/2145816.2145835`. (Cited on page 142.)

D. König. **Theorie Der Endlichen Und Unendlichen Graphen**. Akad. Verlag, Leipzig, 1936. (Cited on page 204.)

N. Kothari, T. Millstein, and R. Govindan. Deriving State Machines from TinyOS Programs Using Symbolic Execution. In **IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks**, pages 271–282, April 2008. doi: 10.1109/IPSN.2008.62. (Cited on page 176.)

Jörg Kreiker, Andrzej Tarlecki, Moshe Y. Vardi, and Reinhard Wilhelm. Modeling, Analysis, and Verification - The Formal Methods Manifesto 2010 (Dagstuhl Perspectives Workshop 10482). **Dagstuhl Manifestos**, 1(1):21–40, 2011. URL `http://dblp.uni-trier.de/db/journals/dagstuhl-manifestos/dagstuhl-manifestos1.html#KreikerTVW11`. (Cited on pages vi and 4.)

Padmanabhan Krishnan, R. Venkatesh, Prasad Bokil, Tukaram Muske, and P. Vijay Suman. Effectiveness of Random Testing of Embedded Systems. In **HICSS**, pages 5556–5563. IEEE Computer Society, 2012. ISBN 978-0-7695-4525-7. (Cited on pages 22, 23, 24, and 287.)

Daniel Kroening and Ofer Strichman. **Decision procedures**, volume 5. Springer, 2008. (Cited on pages 27, 31, and 32.)

Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. Linear Completeness Thresholds for Bounded Model Checking. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, **CAV**, volume 6806 of **Lecture Notes in Computer Science**, pages 557–572. Springer, 2011. ISBN 978-3-642-22109-5. URL `http://dblp.uni-trier.de/db/conf/cav/cav2011.html#KroeningOSWW11`. (Cited on pages 106 and 107.)

Nathan P. Kropp, Philip J. Koopman Jr., and Daniel P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In **Proceedings of the**

Twenty-Eigth Annual International Symposium on Fault-Tolerant Computing (FTCS), Munich, Germany, pages 230–239. IEEE Computer Society, June 1998. ISBN 0-8186-8470-4. URL `http://dblp.uni-trier.de/db/conf/ftcs/ftcs98.html#KroppKS98`. (Cited on page 232.)

S. Kundu, S. Lerner, and R.K. Gupta. **High-Level Verification: Methods and Tools for Verification of System-Level Designs**. SpringerLink : Bücher. Springer New York, 2011. ISBN 9781441993595. URL `http://books.google.de/books?id=oY9AUETVXCgC`. (Cited on page 173.)

Orna Kupferman and Mosche Vardi. **Verification of Open Systems**. Springer, 2006. (Cited on page 16.)

Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. Module Checking. **Inf. Comput.**, 164(2):322–344, 2001. (Cited on page 16.)

Shigeru Kusakabe. Large Volume Testing for Executable Formal Specification Using Hadoop. In **Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW 2011)**, pages 1250–1257. IEEE, 2011. ISBN 978-1-61284-425-1. URL `http://dblp.uni-trier.de/db/conf/ipps/ipdps2011w.html#Kusakabe11`. (Cited on pages 274 and 275.)

Shigeru Kusakabe, Yoichi Omori, and Keijiro Araki. Facilitating Consistency Check between Specification and Implementation with MapReduce Framework. In **Proceedings of the 9th overture workshop**, page 32. Aarhus University, 2010. (Cited on pages 274 and 275.)

Felix Kutzner. A Case Study for Lazy On-The-Fly Model-Based Testing. Bachelor's thesis, Karlsruhe Institute of Technology, 2014. (Cited on pages 8, 325, 326, 339, 354, 355, 357, 360, 369, and 385.)

Felix Kutzner and David Faragó. LazyOTF Extensions to JTorX. web page, 2015. URL `https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki`. (Cited on pages 325 and 331.)

Alfons Laarman. **Scalable Multi-Core Model Checking**. PhD thesis, University of Twente, Formal Methods & Tools, 2014. (Cited on pages 57, 102, 109, 111, 138, 142, and 143.)

Alfons Laarman and David Faragó. Improved On-The-Fly Livelock Detection: Combining Partial Order Reduction and Parallelism for DFS-FIFO. **Proceedings of the 5th NASA Formal Methods Symposium (NFM 2013), NASA Ames Research Center, CA, USA, May 14-16, 2013. LNCS, Springer**, 2013. (Cited on pages 7, 57, 115, 145, and 153.)

Alfons Laarman, Jaco van de Pol, and Michael Weber 0002. Boosting multi-core reachability performance with shared hash tables. In Roderick Bloem and Natasha Sharygina, editors, **FMCAD**, pages 247–255. IEEE, 2010. URL `http://dblp.uni-trier.de/db/conf/fmcad/fmcad2010.html#LaarmanPW10`. (Cited on pages 57, 111, 142, 143, and 147.)

Alfons Laarman, Jaco van de Pol, and Michael Weber 0002. Multi-Core LTSmin: Marrying Modularity and Scalability. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, **NASA Formal Methods**, volume 6617 of **Lecture Notes in Computer Science**, pages 506–511. Springer, 2011a. ISBN 978-3-642-20397-8. URL `http://dblp.uni-trier.de/db/conf/nfm/nfm2011.html#LaarmanPW11`. (Cited on pages 111 and 142.)

Alfons Laarman, Jaco van de Pol, and Michael Weber 0002. Parallel Recursive State Compression for Free. In Alex Groce and Madanlal Musuvathi, editors, **SPIN**, volume 6823 of **Lecture Notes in Computer Science**, pages 38–56. Springer, 2011b. ISBN 978-3-642-22305-1. URL `http://dblp.uni-trier.de/db/conf/spin/spin2011.html#LaarmanP011`. (Cited on page 107.)

Alfons Laarman, Elwin Pater, Jaco van de Pol, and Michael Weber. Guard-based Partial Order Reduction. In **SPIN**, volume submitted for publication of **LNCS**. Springer, 2013a. (Cited on pages 103, 110, and 111.)

A.W. Laarman and J.C. van de Pol. Variations on Multi-Core Nested Depth-First Search. In **PDMC**, pages 13–28, 2011. (Cited on pages 138 and 150.)

A.W. Laarman, R. Langerak, J.C. van de Pol, M. Weber, and A. Wijs. Multi-Core NDFS. In **ATVA**, volume 6996 of **LNCS**, pages 321–335. Springer, 2011c. URL `http://eprints.eemcs.utwente.nl/20337/`. (Cited on pages 138 and 144.)

A.W. Laarman, M. Chr. Olesen, A.E. Dalsgaard, K.G. Larsen, and J.C. van de Pol. Multi-Core Emptiness Checking of Timed Büchi Automata using Inclusion Abstraction. **Proceedings of the 25th International Conference on Computer Aided Verification, CAV 2013, 13-19 July 2013, Saint Petersburg, Russia. LNCS Springer.**, 2013b. (Cited on page 110.)

Hartmut Lackner and Holger Schlingloff. Modeling for automated test generation - a comparison. In Holger Giese, Michaela Huhn, Jan Phillips, and Bernhard Schätz, editors, **Modellbasierte Entwicklung eingebetteter Systeme VIII (MBEES), 2012**, pages 57–70. fortiss GmbH, München, 2012. (Cited on pages 14, 22, 247, 253, 259, 286, 295, and 373.)

Beatriz Pérez Lamancha, Pedro Reales Mateo, Ignacio Rodríguez de Guzmán, Macario Polo Usaola, and Mario Piattini Velthius. Automated model-based testing using the UML testing profile and QVT. In **Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation**, page 6. ACM, 2009. (Cited on page 238.)

Leslie Lamport. What Good is Temporal Logic? In **IFIP Congress**, pages 657–668, 1983. URL `http://dblp.uni-trier.de/db/conf/ifip/ifip83.html#Lamport83`. (Cited on page 71.)

Boris Langer and Michael Tautschnig. Navigating the Requirements Jungle. In Tiziana Margaria and Bernhard Steffen, editors, **Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation**,

volume 17 of **Communications in Computer and Information Science**, pages 354–368. Springer, 2008. ISBN 978-3-540-88478-1. URL `http://dblp.uni-trier.de/db/conf/isola/isola2008.html#LangerT08`. (Cited on pages 14, 17, and 18.)

Kim G Larsen, Alexandre David, Holger Hermanns, Joost-Peter Katoen, Brian Nielsen, and Axel Belinfante. Quasimodo deliverable D5.9: Tool Components and Tool Integration. Technical report, AAU, June 2011. URL `http://www.quasimodo.aau.dk/Final/Quasimodo-D5.9.pdf`. (Cited on page 233.)

Florian Larysch. Improved constraint specification and solving for Lazy On-The-Fly. Bachelor's thesis, Karlsruhe Institute of Technology, 2012. (Cited on pages 8, 233, 325, 334, and 339.)

Alexey L. Lastovetsky. Parallel testing of distributed software. **Information & Software Technology**, 47(10):657–662, 2005. URL `http://dblp.uni-trier.de/db/journals/infsof/infsof47.html#Lastovetsky05`. (Cited on page 273.)

Chris Lattner and Vikram Adve. **LLVM Language Reference Manual**, February 2011. (Cited on page 113.)

Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi A. Junttila. Simple Bounded LTL Model Checking. In Alan J. Hu and Andrew K. Martin, editors, **FMCAD**, volume 3312 of **Lecture Notes in Computer Science**, pages 186–200. Springer, 2004. ISBN 3-540-23738-0. URL `http://dblp.uni-trier.de/db/conf/fmcad/fmcad2004.html#LatvalaBHJ04`. (Cited on page 106.)

Joseph Lawrance, Steven Clarke, Margaret Burnett, and Gregg Rothermel. How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study. In **VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing**, pages 53–60, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2443-5. doi: http://dx.doi.org/10.1109/VLHCC.2005.44. (Cited on page 344.)

Adrian E Lawrence. Acceptances, Behaviours and infinite activity in CSPP. **Communicating process architectures**, pages 17–38, 2002. (Cited on page 60.)

Gary T Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M Zimmerman, et al. JML reference manual, 2008. (Cited on pages 3, 39, and 44.)

D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. **Proceedings of the IEEE**, 84(8):1090–1123, August 1996. ISSN 00189219. doi: 10.1109/5.533956. URL `http://dx.doi.org/10.1109/5.533956`. (Cited on pages 186, 205, 218, and 221.)

Edward A. Lee. The Problem with Threads. **IEEE Computer**, 39(5):33–42, 2006. URL `http://dblp.uni-trier.de/db/journals/computer/computer39.html#Lee06`. (Cited on page 50.)

B. Lewis and D.J. Berg. **Multithreaded Programming With Java Technology**. The Sun Microsystems Press Java Series. Pearson Educacition, 2000. ISBN 9780130170071. URL `http://books.google.de/books?id=rrBQAAAAMAAJ`. (Cited on pages 48 and 142.)

Peng Li and John Regehr. T-Check: Bug Finding for Sensor Networks. In **Proc. 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2010)**, pages 174–185, School of Computing, University of Utah, USA, April 2010. ACM. (Cited on pages 167 and 175.)

Man Liu. SBMC-Based Testcase Generation. Master's thesis, Karlsruhe Institute of Technology, 2015. (Cited on pages 8, 175, and 179.)

Brad Long and Paul A. Strooper. A Case Study in Testing Distributed Systems. In **Proceedings 3rd International Symposium on Distributed Objects and Applications (DOA)**, pages 20–30, 2001. URL `http://dblp.uni-trier.de/db/conf/doa/doa2001.html#LongS01`. (Cited on page 279.)

Nancy A. Lynch. **Distributed Algorithms**. Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1996. ISBN 1-55860-348-4. URL `http://theory.lcs.mit.edu/tds/distalgs.html`. http://theory.lcs.mit.edu/tds/distalgs.html. (Cited on pages 43, 47, 111, and 275.)

Peter Dinda Maciej Swiech, Kyle C. Hale. VMM-based Emulation of Intel Hardware Transactional Memory. **International Journal of High Performance Computing Applications**, 26(2):125–135, May 2012. URL `http://v3vmm.org/papers/NWU-EECS-13-03.pdf`. (Cited on page 51.)

Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. **IEEE Trans. Software Eng.**, 40(1):23–42, 2014. (Cited on pages 24 and 179.)

Jeff Magee and Jeffrey Kramer. **Concurrency: State Models and Java Programs**. Wiley & Sons, New York, NY, USA, 1999. URL `http://www-dse.doc.ic.ac.uk/concurrency/`. (Cited on pages 247 and 248.)

Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An Efficient SAT Solver. In Holger H. Hoos and David G. Mitchell, editors, **SAT**, volume 3542 of **Lecture Notes in Computer Science**, pages 360–375. Springer, 2004. ISBN 3-540-27829-X. (Cited on page 55.)

J. Kazmi S. Mahlknecht. Wireless sensor networks: modeling and simulation. **Discrete Event Simulations, SCIYO,**, pages 247 – 262, 2010. (Cited on page 167.)

K.L. Man, Hong Kong Solari, T. Vallee, H.L. Leung, M. Mercaldi, J. van der Wulp, M. Donno, and M. Pastrnak. TEPAWSN - A Tool Environment for Wireless Sensor Networks (WSNs). **Proceedings of the 4th IEEE Conference on Industrial Electronics and Applications - ICIEA 2009, Xi'an, China, May, 2009.**, May 2009. (Cited on page 167.)

Zohar Manna and Amir Pnueli. The anchored version of the temporal framework. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, **REX Workshop**, volume 354 of **Lecture Notes in Computer Science**, pages 201–284. Springer, 1988. ISBN 3-540-51080-X. URL `http://dblp.uni-trier.de/db/conf/rex/rex88.html#MannaP88`. (Cited on page 96.)

Zohar Manna and Amir Pnueli. **The Temporal Logic of Reactive and Concurrent Systems**. Springer-Verlag, New York, USA, 1992. (Cited on page 115.)

Zohar Manna and Amir Pnueli. **Temporal verification of reactive systems - safety.** Springer, 1995. ISBN 978-0-387-94459-3. (Cited on pages 72, 93, and 157.)

Tiziana Margaria and Bernhard Steffen, editors. **Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies. Proceedings (Part II) of the 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece**, volume 7610 of **Lecture Notes in Computer Science**, October 2012. Springer. ISBN 978-3-642-34031-4. URL `http://dblp.uni-trier.de/db/conf/isola/isola2012-2.html`. (Cited on page 3.)

Filip Maric. Formalization and Implementation of Modern SAT Solvers. **J. Autom. Reasoning**, 43(1):81–119, 2009. URL `http://dblp.uni-trier.de/db/journals/jar/jar43.html#Maric09`. (Cited on page 27.)

Simon Marlow et al. Haskell 2010 language report. **http://www.haskell.org/onlinereport/haskell2010**, 2010. (Cited on page 274.)

Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. **ACM Trans. Model. Comput. Simul.**, 8(1):3–30, 1998. URL `http://dblp.uni-trier.de/db/journals/tomacs/tomacs8.html#MatsumotoN98`. (Cited on pages 351 and 371.)

Industrieanlagen-Betriebsgesellschaft mbH. Part 1: Fundamentals of the V-Modell. page 825, 2006. URL `http://v-modell.iabg.de/dmdocuments/V-Modell-XT-Gesamt-Englisch-V1.3.pdf`. (Cited on page 19.)

Mike McBride. Multicast in the data center overview. Technical report, IETF, February 2013. URL `http://tools.ietf.org/html/draft-ietf-mboned-dc-deploy-00`. (Cited on page 53.)

John McCarthy. Programs with Common Sense. In **Proceedings of the Teddington Conference on the Mechanization of Thought Processes**, pages 75–91, London, 1959. Her Majesty's Stationary Office. URL `http://www-formal.stanford.edu/jmc/mcc59.html`. (Cited on page 185.)

Steve McConnell. **Code Complete: A Practical Handbook of Software Construction**. Microsoft Press, Redmond, WA, 2. edition, 2004. ISBN 978-0-7356-1967-8. (Cited on page 1.)

Michael D. McCool, Arch D. Robison, and James Reinders. **Structured parallel programming: Patterns for efficient computation**. Elsevier/Morgan Kaufmann,

Waltham, MA, 2012. ISBN 9780124159938 0124159931. (Cited on pages 48, 49, 54, and 57.)

Pat McGee and Cem Kaner. Extended Random Regression Testing: Finding valuable bugs by running long pseudo-random sequences of already-passed complex tests. Technical report, CSTER, 2003. URL `http://www.kaner.com/pdfs/automation/Mentsville`. (Cited on page 274.)

Pat McGee and Cem Kaner. Experiments with high volume test automation. **ACM SIGSOFT Software Engineering Notes**, 29(5):1–3, 2004. URL `http://dblp.uni-trier.de/db/journals/sigsoft/sigsoft29.html#McGeeK04`. (Cited on page 274.)

Florian Merz, Carsten Sinz, Hendrik Post, Thomas Gorges, and Thomas Kropf. Abstract Testing: Connecting Source Code Verification with Requirements. In Fernando Brito e Abreu, Joao Pascoal Faria, and Ricardo Jorge Machado, editors, **QUATIC**, pages 89–96. IEEE Computer Society, 2010. ISBN 978-0-7695-4241-6. URL `http://dblp.uni-trier.de/db/conf/quatic/quatic2010.html#MerzSPGK10`. (Cited on pages 173 and 377.)

Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0. Specification, September 2012. URL `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`. (Cited on page 48.)

Bertrand Meyer. Applying design by contract. **IEEE Computer**, 25:40–51, 1992. (Cited on page 39.)

Bertrand Meyer. **Object-Oriented Software Construction**. Prentice Hall, Upper Saddle River, NJ, 2. edition, 1997. ISBN 978-0-13-629155-8. (Cited on page 4.)

Maged M Michael. ABA prevention using single-word instructions. **IBM Research Division, RC23089 (W0401-136), Tech. Rep**, 2004. (Cited on pages 50 and 51.)

L.I. Millet. Slicing Promela and its Applications to Protocol Understanding and Analysis. In **Proceedings of the Fourth SPIN Workshop. 1998.** American Mathematical Society., 1998. URL `http://netlib.bell-labs.com/netlib/spin/ws98/millet.ps.gz`. (Cited on page 107.)

South San Francisco CA Millind Mittal and Tivon (IL) Eval Waldman. Compare and exchange operation in a processing system. Patent, 03 1999. URL `http://www.patentlens.net/patentlens/patent/US_5889983/en/`. US 5889983. (Cited on page 50.)

Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, parts I and II. **Inf. Comput.**, 100(1):1–77, September 1992. (Cited on pages 41, 43, and 54.)

Michael Mlynarski. **Holistic Use of Analysis Models in Model-Based System Testing**. PhD thesis, University of Paderborn, 2011. (Cited on pages 17, 19, and 346.)

Michael Mlynarski, Baris Güldali, Stephan Weißleder, and Gregor Engels. Model-Based Testing: Achievements and Future Challenges. **Advances in Computers**, 86:1–39, 2012. URL `http://dblp.uni-trier.de/db/journals/ac/ac86.html#MlynarskiGWE12`. (Cited on pages 2, 239, and 341.)

Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In **ESEM**, pages 291–301, 2009. ISBN 978-1-4244-4842-5. URL `http://dblp.uni-trier.de/db/conf/esem/esem2009.html#MockusND09`. (Cited on pages 23 and 287.)

J.C. Mogul. Broadcasting Internet Datagrams. RFC 919 (Standard), October 1984a. URL `http://www.ietf.org/rfc/rfc919.txt`. (Cited on page 52.)

J.C. Mogul. Broadcasting Internet datagrams in the presence of subnets. RFC 922 (Standard), October 1984b. URL `http://www.ietf.org/rfc/rfc922`. (Cited on page 52.)

Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. **IEEE Trans. Parallel Distrib. Syst.**, 2(3):264–280, 1991. (Cited on pages 50 and 57.)

G.E. Moore. Cramming more Components onto Integrated Circuits. **Electronics**, 38 (10):114–117, 1965. (Cited on page 47.)

Damon Mosk-Aoyama and Mihalis Yannakakis. Testing hierarchical systems. In **SODA**, pages 1126–1135. SIAM, 2005. ISBN 0-89871-585-7. URL `http://dblp.uni-trier.de/db/conf/soda/soda2005.html#Mosk-AoyamaY05`. (Cited on page 284.)

Luca Mottola, Thiemo Voigt, Fredrik Österlind, Joakim Eriksson, Luciano Baresi, and Carlo Ghezzi. Anquiro: Enabling Efficient Static Verification of Sensor Network Software. In **Proc. 1st International Workshop on Software Engineering for Sensor Networks (SESENA - colocated with 32nd ACM/IEEE International Conference on Software Engineering ICSE)**. ACM, May 2010. (Cited on page 176.)

Rajeev Motwani and Prabhakar Raghavan. **Randomized Algorithms**. Cambridge University Press, Cambridge, England, first edition, June 1995. ISBN 0-521-47465-5. (Cited on pages 241 and 355.)

Matthias Müller-Hannemann and Stefan Schirra, editors. **Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice [outcome of a Dagstuhl Seminar]**, volume 5971 of **Lecture Notes in Computer Science**, 2010. Springer. ISBN 978-3-642-14865-1. (Cited on page 49.)

Glenford J. Myers. **The art of software testing (2. ed.).** Wiley, 2004. ISBN 978-0-471-46912-4. (Cited on page 14.)

Sagar Naik and Piyu Tripathy. **Software testing and quality assurance: theory and practice**. John Wiley & Sons, 2011. (Cited on pages 2, 13, 22, and 252.)

Tadashi Nakatani. Verification of Group Address Registration Protocol using PROMELA and SPIN. In **In International SPIN Workshop**, page 97, 1997. (Cited on page 116.)

V Natarajan and Gerard J Holzmann. Outline for an operational semantics of promela. **The SPIN Verification Systems. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. AMS**, 32:133–152, 1997. (Cited on pages 46 and 47.)

Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. Requirements by Contracts allow Automated System Testing. In **Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 03)**, page 85, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2007-3. (Cited on page 39.)

Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. **ACM Trans. Program. Lang. Syst.**, 1(2):245–257, 1979. URL `http://dblp.uni-trier.de/db/journals/toplas/toplas1.html#NelsonO79`. (Cited on page 31.)

J. Neyman and E. S. Pearson. On the Problem of the Most Efficient Tests of Statistical Hypotheses. **Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character**, 231: 289–337, 1933. ISSN 02643952. URL `http://www.jstor.org/stable/91247`. (Cited on page 25.)

David M. Nicol and Frank H. Willard. Problem Size, Parallel Architecture, and Optimal Speedup. **J. Parallel Distrib. Comput.**, 5(4):404–420, 1988. URL `http://dblp.uni-trier.de/db/journals/jpdc/jpdc5.html#NicolW88`. (Cited on page 49.)

Antti Nieminen, Antti Jääskeläinen, Heikki Virtanen, and Mika Katara. A Comparison of Test Generation Algorithms for Testing Application Interactions. In Manuel Nunez, Robert M. Hierons, and Mercedes G. Merayo, editors, **QSIC**, pages 131–140. IEEE Computer Society, 2011. (Cited on pages 208, 256, 259, 286, 348, and 371.)

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). **ACM**, 53(6):937–977, 2006. URL `http://dblp.uni-trier.de/db/journals/jacm/jacm53.html#NieuwenhuisOT06`. (Cited on page 31.)

Dor Nir, Shmuel S. Tyszberowicz, and Amiram Yehudai. Locating Regression Bugs. In Karen Yorav, editor, **Haifa Verification Conference**, volume 4899 of **Lecture Notes in Computer Science**, pages 218–234. Springer, 2007. ISBN 978-3-540-77964-3. (Cited on page 19.)

Joost Noppen, Pim van den Broek, and Mehmet Aksit. Software development with imperfect information. **Soft Comput.**, 12(1):3–28, 2008. (Cited on page 333.)

Kimmo Nupponen. Distributed Test Generation. 2014. URL `https://www.conformiq.com/2014/02/distributed-test-generation/`. (Cited on pages 252, 259, 272, 274, 278, and 364.)

Jan Henry Nyström. **Analysing Fault Tolerance for Erlang Applications**. PhD thesis, Uppsala University, 2009. (Cited on page 204.)

University of Chicago Press. **The Chicago Manual of Style**. University of Chicago Press, 16th edition, 2010. (Cited on page 10.)

A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In **Mutation testing for the new century**, pages 34–44. Springer, 2001. (Cited on pages 19, 23, and 179.)

Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = Lazy Clause Generation. In Christian Bessiere, editor, **Principles and Practice of Constraint Programming (CP 2007)**, volume 4741 of **Lecture Notes in Computer Science**, pages 544–558. Springer, 2007. ISBN 978-3-540-74969-1. (Cited on page 55.)

OpenDO. Project AGILE, 2009. URL `http://www.open-do.org/projects/agile`. (Cited on page 348.)

Manuel Oriol and Faheem Ullah. YETI on the Cloud. In **Software Testing, Verification, and Validation Workshops (ICSTW), 2010**, pages 434–437. IEEE Computer Society, 2010. URL `http://dblp.uni-trier.de/db/conf/icst/icstw2010.html#OriolU10`. (Cited on pages 241, 272, and 275.)

Bryan O'Sullivan, Don Stewart, and John Goerzen. **Real world Haskell**. O'Reilly, 2009. ISBN 9780596514983. URL `http://www.worldcat.org/isbn/9780596514983`. (Cited on page 274.)

Prasanna Padmanabhan and Robyn R. Lutz. Tool-Supported Verification of Product Line Requirements. **Autom. Softw. Eng.**, 12(4):447–465, 2005. URL `http://dblp.uni-trier.de/db/journals/ase/ase12.html#PadmanabhanL05`. (Cited on page 176.)

Catuscia Palamidessi. Comparing The Expressive Power Of The Synchronous And Asynchronous Pi-Calculi. **Mathematical Structures in Computer Science**, 13(5): 685–719, 2003. (Cited on pages 41 and 42.)

Stephen R. Palmer and John M. Felsing. **A Practical Guide to Feature-Driven Development (The Coad Series)**. Prentice Hall PTR, February 2002. ISBN 0130676152. URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0130676152`. (Cited on page 342.)

Nikos Papadakis, Polydoros Petrakis, Dimitris Plexousakis, and Haridimos Kondylakis. A Solution to the Ramification Problem Expressed in Temporal Description Logics, 2008. (Cited on page 185.)

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In Barbara G. Ryder and Andreas Zeller, editors, **Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2008**, pages 201–212. ACM, 2008. ISBN 978-1-60558-050-0. URL `http://dblp.uni-trier.de/db/conf/issta/issta2008.html#PapiACPE08`. (Cited on pages 4 and 328.)

Pavel Parizek, JirÃ AdÃ¡mek, and Tomas Kalibera. Automated Construction of Reasonable Environment for Java Components. **Electr. Notes Theor. Comput. Sci.**, 253(1):145–160, 2009. URL `http://dblp.uni-trier.de/db/journals/entcs/entcs253.html#ParizekAK09`. (Cited on page 61.)

David Lorge Parnas. On the criteria to be used in decomposing systems into modules. **Communications of the ACM**, 15(12):1053–1058, 1972. (Cited on page 54.)

Tauhida Parveen and Scott R. Tilley. When to Migrate Software Testing to the Cloud? **Third International Conference on Software Testing, Verification, and Validation Workshop**, pages 424–427, 2010. URL `http://dblp.uni-trier.de/db/conf/icst/icstw2010.html#ParveenT10`. (Cited on pages 274 and 278.)

Tauhida Parveen, Scott Tilley, Nigel Daley, and Pedro Morales. Towards a distributed execution framework for JUnit test cases. In **IEEE International Conference on Software Maintenance (ICSM 2009)**, pages 425–428. IEEE, 2009. URL `http://dblp.uni-trier.de/db/conf/icsm/icsm2009.html#ParveenTDM09`. (Cited on page 273.)

Alexander Pascanu. Eine Fallstudie zu modellbasiertem Testen von Webservices. Master's thesis, Universität Karlsruhe, 2010. (Cited on pages 8 and 252.)

E. Pater. Partial Order Reduction for PINS, Master's thesis. Uni. of Twente, 2011. URL `http://essay.utwente.nl/61036/`. (Cited on pages 103 and 111.)

Radek Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In Dragan Bosnacki and Stefan Edelkamp, editors, **SPIN**, volume 4595 of **Lecture Notes in Computer Science**, pages 263–267. Springer, 2007. ISBN 978-3-540-73369-0. URL `http://dblp.uni-trier.de/db/conf/spin/spin2007.html#Pelanek07`. (Cited on page 116.)

Doron A. Peled. All from One, One for All: on Model Checking Using Representatives. In Costas Courcoubetis, editor, **CAV**, volume 697 of **Lecture Notes in Computer Science**, pages 409–423. Springer, 1993. ISBN 3-540-56922-7. URL `http://dblp.uni-trier.de/db/conf/cav/cav93.html#Peled93`. (Cited on page 103.)

Doron A. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In **6th International Conference on Computer Aided Verification, Stanford, California**, 1994. URL `http://www.dcs.warwick.ac.uk/~doron/ps/full.ps`. (Cited on page 104.)

Jan Peleska. Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. In Alexander K. Petrenko and Holger Schlingloff, editors, **Eighth Workshop on Model-Based Testing (MBT 2013)**, volume 111 of **EPTCS**, pages 3–28. EPTCS, 2013. URL `http://dblp.uni-trier.de/db/series/eptcs/eptcs111.html#abs-1303-1006`. (Cited on pages 3, 13, 22, 91, 237, 256, 291, and 345.)

Justyna Petke. **Bridging Constraint Satisfaction and Boolean Satisfiability**. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2015. ISBN 978-3-319-21810-6. (Cited on page 27.)

Mauro Pezzé and Michal Young. **Software testing and analysis - process, principles and techniques.** Wiley, 2007. ISBN 978-0-471-45593-6. (Cited on pages 18, 22, and 343.)

Iain Phillips. Refusal Testing. **Theor. Comput. Sci.**, 50:241–284, 1987. URL `http://dblp.uni-trier.de/db/journals/tcs/tcs50.html#Phillips87`. (Cited on page 184.)

Benjamin C. Pierce. Foundational Calculi for Programming Languages. In Allen B. Tucker, editor, **The Computer Science and Engineering Handbook**, pages 2190–2207. CRC Press, 1997. ISBN 0-8493-2909-4. URL `http://dblp.uni-trier.de/db/books/collections/tucker97.html#Pierce97`. (Cited on pages 41 and 43.)

Benjamin C. Pierce. **Types and programming languages.** MIT Press, 2002. ISBN 978-0-262-16209-8. (Cited on page 3.)

Francisco A. C. Pinheiro and Joseph A. Goguen. An Object-Oriented Tool for Tracing Requirements. **IEEE Software**, 13(2):52–64, 1996. (Cited on page 17.)

Nico Plat and Peter Gorm Larsen. An overview of the ISO/VDM-SL standard. **SIGPLAN Notices**, 27(8):76–82, 1992. URL `http://dblp.uni-trier.de/db/journals/sigplan/sigplan27.html#PlatL92`. (Cited on page 274.)

EN 50128 Plenary. BS EN 50128 - Railway applications. Communication, signalling and processing systems. Software for railway control and protection systems. 2011. (Cited on page 4.)

F. Ployette, B. Jeannet, and T. Jéron. Stg: a symbolic test generation tool for reactive systems. TESTCOM/FATES07 (Tool Paper), June 2007. (Cited on page 251.)

A. Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, **Logics and models of concurrent systems**, chapter In transition from global to modular temporal reasoning about programs, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA, 1985. ISBN 0-387-15181-8. URL `http://dl.acm.org/citation.cfm?id=101969.101977`. (Cited on page 96.)

Amir Pnueli. The Temporal Logic of Programs. In **FOCS**, pages 46–57. IEEE Computer Society, 1977. URL `http://dblp.uni-trier.de/db/conf/focs/focs77.html#Pnueli77`. (Cited on page 69.)

H. Post, C. Sinz, A. Kaiser, and T. Gorges. Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking. In **ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering**, pages 188–197, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2187-9. doi: http://dx.doi.org/10.1109/ASE.2008.29. (Cited on page 178.)

Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. **STTT**, 7(2):156–173, 2005. (Cited on page 155.)

Wolfgang Prenninger and Alexander Pretschner. Abstractions for Model-Based Testing. **Electr. Notes Theor. Comput. Sci.**, 116:59–71, 2005. URL `http://dblp.uni-trier.de/db/journals/entcs/entcs116.html#PrenningerP05`. (Cited on pages 59, 60, 207, and 343.)

Alexander Pretschner and Jan Philipps. Methodological Issues in Model-Based Testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, **Model-Based Testing of Reactive Systems**, volume 3472 of **Lecture Notes in Computer Science**. Springer Berlin Heidelberg, 2005. (Cited on pages 17 and 263.)

Alexander Pretschner, Dominik Holling, Robert Eschbach, and Matthias Gemmar. A Generic Fault Model for Quality Assurance. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, **MoDELS**, volume 8107 of **Lecture Notes in Computer Science**, pages 87–103. Springer, 2013. ISBN 978-3-642-41532-6. URL `http://dblp.uni-trier.de/db/conf/models/models2013.html#PretschnerHEG13`. (Cited on pages 23 and 287.)

Priyanka, Inderveer Chana, and Ajay Rana. Empirical evaluation of cloud-based testing techniques: a systematic review. **ACM SIGSOFT Software Engineering Notes**, 37(3):1–9, 2012. URL `http://dblp.uni-trier.de/db/journals/sigsoft/sigsoft37.html#PriyankaCR12`. (Cited on pages 272 and 278.)

Olli-Pekka Puolitaival. Adapting model-based testing to agile context. **ESPOO2008**, 2008. (Cited on page 348.)

Xiao Qu and Brian Robinson. A Case Study of Concolic Testing Tools and their Limitations. In **ESEM**, pages 117–126. IEEE, 2011. ISBN 978-1-4577-2203-5. URL `http://dblp.uni-trier.de/db/conf/esem/esem2011.html#QuR11`. (Cited on page 174.)

Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In Didier El Baz, Francois Spies, and Tom Gross, editors, **17th Euromicro International Conference on Parallel, Distributed and Network-based Processing**, pages 427–436. IEEE, 2009. (Cited on page 48.)

Michael O. Rabin and D. Scott. Finite Automata and Their Decision Problems. **IBM Journal of Research and Development**, 3(2):114–125, 1959. (Cited on page 197.)

Joseph B. Rainsberger. Integration Tests Are a Scam. **Agile2009**, 2009. doi: http://www.infoq.com/presentations/integration-tests-scam. (Cited on pages 343 and 348.)

Ajitha Rajan. **Coverage Metrics for Requirements-Based Testing**. PhD thesis, University of Minnesota, 2009. (Cited on pages 17, 22, and 271.)

Ajitha Rajan, Michael W. Whalen, and Mats P.E. Heimdahl. The effect of program and model structure on mc/dc test adequacy coverage. In Wilhelm Schäfer, Matthew B.

Dwyer, and Volker Gruhn, editors, **Proceedings of the 30th International Conference on Software Engineering (ICSE 08)**, pages 161–170. ACM, 2008a. ISBN 978-1-60558-079-1. (Cited on pages 21 and 23.)

Ajitha Rajan, Michael W. Whalen, Matt Staats, and Mats P.E. Heimdahl. Requirements Coverage as an Adequacy Measure for Conformance Testing. In Shaoying Liu, T. S. E. Maibaum, and Keijiro Araki, editors, **Proceedings of the 10th International Conference on Formal Methods and Software Engineering (ICFEM), Kitakyushu-City, Japan**, volume 5256 of **Lecture Notes in Computer Science**, pages 86–104. Springer, October 2008b. ISBN 978-3-540-88193-3. URL `http://dblp.uni-trier.de/db/conf/icfem/icfem2008.html#RajanWSH08`. (Cited on pages 22, 253, 261, and 287.)

Ravi Rajwar, Bret L. Toll, Konrad K. Lai, Matthew C. Merten, and Martin G. Dixon. Instruction and logic to test transactional execution status. Patent, 08 2013. URL `http://www.google.com/patents/US20130205119`. US 20130205119 A1. (Cited on page 51.)

V. Nageshwara Rao and Vipin Kumar. Superlinear Speedup in Parallel State-Space Search. In Kesav V. Nori and Sanjeev Kumar, editors, **FSTTCS**, volume 338 of **Lecture Notes in Computer Science**, pages 161–174. Springer, 1988. ISBN 3-540-50517-2. URL `http://dblp.uni-trier.de/db/conf/fsttcs/fsttcs88.html#RaoK88`. (Cited on page 151.)

Nicole Rauch, Eberhard Kuhn, and Holger Friedrich. Index-based Process and Software Quality Control in Agile Development Projects. **CompArch2008**, 2008. doi: http://comparch2008.ipd.kit.edu/fileadmin/user_upload/comparch2008/iert-rauch-andrena.pdf. (Cited on page 345.)

Michel Raynal. **Distributed Algorithms for Message-Passing Systems**. Springer, 2013. ISBN 9783642381232. (Cited on pages 41, 48, and 275.)

P Vijaya Vardhan Reddy and Lakshmi Rajamani. Evaluation of different hypervisors performance in the private cloud with SIGAR framework. **International Journal of Advanced Computer Science and Applications**, 5(2), 2014. (Cited on page 371.)

Ralf H. Reussner and Viktoria Firus. Basic and Dependent Metrics. In Irene Eusgeld, Felix C. Freiling, and Ralf H. Reussner, editors, **Dependability Metrics**, volume 4909 of **Lecture Notes in Computer Science**, pages 37–38. Springer, 2005. ISBN 978-3-540-68946-1. (Cited on page 49.)

Hiralal Agrawal Richard, Richard A Demillo, Bob Hathaway, William Hsu, Wynne Hsu, EW Krauser, RJ Martin, Aditya P Mathur, and Eugene H Spafford. Design Of Mutant Operators For The C Programming Language. Technical Report SERC-TR-41P, Purdue University, West Lafayette, 1989. (Cited on page 23.)

Heinz Riener, Roderick Bloem, and Görschwin Fey. Test Case Generation from Mutants Using Model Checking Techniques. In **ICST Workshops**, pages 388–397. IEEE Computer Society, 2011. ISBN 978-0-7695-4345-1. (Cited on page 179.)

George F. Riley and Thomas R. Henderson. The ns-3 Network Simulator. In Klaus Wehrle, Mesut Günes, and James Gross, editors, **Modeling and Tools for Network Simulation**, pages 15–34. Springer, 2010. ISBN 978-3-642-12330-6. URL `http://dblp.uni-trier.de/db/books/collections/Wehrle2010.html#RileyH10`. (Cited on page 167.)

Petr Rockai, Jiri Barnat, and Lubos Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In **NASA Formal Methods (NFM 2013)**, volume 7871 of **LNCS**, pages 1–15. Springer, 2013. (Cited on page 112.)

Emanuele Di Rosa, Enrico Giunchiglia, Massimo Narizzano, Gabriele Palma, and Alessandra Puddu. Automatic generation of high quality test sets via CBMC. In Markus Aderhold, Serge Autexier, and Heiko Mantel, editors, **VERIFY at IJCAR**, volume 3 of **EPiC Series**, pages 65–78, 2010. (Cited on page 175.)

B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In **15th ACM Symposium on principles of Programming Languages**, pages 12–27, 1988. (Cited on page 157.)

Thomas Roßner, Christian Brandes, Helmut Götz, and Mario Winter. **Basiswissen modellbasierter Test**. Dpunkt.Verlag GmbH, 2010. ISBN 9783898645898. URL `http://books.google.de/books?id=WfGlQQAACAAJ`. (Cited on page 238.)

Christian Esteve Rothenberg. Re-architected Cloud Data Center Networks and Their Impact on the Future Internet. In **New Network Architectures**, pages 179–187. Springer, 2010. (Cited on page 53.)

W. Royce. Managing the Development of Large Software Systems. In **Proceesings. IEEE Wescon**, August 1970. (Cited on pages 17 and 19.)

Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In Michael R. Lightner and Jochen A. G. Jess, editors, **ICCAD**, pages 42–47. IEEE Computer Society, 1993. ISBN 0-8186-4490-7. URL `http://dblp.uni-trier.de/db/conf/iccad/iccad1993.html#Rudell93`. (Cited on page 28.)

Bernhard Rumpe. Agile Test-based Modeling. In **Software Engineering Research and Practice**, pages 10–15, 2006. (Cited on page 348.)

Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An Approach to Symbolic Test Generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, **IFM**, volume 1945 of **Lecture Notes in Computer Science**, pages 338–357. Springer, 2000. ISBN 3-540-41196-8. URL `http://dblp.uni-trier.de/db/conf/ifm/ifm2000.html#RusuBJ00`. (Cited on pages 40, 234, 251, and 287.)

J.J.M.M. Rutten, Marta Kwiatkowska, Gethin Norman, and David Parker. **Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems**, volume 23 of **CRM Monograph Series**. American Mathematical Society, 2004. (Cited on page 113.)

Lukas Rytz and Martin Odersky. Named and default arguments for polymorphic object-oriented languages: a discussion on the design implemented in the Scala language. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, **Proceedings of the ACM Symposium on Applied Computing (SAC), Sierre, Switzerland**, pages 2090–2095. ACM, March 2010. ISBN 978-1-60558-639-7. URL `http://dblp.uni-trier.de/db/conf/sac/sac2010.html#RytzO10`. (Cited on page 40.)

R.T. Saad, S. Zilio, and B. Berthomieu. An Experiment on Parallel Model Checking of a CTL Fragment. In **ATVA**, volume 7561 of **LNCS**, pages 284–299. Springer, '12. ISBN 978-3-642-33385-9. doi: 10.1007/978-3-642-33386-6_23. URL `http://dx.doi.org/10.1007/978-3-642-33386-6_23`. (Cited on page 153.)

Peter Sanders. **Load Balancing Algorithms for Parallel Depth First Search (Lastverteilungsalgorithmen für parallele Tiefensuche).** PhD thesis, KIT, 1997. (Cited on page 57.)

Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In Tarek F. Abdelzaher, Thiemo Voigt, and Adam Wolisz, editors, **IPSN**, pages 186–196. ACM, 2010. ISBN 978-1-60558-988-6. URL `http://dblp.uni-trier.de/db/conf/ipsn/ipsn2010.html#SasnauskasLAWKW10`. (Cited on page 176.)

SC 7, JTC 1. ISO 8807:1989 Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour, 1989. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=16258`. (Cited on pages 43, 247, and 248.)

Ina Schieferdecker and Theofanis Vassiliou-Gioles. Realizing Distributed TTCN-3 Test Systems with TCI. In Dieter Hogrefe and Anthony Wiles, editors, **TestCom**, volume 2644 of **Lecture Notes in Computer Science**, pages 95–109. Springer, 2003. ISBN 3-540-40123-7. URL `http://dblp.uni-trier.de/db/conf/pts/testcom2003.html#SchieferdeckerV03`. (Cited on pages 248 and 275.)

Peter H. Schmitt. Formale Systeme 2, 2012a. (Cited on page 83.)

Peter H. Schmitt. Formale Systeme, 2012b. (Cited on page 81.)

Peter H. Schmitt and Isabel Tonin. Verifying the Mondex Case Study. In **Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)**, pages 47–58. IEEE Computer Society, 2007. ISBN 978-0-7695-2884-7. (Cited on pages 3 and 39.)

Philippe Schnoebelen. The Complexity of Temporal Logic Model Checking. In Philippe Balbiani, Nobu-Yuki Suzuki, Frank Wolter, and Michael Zakharyaschev, editors, **Advances in Modal Logic**, pages 393–436. King's College Publications, 2002. ISBN 0-9543006-2-9. (Cited on pages 97, 101, and 119.)

Johann Schumann. **Automated theorem proving in software engineering**. Springer, 2001. ISBN 978-3-540-67989-9. (Cited on page 3.)

Thomas R. W. Scogland and Wu chun Feng. Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures. In Lizy K. John, Connie U. Smith, Kai Sachs, and Catalina M. Llado, editors, **ICPE**, pages 63–74. ACM, 2015. ISBN 978-1-4503-3248-4. (Cited on page 143.)

Roberto Sebastiani. Lazy satisfiability modulo theories. **Journal on Satisfiability, Boolean Modeling and Computation**, 3:141–224, 2007. (Cited on pages 31 and 55.)

Dirk Seifert, Jeanine Souquières, et al. Using UML Protocol State Machines in Conformance Testing of Components. **Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan**, October 2008. (Cited on page 287.)

K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. Technical Report UIUCDCS-R-2005-2597, UIUC, 2005a. (Cited on page 175.)

Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald Gall, editors, **ESEC/SIGSOFT FSE**, pages 263–272. ACM, 2005b. ISBN 1-59593-014-0. URL `http://dblp.uni-trier.de/db/conf/sigsoft/fse2005.html#SenMA05`. (Cited on pages 173 and 174.)

Muhammad Shafique and Yvan Labiche. A systematic review of model based testing tool support. Technical report, Carleton University, Canada, 2010. (Cited on page 247.)

Muhammad Shafique and Yvan Labiche. A systematic review of state-based test tools. **International Journal on Software Tools for Technology Transfer**, pages 1–18, 2013. (Cited on page 247.)

Ali Shahrokni and Robert Feldt. RobusTest: Towards a Framework for Automated Testing of Robustness in Software. In **Proceedings of the Third International Conference on Advances in System Testing and Validation Lifecycle (VALID)**, pages 78–83, 2011. (Cited on page 232.)

Oliver Sharma, Jonathan Lewis, Alice Miller, Al Dearle, Dharini Balasubramaniam, Ron Morrison, and Joe Sventek. **Model Checking Software**, volume 5578/2009 of **Lecture Notes in Computer Science**, chapter Towards Verifying Correctness of Wireless Sensor Network Applications Using Insense and Spin, pages 223–240. Springer Berlin / Heidelberg, 2009. doi: 10.1007/978-3-642-02652-2. (Cited on page 177.)

Man-Tak Shing and Doron Drusinsky. Architectural design, behavior modeling and run-time verification of network embedded systems. In **Proceedings of the 12th Monterey conference on Reliable systems on unreliable networked platforms**, pages 281–303, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71155-1. URL `http://dl.acm.org/citation.cfm?id=1765571.1765586`. (Cited on page 167.)

James Shore and Shane Warden. **The art of agile development**. O'Reilly, 2007. ISBN 9780596527679. (Cited on page 342.)

F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In **VIII International Symposium on Software Metrics (METRICS'02)**, pages 249–258, Washigton, DC, USA, June 2002. IEEE Computer Society. doi: 10.1109/METRIC.2002.1011343. (Cited on pages 1 and 18.)

Stephen F. Siegel. Transparent partial order reduction. **Formal Methods in System Design**, 40(1):1–19, 2012. URL `http://dblp.uni-trier.de/db/journals/fmsd/fmsd40.html#Siegel12`. (Cited on pages 104 and 137.)

Stephen F. Siegel and Ganesh Gopalakrishnan. Formal Analysis of Message Passing - (Invited Talk). In Ranjit Jhala and David A. Schmidt, editors, **VMCAI**, volume 6538 of **Lecture Notes in Computer Science**, pages 2–18. Springer, 2011. ISBN 978-3-642-18274-7. URL `http://dblp.uni-trier.de/db/conf/vmcai/vmcai2011.html#SiegelG11`. (Cited on pages 41 and 48.)

Marten Sijtema, Axel Belinfante, Mariëlle Stoelinga, and Lawrence Marinelli. Experiences with formal engineering: Model-based specification, implementation and testing of a software bus at Neopost. **Sci. Comput. Program.**, 80:188–209, 2014. URL `http://dblp.uni-trier.de/db/journals/scp/scp80.html#SijtemaBSM14`. (Cited on pages 19, 249, 250, 341, and 355.)

Anthony J. H. Simons. JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. **Autom. Softw. Eng.**, 14(4):369–418, 2007. (Cited on page 55.)

Anthony JH Simons, Neil Griffiths, and Christopher Thomson. Feedback-based specification, coding and testing with JWalk. In **Practice and Research Techniques, (TAIC PART 2008). Testing: Academic & Industrial Conference**, pages 69–73. IEEE, 2008. (Cited on page 55.)

Carsten Sinz, Florian Merz, and Stephan Falke. LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation - (Competition Contribution). In Cormac Flanagan and Barbara König, editors, **TACAS**, volume 7214 of **Lecture Notes in Computer Science**, pages 542–544. Springer, 2012. ISBN 978-3-642-28755-8. URL `http://dblp.uni-trier.de/db/conf/tacas/tacas2012.html#SinzMF12`. (Cited on page 172.)

A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The Complementation Problem for Büchi Automata with Applications to Temporal Logic (Extended Abstract). In Wilfried Brauer, editor, **ICALP**, volume 194 of **Lecture Notes in Computer Science**, pages 465–474. Springer, 1985. ISBN 3-540-15650-X. (Cited on page 79.)

Douglas R. Smith. Top-Down Synthesis of Divide-and-Conquer Algorithms. **Artif. Intell.**, 27(1):43–96, 1985. (Cited on page 54.)

Ian Sommerville. **Software Engineering**. Addison-Wesley, 9 edition, 2010. (Cited on page 19.)

Niklas Sörensson. **Effective SAT solving**. PhD thesis, 2008. (Cited on page 27.)

Andreas Spillner and Tilo Linz. **Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard**. dpunkt, Heidelberg, 3. edition, 2005. ISBN 978-3-89864-358-0. (Cited on page 17.)

Matt Staats, Gregory Gay, Michael W. Whalen, and Mats P.E. Heimdahl. On the Danger of Coverage Directed Test Case Generation. In Juan de Lara and Andrea Zisman, editors, **Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)**, volume 7212 of **Lecture Notes in Computer Science**, pages 409–424. Springer, 2012. ISBN 978-3-642-28871-5. (Cited on pages 23 and 287.)

Herbert Stachowiak. **Allgemeine Modelltheorie**. Springer-Verlag, Wien New York, 1973. (Cited on page 59.)

Eric Starkloff. Designing a parallel, distributed test system. In **AUTOTESTCON Proceedings, 2000 IEEE**, pages 564–567. IEEE, 2000. (Cited on pages 272 and 273.)

Willem Gerrit Johan Stokkink, Mark Timmer, and Mariëlle Stoelinga. Divergent Quiescent Transition Systems. In Margus Veanes and Luca Vigano, editors, **Proceedings of the 7th International Conference on Tests And Proofs (TAP)**, volume 7942 of **Lecture Notes in Computer Science**, pages 214–231. Springer, 2013. ISBN 978-3-642-38915-3. URL `http://dblp.uni-trier.de/db/conf/tap/tap2013.html#StokkinkTS13`. (Cited on page 193.)

Ting Su, Zhoulai Fu, Geguang Pu, Jifeng He, and Zhendong Su. Combining Symbolic Execution and Model Checking for Data Flow Testing. In **37th International Conference on Software Engineering, ICSE**, volume 15, 2015a. (Cited on pages 22, 173, and 334.)

Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. A Survey on Data Flow Testing. **Technical Report**, 2015b. URL `https://tingsu.github.io/files/dft-survey.pdf`. (Cited on pages 22, 173, and 288.)

Ramesh Subramanian and Brian D Goodman. **Peer-to-peer computing: the evolution of a disruptive technology**. IGI Global, 2005. (Cited on page 52.)

Suprapto Suprapto and Reza Pulungan. A Framework for An LTS Semantics for PROMELA. **Proceedings of the 6th SEAMS-UGM Conference, Computer, Graph and Combinatorics**, pages 631–646, 2011. (Cited on pages 46 and 47.)

Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. **Dr. Dobb's Journal**, 30(3):202–210, 2005. URL `http://www.gotw.ca/publications/concurrency-ddj.htm`. (Cited on page 47.)

Herb Sutter. Going Superlinear. **Dr.Dobb's**, January 2008. URL `http://www.drdobbs.com/cpp/going-superlinear/206100542`. (Cited on page 151.)

Herb Sutter. Sharing Is the Root of All Contention. **Dr. Dobb's**, February 2009. URL `http://www.drdobbs.com/architecture-and-design/sharing-is-the-root-of-all-contention/214100002?pgno=1`. (Cited on pages 48, 50, 143, and 144.)

Ranjita Kumari Swain, Prafulla Kumar Behera, and Durga Prasad Mohapatra. Minimal TestCase Generation for Object-Oriented Software with State Charts. **CoRR**, abs/1208.2265, 2012. URL `http://dblp.uni-trier.de/db/journals/corr/corr1208.html#abs-1208-2265`. (Cited on page 284.)

Hirotaka Takeuchi and Ikujiro Nonaka. The New New Product Development Game. **Harvard Business Review**, 1986. URL `http://apln-richmond.pbwiki.com/f/NewNewProdDevelGame.pdf`. (Cited on page 342.)

Zhi-Hong Tao, Cong-Hua Zhou, Zhong Chen, and Li-Fu Wang. Bounded Model Checking of CTL. **J. Comput. Sci. Technol.**, 22(1):39–43, 2007. URL `http://dblp.uni-trier.de/db/journals/jcst/jcst22.html#TaoZCW07`. (Cited on page 93.)

Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. **SIAM J. Comput.**, 1(2):146–160, 1972. (Cited on pages 97 and 101.)

G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, May 2002. (Cited on page 1.)

The "C3 Team". Chrysler Goes to "Extremes". **Distributed Computing**, pages 24–26, 1998. (Cited on page 342.)

Scott Tilley and Tauhida Parveen. HadoopUnit: Test Execution in the Cloud. In **Software Testing in the Cloud**, pages 37–53. Springer, 2012. (Cited on page 273.)

Scott Robert Tilley and Tauhida Parveen, editors. **Software Testing in the Cloud: Perspectives on an Emerging Discipline**. IGI Global, 2013. (Cited on pages 272, 273, 275, and 278.)

Mark Timmer, Marielle Stoelinga, and Jaco van de Pol. Confluence Reduction for Probabilistic Systems. In **TACAS**, pages 311–325, 2011. (Cited on page 103.)

Anthony Widjaja To. **Model Checking Infinite-State Systems: Generic and Specific Approaches.** PhD thesis, University of Edinburgh, 2010. URL `http://homepages.inf.ed.ac.uk/s0673998/papers/phdthesis.pdf`. (Cited on pages 94 and 246.)

G. J. Tretmans and H. Brinksma. TorX: Automated Model-Based Testing. In A. Hartman and K. Dussa-Ziegler, editors, **First European Conference on Model-Driven Software Engineering, Nuremberg, Germany**, pages 31–43, December 2003. ISBN not assigned. (Cited on page 248.)

Jan Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. **Software - Concepts and Tools**, 17(3):103–120, 1996. URL `http://dblp.uni-trier.de/db/journals/stp/stp17.html#Tretmans96`. (Cited on pages 212 and 227.)

Jan Tretmans. Model Based Testing with Labelled Transition Systems. In **Formal Methods and Testing**, pages 1–38. Springer, 2008. doi: 10.1007/978-3-540-78917-8_1. URL http://dx.doi.org/10.1007/978-3-540-78917-8_1. (Cited on pages 41, 183, 185, 186, 187, 191, 193, 196, 199, 202, 203, 208, 211, 221, 225, 227, 228, and 279.)

Bernhard Turban. **Tool-Based Requirement Traceability between Requirement and Design Artifacts**. Springer, 2011. (Cited on pages 18, 19, and 333.)

Alan M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. **Proceedings of the London Mathematical Society**, 42:230–265, 1936. (Cited on pages 3, 25, 30, and 325.)

Richard Turner. Toward Agile Systems Engineering Processes. **Crosstalk: Journal of Defence Software Engineering**, April 2007. URL http://www.stsc.hill.af.mil/CrossTalk/2007/04/0704Turner.html. (Cited on page 19.)

Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, **PPOPP**, pages 179–190. ACM, 2010. ISBN 978-1-60558-877-3. (Cited on pages 50 and 57.)

Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism. **ACM Trans. Program. Lang. Syst.**, 36(3):10:1–10:51, 2014. (Cited on pages 47, 50, 54, and 57.)

Abhishek Udupa, Ankush Desai, and Sriram K. Rajamani. Depth Bounded Explicit-State Model Checking. In Alex Groce and Madanlal Musuvathi, editors, **SPIN**, volume 6823 of **Lecture Notes in Computer Science**, pages 57–74. Springer, 2011. ISBN 978-3-642-22305-1. URL http://dblp.uni-trier.de/db/conf/spin/spin2011.html#UdupaDR11. (Cited on page 93.)

Mattias Ulbrich. **Dynamic Logic for an Intermediate Language: Verification, Interaction and Refinement**. PhD thesis, Karlsruhe Institute of Technology, 2014. (Cited on pages 60 and 177.)

URL:ACMAWARD. ACM Awards - Software System Award. web page, 2013. URL http://awards.acm.org/software_system/. Accessed: Okt 2013. (Cited on pages 108 and 172.)

URL:AMAZONCLOUD. Amazon Elastic Compute Cloud. web page, 2013. URL http://aws.amazon.com/de/ec2/. Accessed: Dec 2013. (Cited on page 278.)

URL:ARC. ARC - AltaRica Checker homepage. web page, 2013. URL https://altarica.labri.fr/forge/projects/arc/wiki. Accessed: March 2013. (Cited on page 101.)

URL:BoogiePL. BoogiePL: A typed procedural language for checking object-oriented programs - Microsoft Research. web page, 2013. URL http://research.microsoft.com/apps/pubs/default.aspx?id=70179. Accessed: June 2013. (Cited on pages 173 and 177.)

URL:BS79251. BS 7925-1 Glossary of Software Testing Terms. web page, 2014. URL `http://www.testingstandards.co.uk/bs_7925-1.htm`. Accessed: Feb 2014. (Cited on page 13.)

URL:CADP. Construction and Analysis of Distributed Processes - Software Tools for Designing Reliable Protocols and Systems. web page, 2012. URL `http://www.inrialpes.fr/vasy/cadp/`. Accessed: March 2012. (Cited on pages 39, 247, and 248.)

URL:CBMC. CBMC - Bounded Model Checking for ANSI-C. web page, 2013. URL `http://www.cprover.org/cbmc/`. Accessed: February 2013. (Cited on page 158.)

URL:C++CSP. C++CSP2 - Easy Concurrency for C++. web page, 2012. URL `http://www.cs.kent.ac.uk/projects/ofa/c++csp/`. Accessed: March 2012. (Cited on page 43.)

URL:CONFORMIQ. Conformiq - Automated Test Design. web page, 2014. URL `http://www.conformiq.com`. Accessed: Jan 2014. (Cited on page 251.)

URL:DIMENSIONDATA. Dimension Data Cloud Services Homepage. web page, 2013. URL `http://www.dimensiondata.com/`. Accessed: Nov 2013. (Cited on page 53.)

URL:DIVINE. DIVINE – Parallel LTL Model Checker homepage. web page, 2013. URL `http://divine.fi.muni.cz/`. Accessed: April 2013. (Cited on pages 101 and 112.)

URL:EclEHP. EclEmma - Java Code Coverage for Eclipse. web page, 2015. URL `http://eclemma.org`. Accessed: Okt 2015. (Cited on page 328.)

URL:EclFAQHP. JaCoCo - Coverage Counter. web page, 2015. URL `http://eclemma.org/jacoco/trunk/doc/counters.html`. Accessed: Okt 2015. (Cited on page 328.)

URL:EmersonSmartWireless. Emerson Smart Wireless. web page, 2010. URL `http://www2.emersonprocess.com/en-US/plantweb/wireless/Pages/WirelessHomePage.aspx`. Accessed: June 2010. (Cited on page 161.)

URL:ETSIglossaryHP. ETSI, Centre For Testing and Interoperability: Glossary. web page, 2015. URL `https://portal.etsi.org/CTI/CTISupport/Glossary.htm`. Accessed: Jun 2015. (Cited on page 13.)

URL:experimentLOTFHP. Experiments for LazyOTF. web page, 2015. URL `http://doiop.com/experimentLOTF`. Accessed: Nov 2015. (Cited on pages 369 and 372.)

URL:F-Soft. NEC laboratories America, Systems Analysis & Verification. web page, 2013. URL `http://www.nec-labs.com/~fsoft`. Accessed: June 2013. (Cited on page 177.)

URL:FuturesPromisesScalaHP. Futures and Promises - Scala Documentation. web page, 2015. URL `http://docs.scala-lang.org/overviews/core/futures.html`. Accessed: Jun 2015. (Cited on page 49.)

URL:GI-TAV TOOP. Arbeitskreit TOOP/MBT: Testen objektorientierter Programme/Model-Based Testing. web page, 2013. URL `http://www1.gi-ev.de/fachbereiche/softwaretechnik/tav/toop/`. Accessed: Sept 2013. (Cited on page iii.)

URL:GOGRID. GoGrid Cloud Platform. web page, 2013. URL `http://www.gogrid.com/cloud-platform`. Accessed: Nov 2013. (Cited on page 53.)

URL:GRAPHML. The GraphML File Format. web page, 2014. URL `http://graphml.graphdrawing.org/`. Accessed: Jan 2014. (Cited on page 249.)

URL:GRAPHVIZ. Graphviz - Graph Visualization Software. web page, 2014. URL `http://www.graphviz.org/`. Accessed: Jan 2014. (Cited on page 249.)

URL:HADOOP. Apache Hadoop Homepage. web page, 2013. URL `http://hadoop.apache.org/`. Accessed: Dec 2013. (Cited on pages 52 and 273.)

URL:Hazel. Hazelcast - The Leading Open Source In-Memory Data Grid. web page, 2014. URL `http://www.hazelcast.org/`. Accessed: Mar 2014. (Cited on pages 53 and 354.)

URL:IntelMIMDHP. MIMD - Intel Developer Zone. web page, 2015. URL `https://software.intel.com/en-us/articles/mimd`. Accessed: Aug 2015. (Cited on page 48.)

URL:ISO26262. ISO Online Browsing Platform - ISO 26262-1:2011(en). web page, 2014. URL `https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en`. Accessed: May 2014. (Cited on pages 4, 13, and 21.)

URL:ISO29119. ISO/IEC/IEEE 29119 Software Testing Standard. web page, 2014. URL `http://www.softwaretestingstandard.org/`. Accessed: Feb 2014. (Cited on page 13.)

URL:JCSP. Communicating Sequential Processes for JavaTM (JCSP). web page, 2012. URL `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`. (March 2012). (Cited on page 43.)

URL:JML. The Java Modeling Language (JML). web page, 2012. URL `http://www.cs.ucf.edu/~leavens/JML/`. Accessed: February 2012. (Cited on page 44.)

URL:JPF. Java PathFinder homepage. web page, 2013. URL `http://babelfish.arc.nasa.gov/trac/jpf/`. Accessed: March 2013. (Cited on page 61.)

URL:JTorX. JTorX - a tool for Model-Based Testing. web page, 2012. URL `https://fmt.ewi.utwente.nl/redmine/projects/jtorx`. Accessed: March 2013. (Cited on pages 233 and 249.)

URL:JTorXwiki. JTorX - FMT Tools. web page, 2015. URL `https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki/`. Accessed: Nov 2015. (Cited on pages viii, 7, 338, and 354.)

URL:JUNIT. JUnit homepage. web page, 2013. URL `http://junit.org/`. Accessed: Dec 2013. (Cited on pages 243 and 273.)

URL:KeY. The KeY Project - Integrated Deductive Software Design. web page, 2012. URL `http://www.key-project.org/`. Accessed: March 2013. (Cited on page 2.)

URL:Klee. The KLEE Symbolic Virtual Machine. web page, 2013. URL `http://klee.llvm.org/`. Accessed: June 2013. (Cited on page 174.)

URL:KleeNet. KleeNet homepage. web page, 2013. URL `https://code.comsys.rwth-aachen.de/redmine/projects/kleenet-public`. Accessed: June 2013. (Cited on page 176.)

URL:KoushikSensProjects. Koushik Sen's projects. web page, 2014. URL `http://srl.cs.berkeley.edu/~ksen/doku.php?id=projects`. Accessed: Jul 2014. (Cited on page 174.)

URL:LazyInitCSharpHP. Lazy initialization - .NET Framework 4.6 and 4.5. web page, 2015. URL `https://msdn.microsoft.com/en-us/library/dd997286.aspx`. Accessed: Jun 2015. (Cited on page 54.)

URL:LC. Software Licensing: CodeMeter License Central. web page, 2014. URL `http://www.wibu.com/software-licensing.html`. Accessed: Feb 2014. (Cited on pages 231, 349, 354, and 385.)

URL:leader4DFSFIFOHP. Leader Election Models Used in Improved On-the-fly Livelock Detection. web page, 2015. URL `http://doiop.com/leader4DFSFIFO`. Accessed: Aug 2015. (Cited on pages 145 and 150.)

URL:listFMToolsHP. GitHub list of verification and synthesis tools. web page, 2015. URL `https://github.com/johnyf/tool_lists/blob/master/verification_synthesis.md`. Accessed: Jun 2015. (Cited on page 108.)

URL:listVerifToolsHP. YAHODA: Verification Tools Database. web page, 2015. URL `http://anna.fi.muni.cz/yahoda/`. Accessed: Jun 2015. (Cited on page 108.)

URL:LLBMC. LLBMC – The Low-Level Bounded Model Checker. web page, 2013. URL `http://llbmc.org/`. Accessed: June 2013. (Cited on page 172.)

URL:LLVM. The LLVM Compiler Infrastructure Project. web page, 2012. URL `http://www.llvm.org/`. Accessed: June 2013. (Cited on pages 172 and 174.)

URL:Lombok. Project Lombok homepage. web page, 2013. URL `http://projectlombok.org`. Accessed: March 2013. (Cited on page 54.)

URL:LOTOS. WELL - World-Wide Lotos. web page, 2014. URL `http://www.cs.stir.ac.uk/~kjt/research/well/`. Accessed: Jan 2014. (Cited on pages 247 and 248.)

URL:LTL2BUECHI. A Genealogy of LTL-to-Büchi Translators. web page, 2013. URL `http://spot.lip6.fr/wiki/LtlTranslationAlgorithms`. Accessed: Okt 2013. (Cited on page 88.)

URL:LTSmin. Model Checking and LTS Minimization homepage. web page, 2013. URL `http://fmt.cs.utwente.nl/tools/ltsmin/`. Accessed: April 2013. (Cited on pages viii, 7, 101, and 109.)

URL:MBT-C. Model-Based Testing Community, Connecting the MBT world. web page, 2013. URL `http://model-based-testing.info/`. Accessed: Sept 2013. (Cited on page iii.)

URL:MBTbenchmark. Models. Model-based Testing Community. web page, 2014. URL `http://model-based-testing.info/models/`. Accessed: Mar 2015. (Cited on page 373.)

URL:MBTtoolsHP. Model-based Testing (MBT). web page, 2015. URL `http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html`. Accessed: Jun 2015. (Cited on pages 247 and 251.)

URL:NuSMV. NuSMV homepage. web page, 2013. URL `http://nusmv.fbk.eu/`. Accessed: March 2013. (Cited on page 97.)

URL:occam. occam-pi in a nutshell. web page, 2012. URL `http://pop-users.org/wiki/occam-pi`. Accessed: March 2012. (Cited on page 43.)

URL:omegaprojectHP. Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs. web page, 2015. URL `http://www.cs.umd.edu/projects/omega/`. Accessed: Sept 2015. (Cited on page 335.)

URL:PARALLELblogpost. Parallelization to speed up MBT - Model-based Testing Community. web page, 2013. URL `http://model-based-testing.info/2013/12/25/parallelization-to-speed-up-mbt/`. Accessed: Mar 2014. (Cited on page 274.)

URL:ParCompHP. Introduction to Parallel Computing. web page, 2015. URL `https://computing.llnl.gov/tutorials/parallel_comp/`. Accessed: Aug 2015. (Cited on page 48.)

URL:PBSHP. Global Leader in HPC Workload Management. web page, 2015. URL `http://www.pbsworks.com`. Accessed: Okt 2015. (Cited on page 370.)

URL:PRISM. PRISM - Probabilistic Symbolic Model Checker. web page, 2012. URL `http://www.prismmodelchecker.org/`. Accessed: March 2012. (Cited on pages 43 and 113.)

URL:PromelaDatabase. Promela Database. web page, 2012. URL `http://www.albertolluch.com/research/promelamodels`. Accessed: March 2013. (Cited on pages 44, 116, and 145.)

URL:PROMELAHP. Promela Manual Pages. web page, 2015. URL `http://spinroot.com/spin/Man`. Accessed: Jun 2015. (Cited on pages 44 and 46.)

URL:RACKSPACE. Rackspace: The Leader in Hybrid Cloud. web page, 2013. URL `http://www.rackspace.com/`. Accessed: Nov 2013. (Cited on page 53.)

URL:recomputationHP. recomputation.org. web page, 2015. URL `http://www.recomputation.org`. Accessed: Okt 2015. (Cited on page 354.)

URL:REDHATCVE. Red Hat Common Vulnerabilities and Exposures Database. web page, 2013. URL `https://access.redhat.com/security/cve/`. Accessed: Nov 2013. (Cited on page 2.)

URL:ROIblogpost. Articles about ROI of MBT. web page, 2013. URL `http://model-based-testing.info/2013/10/21/articles-about-roi-of-mbt/`. Accessed: Mar 2014. (Cited on page 341.)

URL:SAL. Symbolic Analysis Laboratory homepage. web page, 2013. URL `http://sal.csl.sri.com/index.shtml`. Accessed: March 2013. (Cited on page 101.)

URL:SATABS. SATABS: Predicate Abstaction using SAT. web page, 2013. URL `http://www.cprover.org/satabs`. Accessed: June 2013. (Cited on page 177.)

URL:SATChallenge2012. SAT Challenge 2012 Results Application SAT+UNSAT. web page, 2014. URL `http://baldur.iti.kit.edu/SAT-Challenge-2012/results.html#main_app`. Accessed: Jul 2014. (Cited on page 27.)

URL:SATcompetition2014HP. SAT Competition 2014. web page, 2015. Accessed: Jul 2015. (Cited on page 27.)

URL:ScalaByNameHP. Scala - Basic Declarations and Definitions - By-Name Parameters. web page, 2015. URL `http://www.scala-lang.org/files/archive/spec/2.11/04-basic-declarations-and-definitions.html#by-name-parameters`. Accessed: Jun 2015. (Cited on page 54.)

URL:SELENIUM. Selenium - Web Browser Automation. web page, 2013. URL `http://www.seleniumhq.org/`. Accessed: Dec 2013. (Cited on page 273.)

URL:sigarHP. Sigar - Hyperic Support. web page, 2015. URL `https://support.hyperic.com/display/SIGAR/Home`. Accessed: Sept 2015. (Cited on page 371.)

URL:SMT-LIB. SMT-LIB: The Satisfiability Modulo Theories Library. web page, 2013. URL `http://www.smtlib.org/`. Accessed: April 2013. (Cited on page 31.)

URL:SMTCOMPHP. SMT-COMP 2015. web page, 2015. URL `http://smtcomp.sourceforge.net`. Accessed: Jun 2015. (Cited on page 47.)

URL:SMV. The SMV system. web page, 2015. URL `http://www.cs.cmu.edu/~modelcheck/smv.html`. Accessed: Aug 2015. (Cited on page 101.)

URL:Spec Explorer. Spec Explorer 2010 Visual Studio Power Tool. web page, 2012. URL `http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745/`. Accessed: January 2013. (Cited on pages 251 and 335.)

URL:Spin. SPIN - on-the-fly, LTL model checking. web page, 2013. URL `http://spinroot.com`. Accessed: March 2013. (Cited on pages 101 and 108.)

URL:STSimulator. STSimulator homepage. web page, 2014. URL `http://java.net/projects/stsimulator`. Accessed: January 2014. (Cited on pages 252 and 325.)

URL:SureCross. SureCross Wireless Industrial I/O Sensor Network Applications. web page, 2010. URL `http://www.bannerengineering.com/en-US/wireless/surecross_web_appnotes`. Accessed: June 2010. (Cited on page 161.)

URL:TinyOS. TinyOS: An open-source OS for the networked sensor regime. web page, 2010. URL `http://www.tinyos.net`. Accessed: June 2010. (Cited on page 163.)

URL:TorX. The TorX Test Tool website. web page, 2012. URL `https://fmt.ewi.utwente.nl/tools/torx/`. Accessed: February 2012. (Cited on page 248.)

URL:TTCN. ETSI'S OFFICIAL TTCN-3 HOMEPAGE. web page, 2013. URL `http://www.ttcn-3.org`. Accessed: Dec 2013. (Cited on page 275.)

URL:VCC. VCC: A C Verifier – Microsoft Research. web page, 2013. URL `http://research.microsoft.com/en-us/projects/vcc/`. Accessed: June 2013. (Cited on page 173.)

URL:Verialg. KIT Research Group "Verification Meets Algorithm Engineering". web page, 2014. URL `https://verialg.iti.kit.edu/english/index.php`. Accessed: Mar 2014. (Cited on page 354.)

URL:VirtualBox. Oracle VM VirtualBox. web page, 2014. URL `https://www.virtualbox.org/`. Accessed: Mar 2014. (Cited on page 354.)

URL:whiteboxToolsHP. Code-based Test Generation. web page, 2015. URL `http://mit.bme.hu/~micskeiz/pages/code_based_test_generation.html`. Accessed: Jun 2015. (Cited on page 173.)

URL:WS. W3C - Web Service Architecture. web page, 2014. URL `http://www.w3.org/TR/2002/WD-ws-arch-20021114/`. Accessed: Jan 2014. (Cited on page 53.)

URL:WSBPEL. OASIS- Web Services Business Process Execution Language Version 2.0. web page, 2014. URL `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`. Accessed: Jan 2014. (Cited on page 53.)

URL:WSDL. W3C - Web Service Description Language. web page, 2014. URL `http://www.w3.org/TR/wsdl`. Accessed: Jan 2014. (Cited on page 53.)

URL:WSDLS. W3C - Web Service Semantics - WSDL-S. web page, 2014. URL `http://www.w3.org/Submission/WSDL-S/`. Accessed: Jan 2014. (Cited on page 53.)

URL:XeonHP. Intel Xeon Processor E5430 Specifications. web page, 2015. URL `http://ark.intel.com/products/33081/Intel-Xeon-Processor-E5430-12M-Cache-2_66-GHz-1333-MHz-FSB`. Accessed: Sept 2015. (Cited on page 388.)

URL:XTEXT. Xtext - Language Development Made Easy! web page, 2014. URL `http://www.eclipse.org/Xtext/`. Accessed: Jan 2014. (Cited on page 233.)

URL:yEdHP. yEd - Graph Editor. web page, 2015. URL `http://www.yworks.com/en/products/yfiles/yed/`. Accessed: Okt 2015. (Cited on page 385.)

Mark Utting and Bruno Legeard. **Practical Model-Based Testing: A Tools Approach**. Morgan Kaufmann, 1 edition, 2007. ISBN 0123725011. (Cited on pages 14, 17, 18, 19, 21, 22, 23, 241, 244, 247, 251, 252, 259, 263, 287, and 347.)

Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical report, The University of Waikato, April 2006. URL `http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf`. (Cited on pages 240, 241, and 377.)

Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, **Applications and Theory of Petri Nets**, volume 483 of **Lecture Notes in Computer Science**, pages 491–515. Springer, 1989. ISBN 3-540-53863-1. URL `http://dblp.uni-trier.de/db/conf/apn/apn1989.html#Valmari89`. (Cited on page 103.)

Antti Valmari. On-the-Fly Verification with Stubborn Sets. In Costas Courcoubetis, editor, **CAV**, volume 697 of **Lecture Notes in Computer Science**, pages 397–408. Springer, 1993. ISBN 3-540-56922-7. (Cited on pages 131, 137, and 377.)

F. I. van der Berg and A. W. Laarman. SpinS: Extending LTSmin with Promela through SpinJa. In K. Heljanko and W. J. Knottenbelt, editors, **11th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2012, London, UK**, Electronic Notes in Theoretical Computer Science, Amsterdam, September 2012. Elsevier. (Cited on pages 46, 110, and 111.)

Machiel van der Bijl and Fabien Peureux. I/O-automata Based Testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, **Model-Based Testing of Reactive Systems**, volume 3472 of **Lecture Notes in Computer Science**, pages 173–200. Springer, 2004. ISBN 3-540-26278-4. URL `http://dblp.uni-trier.de/db/conf/dagstuhl/test2004.html#BijlP04`. (Cited on page 227.)

Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Action Refinement in Conformance Testing. In Ferhat Khendek and Rachida Dssouli, editors, **TestCom**, volume 3502 of **Lecture Notes in Computer Science**, pages 81–96. Springer, 2005. ISBN 3-540-26054-4. URL `http://dblp.uni-trier.de/db/conf/pts/testcom2005.html#BijlRT05`. (Cited on pages 60, 207, 208, and 234.)

T. van Dijk, A. W. Laarman, and J. C. van de Pol. Multi-Core BDD Operations for Symbolic Reachability. In K. Heljanko and W. J. Knottenbelt, editors, **11th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2012, London, UK**, Electronic Notes in Theoretical Computer Science, Amsterdam, September 2012. Elsevier. (Cited on page 57.)

Moshe Y. Vardi. Branching vs. Linear Time: Final Showdown. In Tiziana Margaria and Wang Yi, editors, **TACAS**, volume 2031 of **Lecture Notes in Computer Science**,

pages 1–22. Springer, 2001. ISBN 3-540-41865-2. URL http://dblp.uni-trier.de/db/conf/tacas/tacas2001.html#Vardi01. (Cited on pages 95 and 96.)

Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In **LICS**, pages 332–344. IEEE Computer Society, 1986. ISBN 0-8186-0720-3. (Cited on page 97.)

Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, **Formal Methods and Testing**, volume 4949 of **Lecture Notes in Computer Science**, pages 39–76. Springer, 2008. ISBN 978-3-540-78916-1. URL http://dblp.uni-trier.de/db/conf/fortest/fortest2008.html#VeanesCGSTN08. (Cited on pages 233, 251, 287, and 335.)

R Venkatesh, Ulka Shrotri, Priyanka Darke, and Prasad Bokil. Test generation for large automotive models. In **Industrial Technology (ICIT), 2012 IEEE International Conference on**, pages 662–667. IEEE, 2012. (Cited on page 175.)

Sergiy Vilkomir. Cloud Testing: A State-of-the-Art Review. **Information & Security: An International Journal**, 28(2):213–222, 2012. (Cited on page 272.)

Willem Visser and Howard Barringer. Practical CTL* Model Checking: Should SPIN be Extended? **STTT**, 2(4):350–365, 2000. URL http://dblp.uni-trier.de/db/journals/sttt/sttt2.html#VisserB00. (Cited on page 101.)

Willem Visser and Peter C. Mehlitz. Model Checking Programs with Java PathFinder. In Patrice Godefroid, editor, **SPIN**, volume 3639 of **Lecture Notes in Computer Science**, page 27. Springer, 2005. ISBN 3-540-28195-9. URL http://dblp.uni-trier.de/db/conf/spin/spin2005.html#VisserM05. (Cited on page 173.)

Oliver Vogel, Ingo Arnold, Arif Chughtai, and Timo Kehrer. **Software Architecture - A Comprehensive Framework and Guide for Practitioners.** Springer, 2011. ISBN 978-3-642-19735-2. (Cited on page 52.)

Michele Volpato and Jan Tretmans. Towards Quality of Model-Based Testing in the ioco Framework. **QuoMBaT**, July 2013. (Cited on pages 187, 203, 217, 228, and 234.)

W3C. Transport Message Exchange Pattern: Single-Request-Response. Technical report, World Wide Web Consortium, October 2001. URL http://www.w3.org/2000/xp/Group/1/10/11/2001-10-11-SRR-Transport_MEP. (Cited on pages 349 and 385.)

Yusuke Wada and Shigeru Kusakabe. Performance Evaluation of A Testing Framework Using QuickCheck and Hadoop. **Journal of Information Processing**, 20(2):340–346, 2012. doi: 10.2197/ipsjjip.20.340. URL http://dx.doi.org/10.2197/ipsjjip.20.340. (Cited on page 274.)

Nick Walther. Generierung Von Testsequenzen Für Die Klassifikationsbaum-Methode. Master's thesis, Humboldt-Universität zu Berlin, 2011. (Cited on page 81.)

Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole Partial Order Reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, **TACAS**, volume 4963 of **Lecture Notes in Computer Science**, pages 382–396. Springer, 2008. ISBN 978-3-540-78799-0. URL `http://dblp.uni-trier.de/db/conf/tacas/tacas2008.html#WangYKG08`. (Cited on page 103.)

Yong Wang. A Survey on Formal Methods for Workflow-Based Web Service Composition. **CoRR**, abs/1306.5535, 2013. (Cited on page 53.)

David A. Watt and William Findlay. **Programming language design concepts.** Wiley, 2004. ISBN 978-0-470-85320-7. (Cited on page 54.)

Andreas Weber. Modularization and Optimization for the SBMC-Algorithm of LLBMC. Bachelor's thesis, Karlsruhe Institute of Technology, 2014. (Cited on page 8.)

Carsten Weise. An incremental formal semantics for PROMELA. In **Proceedings of the 3rd International SPIN Workshop**, 1997. (Cited on page 46.)

Benjamin Weiß. **Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction**. KIT Scientific Publishing, 2010. (Cited on page 3.)

Stephan Weißleder. **Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines**. PhD thesis, Humbold-Universität zu Berlin, 2009. (Cited on pages 2, 14, 15, 17, 19, 20, 21, 22, 23, 237, 241, 252, 287, and 346.)

Stephan Weißleder, Baris Güldali, Michael Mlynarski, Arne-Michael Törsel, David Farag'o, Florian Prester, and Mario Winter. Modellbasiertes Testen: Hype oder Realität? **OBJEKTspektrum**, 6:59–65, Oktober 2011. (Cited on pages v, 1, 2, 8, 208, and 341.)

Frank Werner. **Applied Formal Methods in Wireless Sensor Networks.** PhD thesis, Karlsruhe Institute of Technology, 2009. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000012355`. http://d-nb.info/1014220807. (Cited on pages 156 and 160.)

Frank Werner and David Faragó. Correctness of Sensor Network Applications by Software Bounded Model Checking. In **Formal Methods for Industrial Critical Systems - 15th International Workshop, FMICS 2010, Antwerp, Belgium, September 20-21, 2010. Proceedings**, LNCS, pages 115–131. Springer, 2010. (Cited on pages 7, 155, 160, and 175.)

Frank Werner and Peter H Schmitt. Analysis of the authenticated query flooding protocol by probabilistic means. In **Wireless on Demand Network Systems and Services, 2008. WONS 2008. Fifth Annual Conference on**, pages 101–104. IEEE, 2008. (Cited on page 113.)

Frank Werner and Raoul Steffen. Modeling Security Aspects of Network Aggregation Protocols. In **8. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze**, pages

83–86, Hamburg, August 2009. URL http://doku.b.tu-harburg.de/volltexte/2009/581/. (Cited on pages 177 and 179.)

Linda Westfall. Software requirements engineering: what, why, who, when, and how. **Software Quality Professional**, 7(4):17, 2005. (Cited on page 18.)

Elaine J. Weyuker and Bingchiang Jeng. Analyzing Partition Testing Strategies. **IEEE Transactions on Software Engineering**, 17(7):703–711, July 1991. (Cited on pages 23 and 287.)

Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In Lori L. Pollock and Mauro Pezze, editors, **ISSTA**, pages 25–36. ACM, 2006. ISBN 1-59593-263-1. (Cited on pages 14 and 22.)

Michael W. Whalen, Gregory Gay, Dongjiang You, Mats P.E. Heimdahl, and Matt Staats. Observable Modified Condition/Decision Coverage. In **Proceedings of the 2013 International Conference on Software Engineering**. ACM, May 2013. (Cited on page 21.)

Tom White. **Hadoop: The Definitive Guide**. Oreilly and Associate Series. O'Reilly, 2012. ISBN 9781449311520. URL http://books.google.de/books?id=drbI_aro20oC. (Cited on pages 52 and 273.)

Rand R. Wilcox. **Fundamentals of Modern Statistical Methods: Substantially Improving Power and Accuracy**. Springer, 2010. ISBN 9781441955241. URL http://books.google.de/books?id=uUNGzhdxk0kC. (Cited on page 371.)

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. **Experimentation in Software Engineering.** Springer, 2012. ISBN 978-3-642-29043-5. (Cited on pages 369 and 371.)

Pierre Wolper. Temporal Logic Can Be More Expressive. **Information and Control**, 56(1/2):72–99, January/February 1983. URL http://dblp.uni-trier.de/db/journals/iandc/iandc56.html#Wolper83. (Cited on page 80.)

Pierre Wolper. On the Relation of Programs and Computations to Models of Temporal Logic. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, **Temporal Logic in Specification**, volume 398 of **Lecture Notes in Computer Science**, pages 75–123. Springer, 1987. ISBN 3-540-51803-7. URL http://dblp.uni-trier.de/db/conf/tls/tls1987.html#Wolper87. (Cited on page 88.)

Pierre Wolper. Constructing Automata from Temporal Logic Formulas: A Tutorial. In Ed Brinksma, Holger Hermanns, and Joost-Pieter Katoen, editors, **European Educational Forum: School on Formal Methods and Performance Analysis**, volume 2090 of **Lecture Notes in Computer Science**, pages 261–277. Springer, 2000. ISBN 3-540-42479-2. URL http://dblp.uni-trier.de/db/conf/eef/eef2000.html#Wolper00. (Cited on page 81.)

W. Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim, and Michael F. Siok. Recent Catastrophic Accidents: Investigating How Software was Responsible. In

SSIRI, pages 14–22. IEEE Computer Society, 2010. ISBN 978-0-7695-4086-3. URL http://dblp.uni-trier.de/db/conf/ssiri/ssiri2010.html#WongDSKS10. (Cited on page 2.)

Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. **ACM Comput. Surv.**, 41(4):19:1–19:36, October 2009. ISSN 0360-0300. doi: 10.1145/1592434.1592436. URL http://doi.acm.org/10.1145/1592434.1592436. (Cited on pages 2 and 3.)

William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. **Communications of the ACM**, 17(6):337–345, 1974. URL http://dblp.uni-trier.de/db/journals/cacm/cacm17.html#WulfCCJLPP74. (Cited on page 329.)

Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A Wireless Sensor Network For Structural Monitoring. In **SENSYS**, pages 13–24. ACM, 2004. (Cited on page 161.)

Wang Yi, Moshe Vardi, Jürgen Giesl, et al. Liveness Manifestos. **Beyond Safety, International Workshop, Schloß Ringberg, Germany**, April 2004. URL http://www.cs.nyu.edu/acsys/beyond-safety/liveness.htm. (Cited on pages 155 and 240.)

Yuen-Tak Yu and Man Fai Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. **Journal of Systems and Software**, 79(5):577–590, 2006. URL http://dblp.uni-trier.de/db/journals/jss/jss79.html#YuL06. (Cited on pages 21 and 22.)

Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman, editors. **Model-based testing for embedded systems**. Computational analysis, synthesis, and design of dynamic systems. CRC Press, Boca Raton, 2011. (Cited on pages 15 and 241.)

Hao Zhang, Yonggang Wen, Haiyong Xie, and Nenghai Yu. **Distributed Hash Table - Theory, Platforms and Applications.** Springer Briefs in Computer Science. Springer, 2013. ISBN 978-1-4614-9008-1. URL http://dx.doi.org/10.1007/978-1-4614-9008-1. (Cited on page 53.)

Michael Zhivich and Robert K. Cunningham. The Real Cost of Software Errors. **IEEE Security & Privacy**, 7(2):87–90, 2009. URL http://dblp.uni-trier.de/db/journals/ieeesp/ieeesp7.html#ZhivichC09. (Cited on pages 1 and 2.)

Hong Zhu. A Formal Analysis of the Subsume Relation Between Software Test Adequacy Criteria. **IEEE Trans. Software Eng.**, 22(4):248–255, 1996. (Cited on page 21.)

Martina Zitterbart and Erik-Oliver Blaß. An Efficient Key Establishment Scheme for Secure Aggregating Sensor Networks. In **ACM Symposium on Information, Computer and Communications Security**, pages 303–310, Taipei, Taiwan, March 2006. URL http://doc.tm.uka.de/2006/blass2.pdf. ISBN 1-59593-272-0. (Cited on page 168.)

Index

Symbols

"
  – (PROMELA), **45**
  – (output prefix), **188**
(S, T), **32**
(S, T, S^0), **32**
(S, T, Σ, I), **34**
(S, T, Σ, I, S^0), **32**
$(S, \mathfrak{T}, L, S^0)$, **32**
$(S, \mathfrak{T}, L, \Sigma, I, S^0)$, **32**
$(\cdot, \ldots, \cdot)$ (open interval over $\omega + 1$), **10**
$(\ddot{o}_i)_{i \in [1,\ldots,p]}$, **315**
$(\mathcal{S})_{\geq k}$ (suffix of a linear Kripke structure), **67**
$(\mathcal{S})_{\leq k}$ (prefix of a linear Kripke structure), **67**
$(\mathbb{T}_i^{full})_{i \in [1,\ldots,1+r_{curr})}$ (full TC seq), **265**, 301
$(\mathbb{T}_i)_{i \in [1,\ldots,1+p_{curr})}$ (TC seq), **265**, 301
$(\mathbb{W}_i^{full})_{i \in [1,\ldots,1+r_{curr})}$ (full WTC seq), **301**
$(\mathbb{W}_i)_{i \in [1,\ldots,1+p_{curr})}$ (WTC seq), **301**
$(\nu x); \cdot$ (restriction), **42**
$(\pi_i^{full})_{i \in [1,\ldots,1+r_{curr})}$ (full path sequence), **265**
$(\pi_i)_{i \in [1,\ldots,1+p_{curr})}$ (path sequence), **265**
$(\sigma_i^{full})_{i \in [1,\ldots,1+r_{curr})}$ (full trace sequence), **265**
$(\sigma_i)_{i \in [1,\ldots,1+p_{curr})}$ (trace sequence), **265**
$(b_i)_{i \in [1,\ldots,1+p_{curr})}$ (bound sequence), **265**
$(l_i)_{i \geq k}$ (suffix, LTS), **36**
$(l_i)_{i \leq k}$(prefix, trace), **36**
$(r_i)_{i \in [1,\ldots,1+r_{curr})}$ (restart sequence), **265**
$(x_i)_i$, **10**
$(x_i)_{i \in I}$, **10**
$+(p2W, +(p \cdot z_0, (1-p) \cdot mean(\ldots)))$ (aggregation), **308**
$+(p2W, +(\ldots))$ (aggregation), **307**
$+(p2W, max(\ldots))$ (aggregation), **307**
$+(p2W, mean(\ldots))$ (aggregation), **307**
$+(p2W, min(\ldots))$ (aggregation), **307**
0, **9**
0 (inert process), **41**
1, **9**, 36
$?r$ (reified implicit reset capability), **262**
A-complete, **199**
A-enabled $\mathcal{S}$, **199**
A-inhibited, **199**
A-triggered, **199**
ABA problem, 50
$D_{\Sigma_{FOL}}$, **30**
$Eeven(p)$, **86**
F (function symbols), **29**
$FORM$, **29**
$FORM_{\Sigma_{FOL}}$, **29**
$FORM_{\Sigma_{FOL}}(Var)$, **29**
F_T^b, **106**
F_i^b, **106**
$F_{L,i}^b$, **106**
F_{S^0}, **106**
GO packet, **163**
G_b, **122**
I, **184**
$I(\cdot, \cdot)$ (interpretation function), **35**
$I_{\Sigma_{FOL}}$, **30**
L, **35**
L (for STSs), **40**
L_I, 16
L_I (input), **187**
L_U, 16
L_U (output), **187**
L_ϵ, 188
L_τ, 188
$L_{\delta\epsilon}$, 191
$L_{\epsilon\delta}$, 191
$L_{\tau\delta}$, 191
$L_{\tau_\not{a}\tau_u}$, 191
$L_{x_1 \ldots x_n}$, **187**
P, **138**, 275
P (number of processes), **49**
P (predicate symbols), **29**
$P_{\ddot{s}}$, **260**
$Prop()$, **69**
$Prop(\mathcal{L})$, **80**
$Prop_{lin}()$, **69**
$RSEM(x)$, **355**
S, **32**

S (for STSs), **40**
$SEM(x)$, **355**
SET packet, **163**
$SETAGG$ packet, **163**
S^0 (for STSs), **40**
S^0 (initial states), **32**
$S^{\mathcal{P}}$, **115**
S_P (speedup), **49**
S_{POR}, **103**
S_{det}, **196**
S_{udet}, **228**
T, **32**
T-solver, 55
$T(s)$, **34**
$TERM$, **29**
$TERM_{\Sigma_{FOL}}$, **29**
$TERM_{\Sigma_{FOL}}(Var)$, **29**
TL, **68**
U, **184**
$[[\mathcal{S}, L]]^b$, **106**
$[[\mathcal{S}_{\mathcal{V}^0}]]$, **41**
$[[\mathcal{S}_{\mathcal{V}^d_{LC}}]]$, 350
$\Box$ (operator), **73**
$\mapsto$, 10
$\Diamond$ (operator), **73**
$\mathcal{LTS}(L_I, L_U)$, 187
$\mathcal{LTS}(L_I, L_U, *)$, 187
$\mathcal{LTS}(L_I, L_U, +)$, **187**
$\mathcal{LTS}(L_I, L_U, \delta, \epsilon)$, 191
$\mathcal{LTS}(L_I, L_U, \epsilon)$, 188
$\mathcal{LTS}(L_I, L_U, \epsilon, \delta)$, 191
$\mathcal{LTS}(L_I, L_U, \tau, \delta)$, 191
$\Leftrightarrow$, **10**
$\mathbb{N}_{>0}$, **9**
$\mathbb{N}_{\geq 0}$, **9**
Φ function, **157**
Σ, **34**
Σ for propositional logic, **26**
Σ_{lazy}, **263**
Σ_{FOL}, **29**
Σ_{sv}, **91**
$\Sigma_{sv}(X)$, **106**
$\mathcal{S}_{\natural}$, **199**
$\mathcal{S}_{\natural}$, **199**
$Straces(\cdot)$, 194
$Straces_{\mathcal{S}_{\delta\tau^*}}(\cdot)$, 194
$Utraces(\cdot)$, 194, 228
$after(\cdot, \cdot)$, 194
$after_{\mathcal{S}_{\delta\tau^*}}(\cdot, \cdot)$, 194
α (arity), **29**
$\approx_\Sigma$ (equivalence relation), **66**
$\approx_{Straces}$, **194**
$\approx_{Straces}$ (on sets), **194**
$\approx_\Sigma$ (equivalence relation over Kripke structures), **66**
$\approx_\Sigma$ (equivalence relation over paths), **66**
$\approx_\Sigma$ (equivalence relation over states), **66**
β (variable assignment), **30**
$\boldsymbol{X_w}$, **76**
$\bot$, **72**
$\cdot$ (concatenation of LTS paths), **35**
$\cdot$ (concatenation of TS paths), **33**
$\cdot$ (concatenation of traces), **36**
$\cdot(\cdot)$ (input prefix), **42**
$\cdot + \cdot$ (choice operator), **42**
$\cdot;\cdot$ (sequential composition), **42**
$\cdot \mid \cdot$ (parallel composition), **42**
$\cdot \mid\mid \cdot$ (synchronous product), **42**
$\cdot \mid\mid\mid \cdot$ (asynchronous product), **42**
$\cdot \mid [c_1, \ldots, c_n] \mid \cdot$, **42**
$\cdot$ (parameter for a 10
χ (chaos), **199**
$|\cdot|$ (of an FSM), **38**
$|\mathcal{S}|$, **34**
δ, 16
$\delta(s)$, **191**
$\delta_u(s)$, **191**
$\delta_{\circlearrowleft}$, **191**, 192
$\delta_{\curvearrowright}$, **192**
$\delta_{\not\natural u}(s)$, **191**
$\delta_{\not\natural}(s)$, **191**
$\xrightarrow[d]{}$, **196**
$I^{lazy}(\cdot)$, **264**
$I^{lazy}_o(\cdot)$, **310**
$I^{lazy}_{\ddot{o}}(\cdot)$, **310**
$I^{lazy}(\cdot)$, **263**
S_{isol}, **222**, 331, 352
$\approx_{st}$ (stutter equivalence), **76**
$\mathbb{B}$, **9**
$\mathbb{B}$ in Dumont, 325
$\mathbb{S}_{Kripke,1}$ (one initial state), **35**
$\mathbb{S}_{Kripke,<\omega}$ (Kripke structures for finite trace semantics), **71**
$\mathbb{S}_{Kripke,\Sigma_{sv}}$, **91**
$\mathbb{S}_{Kripke,\geq\omega}$ (Kripke structures for infinite trace semantics), **71**
$\mathbb{S}_{Kripke,finite}$, **35**
$\mathbb{S}_{\omega-\text{automaton}}$, **76**
$\mathcal{A}(b_1, \ldots, b_n)$, **86**
$\mathcal{CT}$ (completeness threshold), **94**
$\mathcal{ET}$ (exhaustiveness threshold), **222**
$\mathcal{F}_L$, **35**
$\mathcal{F}_w$, **297**

$\mathcal{F}_{TC}$, **206**
$\mathcal{F}_{mod}$, **206**
$\mathcal{IOTS}_A(L_I, L_U)$, **199**
$\mathcal{IOTS}_A(L_I, L_U, x_1, \ldots, x_n)$, **199**
$\mathcal{IOTS}_A(L_I, L_U, x_1, \ldots, x_n, *)$, **199**
$\mathcal{IOTS}_A(L_I, L_U, x_1, \ldots, x_n, +)$, **199**
$\mathcal{LTS}(L_I, L_U, \delta, \epsilon, *)$, **191**
$\mathcal{LTS}(L_I, L_U, \epsilon, *)$, **188**
$\mathcal{LTS}(L_I, L_U, \epsilon, \delta, *)$, **191**
$\mathcal{LTS}(L_I, L_U, \tau)$, **188**
$\mathcal{LTS}(L_I, L_U, \tau, \delta, *)$, **191**
$\mathcal{LTS}(L_I, L_U, \tau_{\not\delta}, *)$, **189**
$\mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n)$, **187**
$\mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n, *)$, **187**
$\mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n, +)$, **187**
$\mathcal{P}$, **115**
$\mathcal{S} \in \mathcal{LTS}(L_I, L_U, \tau_{ut}, *)$, **189**
$\mathcal{S}_\delta$, **191**
$\mathcal{S}_\pi$, **67**
$\mathcal{S}_{\text{POR}}$, **103**
$\mathcal{S}_{LC}$, **349**, 385
$\mathcal{S}_{\delta\tau^*}$, **191**
$\mathcal{S}_{\tau^*\delta}$, **191**
$\mathcal{S}_{\tau^*}$, **188**
$\mathcal{S}_{det}$, **196**
$\mathcal{S}_{udet}$, **228**
$\mathcal{TTS}(L_I, L_U, \delta)$, **204**
$\mathcal{WTTS}(L_I, L_U, \delta)$, **297**
$\mathfrak{T}$, **35**
$\mathfrak{T}^{\mathcal{P}}$, **132**
$\mathfrak{T}_{\text{POR}}$, **103**
Models(), **69**
$Models_{lin}()$, **69**
NPC_ϕ, **129**
Rtraces, 234
$Straces_{\mathcal{S}_{\tau^*\delta}}(\cdot)$, **194**
$Utraces_{\mathcal{S}_{\tau^*\delta}}$, **228**
$Utraces_{\mathcal{S}_{\tau^*\delta}}(\cdot)$, **229**
Var (for STSs), **40**
$after_{\mathcal{S}_{\tau^*\delta}}(\cdot,\cdot)$, **194**
$branchout_{\mathcal{S}}$, **194**
$branch_{\mathcal{S}}$ (LTS), **36**
$branch_{\mathcal{S}}$ (TS), **34**
$branch_{\mathcal{S}}$ (of an FSM), **38**
$computationTree(\mathcal{S})$, **67**
$depth(\mathbb{T})$ (of a test case), **204**
$depth_{\mathcal{S}}$ (TS), **34**
$dest(s, \rightarrow)$ (in a TS), **34**
$dest(s, \rightarrow^*)$ (in a TS), **34**
$dest(s, \rightarrow^+)$ (in a TS), **34**
$dest_{\mathcal{S}}(\cdot,\cdot)$ (in a TS), **34**
$discharge(\cdot)$, **264**
$discharge_o(\cdot)$, **310**
$discharge_{\ddot{o}}(\cdot)$, **310**
$distance_o$, **312**
$faultable(\cdot)$ (for *Straces*), **203**
$faultable(\cdot)$ (for states), **202**
init, 188
$init_{\mathcal{S}_{det}}$, **196**
$init_{\mathcal{S}_{udet}}$, **228**
$init_{\mathcal{A}}$, **38**
$ioco_{\mathcal{F}}$, **227**
ioco, **202**
phaseVariant, **295**
$refines_{\mathcal{F}}$, **230**
refines, **230**
$subF_{lin}(\cdot)$, **83**
uioco (w/o using $\mathcal{S}_{udet}$, **229**
urefines, **231**
$verd_{\mathbb{M}}(\mathbb{T})$, **206**
$verd_{\mathbb{S}}(\mathbb{T})$, **207**
b_{exec}, **319**
b_{future}, **319**
i_δ, 336
i_δ caching, **336**
p_{curr} (current phase number), **265**
r_{curr} (current restart number), **265**
t^{TO}_{curr}, 244, 245, **267**
t_{curr}, **241**, 242
ϵ, **10**
$\equiv$ (for property descriptions in temporal logics), **69**
$\equiv$ (for temporal logics), **80**
$\exists$, 10
$\exists$, **70**
$\forall$, 10
$\forall$, 10
$\geq$ (for temporal logics), **80**
$\gtrsim$ (for temporal logics), **80**
i_δ, **204**
λ, 10
$\text{gen}_c(\cdot)$, **228**
$\ddot{s}$, 260
$\ddot{\mathbb{T}}$, **204**
$\ddot{\mathbb{W}}$, **297**
$\ddot{o}$, 254
$\ddot{o}$ (active test objectives), **260**
$\ddot{o}$ (as test objective), **310**
$\ddot{o}_{init}$, **310**
$\mathbb{S}_{sim\,\mathcal{S}}$, 210, **220**
$\mathbb{V}$, **20**
$\mathbb{T}||\mathbb{M}_{\delta\tau^*}$, 206
$\mathbb{T}$ (TC), **204**

$\mathbf{P}_{enabled(\ddot{s})}[\ddot{s} \xrightarrow{l} \ddot{s}']$, **320**
$\mathcal{D}$, **30**
$\mathcal{D}_{\Sigma_{FOL}}$, **30**
$\mathcal{F}$, **202**
$\mathcal{I}$, **40**
$\mathcal{V}$, **40**
$\mathcal{V}^d$ (default variable initialization), **41**
$\mathcal{L}$ bounded model checking, **93**
$\mathcal{L}$ model checking, **90**
Strace coverage, 217, 219
aggPath2Ws$(\cdot)$, **310**
aggWTCs, **298**
covFiningPath2W, **303**
cycleFiningPath2W, 323
cycleFinintPath2W, **317**
finingPath2W, **302**
length$_{discharge}$, 320
path2W(π), **298**
path2W$_o$ difference, 333
path2W$_o(\cdot)$, **310**
path2W$_{\ddot{o}}(\cdot)$, **310**
$|\cdot|$ (path length, LTS), **35**
$|\cdot|$ (trace length), **36**
$|\mathbb{T}|$ (size of a TC), **204**
$|\pi|$, **33**
$|\cdot|$ (path length, TS), **33**
$\models$, **30**, **66**
$\models$ for ω-automata, **77**
$\models$ for ECTL*, **86**
$\models$ for FOL, **30**
$\models$ for propositional logics, **26**
$\models$ for temporal logics, **69**
μ (least fix-point), **105**
ν (greatest fix-point), **105**
ω, **9**
ω-automaton, **76**
ω-regular language, 129
$\omega + 1$, **10**
$\dot{\cdot}\langle\cdot\rangle$ (output prefix), **42**
$\overline{x}$ (mean), **355**
$\xrightarrow{\alpha}_{\text{POR}}$, **103**
$\xrightarrow{\cdot}^*$, **36**
$\xrightarrow{\cdot}^+$, **36**
$\xrightarrow{\pi}^*$, **34**
$\xrightarrow{\pi}^+$, **34**
$\xrightarrow{\sigma}^*$ (in an FSM), **38**
$\xrightarrow{\sigma}^+$ (in an FSM), **38**
$\xrightarrow{l}$, **35**
π-calculus, 41, 43
$\pi \cap \mathcal{P} \neq \varnothing$, **115**
π^{full} (concatenated full path), **266**
$\pi_{(\mathcal{S},\{init\})}$, **67**
$\pi_{\mathcal{S}}$, **67**
$\pi_{\geq k}$ (suffix), **33**
$\pi_{\geq k}$ (suffix, LTS), **35**
$\pi_{\leq i}$(prefix, LTS), **35**
$\pi_{\leq k}$ (prefix), **33**
$\rightarrow$, **34**
$\rightarrow$ (for STSs), **40**
$\rightarrow$ (in an FSM), **38**
$\rightarrow^*$, **34**
$\rightarrow^*$ (in an FSM), **38**
$\rightarrow^+$, **34**
$\rightarrow^+$ (in an FSM), **38**
$\sigma(x)$ (standard deviation), **355**
σ^{full} (concatenated full trace), **266**
$\sigma_{\ddot{s}}$, 222
τ, **188**, 350
– internally like output, **188**
– may consume time, **188**
τ (silent prefix), **42**
τ abstraction, **188**
τ abstraction after reifying quiescence, **191**
τ+ reduction, **112**
τ-closed, **188**
τ-closure, **188**
τ-cycle, **189**
τ-cycle elimination, 111
$\tau_{\not t}$, **188**
τ_u, **188**
$\tau_{\not\nu}$, **188**
$\tau_{\not\nu\not t}$, **188**
$\tau_{\not u\not t}$, **188**
$\tau_{\not u}$, **188**
τ_t, **188**
$\tau_{u\not\nu}$, **188**
τ_{ut}, **188**
τ_u, **188**
$\xrightarrow[\tau^*]{}$, **188**
$\xrightarrow[\delta\tau^*]{}$, **191**
$\xrightarrow[\tau^*\delta]{}$, **191**
$\xrightarrow[ud]{}$, **228**
* (Kleene star), **10**
$^+$ (Kleene plus), **10**
$^\omega$ (infinite sequences), **10**
active, **264**
active$_o$, **310**
active$_{\ddot{o}}$, **310**
b (bound for BMC), **93**
b (bound for genTS), **215**
$b(\cdot)$, **207**

$b(\mathbb{T})$, **207**
$b_I(\cdot)$, **207**
$b_U(\cdot)$, **207**
$b_+(\cdot)$, **285**
$b_-(\cdot)$, **285**
$b_{p_{curr}}$, 285
b_{max}, **258**, 285
b_{min}, **258**, 285
c (canonical embedding), **38**
$c_{\ddot{s}}$, **336**
$even(p)$, **80**
$exists$, 10
$f_{distr}(\cdot)$ (coverage), **302**
$f_{distr}(\cdot)$ (for nondeterminism on output), **303**
f_{agg}, **162**
$f_{new}(\cdot)$ (coverage), **302**
$f_{new}(\cdot)$ (for nondeterminism on output), **303**
$free(\cdot)$, **29**
$i$uptoBound, **361**
$in(\cdot)$, 194
$in^{\cap}_{\mathcal{S}_{\tau^*}}(\cdot)$, **228**
$in^{\cap}(\cdot)$, 228
$in_{\mathcal{S}_{\tau^*}}(\cdot)$, **194**
$l(s)$, **36**
$max(p2W, max(\dots))$ (aggregation), **307**
$max(p2W, min(\dots))$ (aggregation), **307**
$max(x)$, **355**
$median(x)$, **355**
$min(x)$, **355**
n (sample size), **355**
n-choices state coverage, **290**, **303**
o (abstract TO def.), **260**
o (test objective), **264**, 310
$out(\cdot)$, 194
$out_{\mathcal{S}_{\delta\tau^*}}(\cdot)$, 194
$out_{\mathcal{S}_{\tau^*\delta}}(\cdot)$, **194**
p_+, **285**
p_-, **285**
$paths_{\text{fail}}(\mathcal{S}_{det})$, **213**
$paths_{\mathbb{V}}(\mathcal{S}_{det})^{r_{curr}}$, **265**
$paths_{\mathbb{V}}(\mathcal{S}_{det})$, **213**
$supp(I(\cdot, s))$, 35
$supp(I(p, \cdot))$, 35
$supp(f(\cdot))$, 9
$traces_{\mathcal{S}_{\tau^*}}(\cdot)$, **194**
$type(\cdot)$, **40**
val_I, **26**
$val_{\mathcal{D},\beta}$, **30**
w (weight function), **297**
$w_{\mathcal{WTTS}}$, **297**
$[\cdot, \dots, \cdot]$ (closed interval over $\omega + 1$), **10**
(A-)fairness, 133
(miss, fail), **20**
(miss,pass), **20**
(super-)linear speedup, **365**
$=$, **45**
? (input prefix), **188**
Rtraces, 234
Überdeckungen, viii
$Utraces_{\mathcal{S}_{\tau^*\delta}}$, 228
$Utraces_{\mathcal{S}_{\tau^*\delta}}(\cdot)$, 229
Var, **29**
CTL* semantics, 71
$S_{\rightarrow^*}$(state space), **34**
$T_{\rightarrow^*}$, **34**
$\mathcal{S}_{\rightarrow^*}$, **34**
$\mathfrak{T}_{\rightarrow^*}$, **35**
$\ddot{o}_{S_{isol}}$, **352**
$\ddot{o}_{cov}$, **351**
maxprt, 353
MOD, **185**
MOD, 201
Rtraces, 217
Rtraces, 203
STATUS packet, **162**
SUT, **16**
SUT, **183**
TEST, **185**
Var, **29**
refines, 9
sioco, 234
subF, **81**
uioco, 217, **228**
$\overline{t^{max}_{curr}}$, **356**
$\mathrm{P}_{coverViaTOs}$, **293**
$\mathrm{P}_{coverage}$, **292**
$\mathrm{P}_{coverage}$ via weights, **302**
$\mathrm{P}_{discharge}$, **293**
P_{fair}, **295**, 297
P_{goal}, **295**
P_{goal}, 296
P_{phase}, **295**
P_{phase}, 296
$\mathrm{P}_{vanishing}$, **293**
$\mathrm{P}_{\rightarrow OTF}$, **291**
$\mathrm{fairness}_{model}$, **216**
$\mathrm{fairness}_{model}$ (abstr. def.), **186**
$\mathrm{fairness}_{spec}$, **219**
$\mathrm{fairness}_{spec}$ (abstr. def.), **186**
$\mathrm{fairness}_{test}$, **218**
$\mathrm{fairness}_{test}$ (abstr. def.), **186**
$\mathrm{leader}_{\mathrm{Itai}}$, **116**
leader_t, **116**
o_{NFRSML}, **352**

o_{RSML}, **351**
o_{dec}, **352**
$pc\_$, **46**
t_{curr}^{max}, **353**
t_{curr}^{max}, **348**
t_{curr}, **265**
n/a, 149
−> (PROMELA), **45**
? (PROMELA), **45**
; (PROMELA), **45**
CollapsedLocationsSetPath2W, **327**
DecisionStrategy, **327**
FinalLocationSetDefaultPath2W, **327**
INV, **39**
LazyOTF, vii, 6, **253**
LazyOTF debug console, 328, **332**
LazyOTF introspection window, **331**
LazyOTF is active, **260**
LazyOTF manual, **325**
LazyOTF with weight heuristics, **301**
LazyOTF$_i$, **275**
LazyOTF's XML configuration files, 329
LengthPath2W, **327**
Location.LocationTestType.DecisionStrategy, **327**
LocationValuation2Weight, **327**
Locations2Weight, **327**
MappingStrategy, **327**
MessageSetPath2W, **327**
NodeValue2Weight<N>, **327**
NodeValueSummingPath2W, **327**
NondetFiningFinalLocationSetPath2W, **328**
NondeterminismFiningNodeValue2Weight, **327**
POST, **39**
PRE, **39**
Path2Weight<N, E>, **327**
SatStrategy, **327**, 352
SumOfOtherNodeValue2Weight<N>, **327**
SuperLocationSizeNodeValue2Weight, **327**
TestTypeDefaultNode2Weight, **327**
backtrack, 334
cacheMap, **336**
dynamicInfo, **254**, 254, 267
exactBound, **361**
exitCriterion (dynamicInfo), **253**
explore, 334
fail , **204**
fail (abstr. def.), **20**
gen, **209**
gen$_{exec}(\mathcal{S}, \mathbb{S})$, **209**
generateLicense, **349**, 385
havoc command, **157**
hit, **20**
inconclusive, **20**
innerDFS(), **99**
jump, 334
localBackup, **349**, 385
miss, **20**
nondet command, **157**
np_, 115
pass, **204**, 326
pass (abstr. def.), **20**
progress label, 115, 133
promise broken, 50
promise kept, 50
promise pending, 50
pullExternalDischarges(), **276**
pushInternalDischarge($\ddot{o}$ o), **276**
receive, 334
remoteBackup, **349**, 385
removeAll, **349**, 385
removeExpired, **349**, 385
removeLicense, **349**, 385
showLicenses, **349**, 385
stack, **99**
take, 334
testExecutionSubphase($\mathbb{T}_p$, $\mathbb{S}$, dynamicInfo), **254**
traversalSubphase($\mathcal{S}$, $\ddot{o}$, dynamicInfo), **254**
fifo , 125
hash table, 99
P$_{exh \Rightarrow disch}$, **294**
~~30~~', 149
OWCTY, **145**
OWCTY, **112**
$\ddot{o}$ (as test objective), 310
1-choice state coverage, **290**
10-minute build, 343
2-choices state coverage, **290**

A

A operator, **73**
a priori, 243
A□, 74
A◊, 74
ABA problem, 50
abrasion, 184
absolute speedup, **49**, 147, 366
abstract assembler language, 157
abstract behavior model, **164**, 165
abstract behavior model, 164
abstract from the full system, 60
abstract interpretation, **102**, 107
abstract method, 122

abstract state, **40**
abstract state coverage, 330
Abstract State Machine Language, 251
abstract state machines, 251
abstract syntax tree, 26
abstract test case, **207**, 249
abstract testing, **173**
abstract use case, **252**
abstraction, **59**, 59, 96, 158, 207
  – τ, **188**
abstractions, 33
abstractions by the test adapter, **207**
Acamar, 354
accept, **46**
accept label, 46
accept of Büchi paths, **77**
accept of extended Büchi paths, **77**
acceptance
  – label, 46
  – state, **46**
acceptance condition, **77**
acceptance cycle, **98**
acceptance cycle detection, **99**, 109
acceptance testing, **18**
acceptance tests, 328
accepting state, 37
accumulated WC time, 366, 370, 372, 375, 378, **392**
achieved TG, **260**
acquire memory barrier, **51**
action, **32**
action refinement, **207**, 207
ActionScript, 172
activated, **19**
active, **260**
active test goal, **261**
active testing, **14**
ACTL*, **91**
m , **91**
Actor Model, 48
actually missing information, **59**
AD, 341, **342**
Ada, 172
adapter, **249**
adequacy metric, **23**, 24, 223
$\mathcal{A}_{\Diamond\Box \mathrm{np\_}}$, **118**
adversary, 307
  – good, 307
  – mean, 307
adverse acting entity, 162
A-enabled
  – $\mathcal{S}$, **199**
A-enabled
  – state, **199**
$\mathcal{A}_{even}$, **80**
AF, 74
after$(\cdot,\cdot)$, 194
after-progress state, **138**
$after_{\mathcal{S}_{\delta\tau^*}}(\cdot,\cdot)$, 194
$after_{\mathcal{S}_{\tau^*\delta}}(\cdot,\cdot)$, 194
AG, 74
aggregated to weights of TCs, 298
aggregation function, **162**
aggregation path, **161**
aggregation tree, **161**
aggWTCs, 298
agile, 260
AGILE, **348**
Agile Applications, **345**
Agile Conference, **341**
agile modeling, **348**
agile quality management, 345
agile software development, 9, **342**
Agile Unified Process, 342
A-inhibited, 199
Aldebaran, 248
algorithm, 275
algorithm back-ends for property checks, 111
alias analysis, 175
all full path sequences of $\mathcal{S}$, **265**
all WC time, **353**, 371
all-choices state coverage, **290**, **303**
all-choices transition coverage, 303
all-def-use-paths, **22**
all-defs, **21**
all-progress cycle, **135**
all-uses, **21**
α (arity), **29**
alphabet, **37**
alternating simulation, 234, 251
alternating TO period, **315**
always operator, **73**
Amdahl's law, 49
ample(), **103**
ample decomposition proviso, **103**
ample set, **103**, 103, 135
analysis, 277
angelic completion, 61, **199**
anomaly, **20**
anonymous ring, 116
Anquiro, **176**
AP, 278
AR, 74
architecture, 331

arithmetic error, 158
arithmetic operators in Dumont, 325
arity function, **29**
ARM, 172
array, 177
array (PROMELA), **44**
array bounds, 3
array index out of bound, 158, 169, 173
array index out of bound error, 171
ArraysEx theory, 31
AsmL, 251
assembleTC, **211**
assert () (PROMELA), **109**
assert (·) statement for CBMC, **158**
assertion, **109**
assertion (abstr. def.), **14**
assertion violation, **109**
associative, 26
assume-guarantee, 96, 167
assume(f) statement for CMBC, **158**
asynchronous
  – execution, 47
  – transmission, **41**
asynchronous communication, 195
asynchronous parallel composition, **42**
asynchronous product, **42**, 91
at least as expressive as, **80**
atomic, 207
atomic, **47**
atomic behavior, 32
atomic condition coverage, 252
atomic linear input-inputs refinement, 208
atomic linear refinement, 208
atomic propositional variable, 34
atomic refinement, 208
AtomicNumber, 53
A-triggered, 199
AU, 74
Audition, 252
AUFLIA, 31
Aut, 248
authenticity, 161
automata theory, 32
automated test case generation, **15**
automated testing, **15**
automatic, 250
automatic nightly tests, 266
automatic traceability, **17**, 256
autonomous, 161
autonomous sensor, 155
autostart function, **164**
auxiliary TO, 261
Availability, 278
AW, 74
awards in PRISM, 113
AX, 74
axiom, **30**

B

b (bound (for genTS)), 215
b (bound), 93
$\mathbb{B}$ (Boolean values), 9
$b(\cdot)$ (binding), 207
$b_+(\cdot)$ (depth bound increment function), **285**
$b_-(\cdot)$ (depth bound decrement function), **285**
Büchi, **77**, 109
Büchi automaton, **77**, 118
Büchi automaton over formulas, **86**
back-link, 126
backtracking, 6, 99
backtracking by eager scheduling, **59**
backward recovery, 204
backward traceability, 17
backward traversal, 97
bad weather behavior, **19**
balanced aggregation, **307**
baseline, 372
basic command, **45**
basic depth-first search, **99**
basic statement, 35, **45**
basic test objective, **310**
batch mode, **249**
BCG, **39**, 110
BCG graphs, 247, 248
BDD, **28**, 92, 111
BDUF, **344**
behavior, 32
behavioral MBT, 184
behavioral property, **66**
behavioral property of a TG, **260**
benchmark, 373
Beschränkungs-Heuristik, viii
best-first heuristic, 174
β (variable assignment), 30
b_{exec}, **319**
BFS, **125**
b_{future}, **319**
$(b_i)_i$ (bound sequence), 265
$b_I(\cdot)$ (binding), 207
big design up front, 55, **344**
Binary Coded Graph, **39**
binary decision diagram
  – reduced ordered, **28**
  – shared reduced ordered, **28**

binary modality, 70
binding, **207**
bit (PROMELA), **44**
bit-precise, 174
bit-precise memory model, 172
bitblasting, 55
bitfield, 172
bitstate hashing, **108**
black-box, **14**, 183
black-box testing, **14**
block, 45
blocked
– label, **35**
– process, **89**
– sequence, 45
blocking never claim, 47
blueprint, 221, 229
b_{max} (maximal bound), 258
b_{max} (maximal depth bound), **285**
BMC, 3, **93**, 122
BMC encoding, 106
BMC tool with a bound check, 223
b_{min} (minimal bound), 258
b_{min} (minimal depth bound), **285**
bold, **10**
BoogiePL, **173**
bool (PROMELA), **44**
Boolean, **9**
Boolean decision procedure, 27
Boolean satisfiability problem, **27**
Boolector, 158, 172
Boolector SMT solver, 55
bottleneck (in a state space), 374
bound, 285
bound (for genTS), **215**
bound (for BMC), **93**
bound (for $\text{DFS}_{\text{incremental}}$), **122**
bound by a quantifier, **29**
bound check (for genTS), **223**
bound check (for BMC), **94**
bound heuristics, 6
bound heuristics, 351
bound sequence, **265**
bound variable, **29**
boundary value analysis, 252
bounded DFS, 122, 174
bounded integers, 44
bounded model checking, 3, **93**, 106, 215
$\Box$ operator, **73**
box operator, **73**
BPEL, 252
branch coverage, 252, 275, 328
branching bisimulation, 111
branching time formula, **70**
branching time logics, **70**
branching time operators, **74**
branching time property, **66**
$branchout_{\mathcal{S}}$, 194
$branch_{\mathcal{S}}$ (LTS), 36
$branch_{\mathcal{S}}$ (TS), 34
breadth-first search, 125
break, **46**
broadcast, **52**
$b_U(\cdot)$ (binding), 207
buckets, 108
buckets of hash tables, 111
buffered channel, **41**
bug, 1, **19**
bug finding, 3, **93**, 175, 176, 215, 291
built-in standard checks, **158**
business requirements, **17**
busy-waiting, **88**
byte (PROMELA), **44**

C

c, **21**
C, 107, 172
C Bounded Model Checker, **158**
C flag, **331**
C language, 23, 44, 158
C programming language, 45, 112, 141, 156, 158
C++, 112, 156, 172
C++11, 112
C0, **105**
C1, **105**
C2, **105**
$\text{C2}^{\mathfrak{T}}$, **136**
$\text{C2}^{\mathfrak{T}}_{\text{lazy}}$, **138**
C2^{S}, **136**
$\text{C2}^{S}_{\text{lazy}}$, **138**
$\text{C2}^{transparent}$, **104**
C2^{S}, **136**
$\text{C2}^{S}_{\text{lazy}}$, **138**
$\text{C2}^{\mathfrak{T}}$, **136**
$\text{C2}^{\mathfrak{T}}_{\text{lazy}}$, **138**
C3, **105**
C3', **105**
C#, 172, 251
cache coherency protocol, **51**
cache line, **51**
cache oriented, 111
cacheMap, 336

caching, **51**
caching local transitions, 111
CADP, 39, 110, **247**, 248
CAESAR, 247, 248
calendar dates in Dumont, 325
call stack, 94
call-by-value, 171
canonical embedding, **38**
canonical heap layout, **112**
CAP theorem, 278
capacity of a channel, **41**
capacity of a channel (PROMELA), 45
cartesian POR, 103
CAS, **50**, 142
cause, 20
CBMC, 155, **158**
CENELEC EN 50128, 4
centralized computing, 273
centralized network, **52**
centralized test controller, **273**
certification, **4**, 237
CESMI, 112
CGλ, **252**
channel, **41**
channel (PROMELA), **44**
chaos, 229
chaos state, 199
chaos state (χ), **199**
check tasks, **55**
Checker Framework, 4, 328
choice, **42**
CI, **343**
classical guidance heuristics, **286**
classical test selection heuristics, 286
classical testing, 2
client machine, **273**
client-server protocol, 116
client/server network, **52**, 273
clock speed, 47, 388
closed (logic), **69**
closed formula, **29**
closed system, **16**, 157
closed under, **69**
closed under stuttering, **76**
closure, **70**
closure, reflexive, transitive, 34
closure, transitive, 34
cloud computing, 53, 379
Cloud9, 275
cluster, 53, 354
CMC, **91**
CMPXCHG, 50
code contract, **39**
code coverage, 174
code coverage criterion, **20**
code instrumentation, 112, **164**
code safety properties, 169
CodeMeter License Central, **349**, 385
coffee machine, 184
coherency granularity, **51**
CoIn, 112
collapse compression, **108**, 145
collection
  – list, 53
  – map, 53
  – set, 53
collision, 108
combinator, **250**
combinatorial explosion, 91, 155, 163
command
  – basic, 45
  – compound, 45
command (PROMELA), 44
Common Explicit-State Model Interface, 112
Common Lisp, 172
Common Object Request Broker Architecture, 116
common trace, 346
communication, 48, 142
communication cost, 49
communication overhead, 52
communication performance failure, 277
commutativity, **104**
compare-and-swap, 112
Compare-And-Swap, **50**
compassion, **95**
compiler, 60, 172
compiler backend, 172
compiler frontend, 172, 177
compiler optimization, 159
complement Büchi automaton, **79**
complete, 3, **25**
  – model checker, **90**
  – model-based testing with execution, **239**
  – model-based testing without execution, **239**
  – SAT solver, **27**
  – test generation, **210**
  – test suite, **210**
complete abstraction, **163**
complete trace, **36**
completeness of $\mathrm{DFS}_{\mathrm{FIFO}}$, 127
completeness of $\mathrm{DFS}_{\mathrm{FIFO}}$ with POR, 136

completeness threshold, 3, **94**, 288
complex type, 44
complexity, 1, 59
complexity of $\mathcal{S}$, **34**
component, 1, 163, 176
component automata, 43, 103
component interaction automata, 112
component-based architecture, 249
composed test objective, **310**
composed type, 44
composite transition (PROMELA), 133
composition, 7, 96, 177, 260
– asynchronous parallel, 42
– full parallel, 42
– restricted parallel, 42
– synchronous parallel, 42
compositional testing, 189
compositionality, 302
compositionally, 115
compound command, **45**
compound statement, **45**
computable, 25, 239
computably enumerable, 239
computation tree, **67**, 201
Computation Tree Logic, **74**
Computation Tree Logic$*$, **70**
concasts, **161**
concatenated full path, **266**
concatenated full trace, **266**
concatenated path, **33**
concatenated transitions, 34
concatenation (PROMELA), 45
concatenation of LTS paths, **35**
concatenation of paths, **33**
concatenation of TCs, **204**
concatenation of traces, **36**
concatenation,FSM, **38**
concatenation,trace in an FSM, **38**
concolic testing, **174**
concolic unit testing engine, **174**
concrete test case, **207**, 249
concurrency, 252
concurrency optimizations, 111
concurrency utility, 53
concurrent object, 48
concurrent testing, 272
concurrently, 138, 276
condition, 21
condition for CMC, **91**
conditional model checking, **91**, 93, 154
conference protocol, 374
confidence, 13, 20
configuration, 325
conflicting clause, 55
confluence reduction, **103**
conform, **183**
conformance index, 310
conformance testing, **183**
Conformiq, 274
Conformiq Designer, 44, 245, **251**
Conformiq Grid, 274
Conformiq Modeling Language, **252**
congestion, 52, 277
conjunctive Büchi automaton, **78**
conjunctive normal form, 27
connected graph, 28, 203
Consistency, 278
consistency model, 51
constraining execution, 47
constraint satisfaction problem, **27**
constraint solver, **27**, 249, 326
constraint solving, 241, 252
Construction and Analysis of Distributed Processes, **247**
container data type, 349
contention, **50**, 51, 138, 142, 273, 388
contention for the network resource, 52, 277
context switch, **169**
Contiki OS, 176
continuation-passing style, 157
continuous, 33
continuous *refines* checks, **345**
continuous integration, **343**
continuous systems, 186
continuous time Markov chain, **113**
contract, **39**, 142
contract-based testing, 348
contraction, **60**
control
– by the SUT, **184**
– by the tester, **184**
control flow, 157, 174
control flow analysis, 157
control flow construct (PROMELA), **45**
control flow coverage criteria, **21**
control flow graph, 20, 21
control parallelism, **50**
controllable nondeterminism, **194**, 350
conventional testing, **2**
convergent, **188**
CORBA, 116, 274
Cord, 251
core theory, 31
corner cases, 239

correct, 13
corrected sample standard deviation, **355**
correctness, 1, **2**
correctness, 276
correctness by lazy scheduling, 58
costs in PRISM, 113
countable set, **9**
counter, 115
counterexample, **68**, 113
– shortest, 131
counterexample for a bound, **94**
counterexample path, **206**
counterexample test case, **206**
counterexample trace, **206**
counterexamples, 277
countermeasures to a TO period, **315**
covariant types, 328
cover, **20**
cover via TOs proviso, **293**
coverage, 7, 327
– *Straces* of $\mathcal{S}$, 216
– code, 20
– requirements, 22
– specifications, 22
– states of $\mathbb{M}_{det}$, 216
– states of $\mathcal{S}_{det}$, 216
– transitions of $\mathcal{S}_{det}$, 216
coverage criteria, **216**, 216, 283
– nondeterministic, **216**
– on test cases, **216**
coverage criteria for uncontrollable nondeterminism, **290**
coverage criterion, 6, **20**, 243, 287
coverage distribution, **302**
coverage distribution for nondeterminism on output, **303**
coverage efficiency, **302**
coverage efficiency for nondeterminism on output, **303**
coverage level, **20**, 304
coverage of *faultable*(S) in isolation, **221**, 331
coverage proviso, **292**
coverage task, **20**, 352
CPU, 47
CPU bound, **49**
CPU time, 353, 371
CPU-bound, 389
crash failures, **142**, 278
CRASH scale, 232
cross product, **42**
crossover, 288
CruiseControl, 343
CSP, **27**, 43
$\mathcal{CT}$ (completeness threshold), 94
CTL, **74**
CTL operators, **74**
m , **70**
CTMC, **113**
current number of phases, **265**
current number of restarts, **265**
customer feedback, 343
customer requirement, **343**
CUTE tool, **174**
cyber-physical system, **13**
cycle, **33**
cycle detection, 112, 144
cycle implementation proviso, **104**
cycle proviso, 6, **104**
cycle warning, **316**, 368
cycle warning threshold, **316**, 333
cycle, LTS, **35**
cycle,FSM, **38**
cycleClosing proviso, **105**
cycling heuristic, 249, **288**, 317

D

d, **21**
D, 172
D, **30**
$\mathcal{D}_{\Sigma_{FOL}}$, **30**
DAG (directed, acyclic graph), 28, 203
DART tool, **174**
data, 40
data centers, 53
data flow coverage, 334
data flow coverage criteria, **21**
data grid, 53
data type abstraction, 102
data-dependent control flow, 40
database, 53
datagram, 52
DbC, **39**
dd anomaly, **21**
deadlock, **89**
deadlock detection, **89**, 103
Deadlock detection, 109
deadlocks, 142
debugging, 167, 328, 343
decentralization, 273, 276
decentralized, 273
decentralized network, **52**
decision, **21**
decision strategy, **327**

decompose, 96
decomposition, 18, 97
deconstruct, 46
deductive verification, 2, 241
deep state, **350**
deep TO, 374
def-clear path, 21
def-use pairs, **21**
default decision strategy, **327**
default variable initialization, **41**
defect, **19**
definitions of done, **343**
Demand Based Switching, 388
demonic completion, 61, **199**, 229
dependency proviso, **105**
dependency relation, **104**
dependent, **104**
dependent metric, 49
deploy, 163
deployment, 281
depth bound, **285**
– decrement function, 285
– increment function, 285
– maximal, 285
– minimal, 285
– threshold, 285
depth bound decrement function, **285**
depth bound increment function, **285**
depth bound threshold, **285**
depth bounds, 254
depth of a test case, 204
depth-first search, **99**
– incremental, **122**
– nested, **99**
$depth_{\mathcal{S}}$ (TS), 34
$depth(\mathbb{T})$ (of a test case), 204
derivation engine, **250**
descriptive statistics, 372
design diversity, 43
design-by-contract, 4, **39**, 173
desired functional behavior, 59
desired state, 295, 305, 306
$dest(s, \rightarrow)$, **34**
$dest(s, \rightarrow^*)$, **34**
$dest(s, \rightarrow^+)$, **34**
$dest(s, \rightarrow)$ (in an FSM), **38**
destination, **34**
destination of a finite path
– FSM, **38**
– TS, **33**
$dest(\cdot)$ (destination of a path
– LTS), **35**
– TS), **33**
$dest_{\mathcal{S}}(\cdot, \cdot)$, **34**
determinism, 319
deterministic, 36
deterministic LTS, **36**
deterministic test case generation algorithm, 6
deterministic transition system, **34**
determinization, **196**, 325
determinized suspension automaton, **196**
development phase, 1, 18
deviation, 19
DFS, **99**
– incremental, **122**
– nested, **99**
$DFS_{prune,NPC}(S_{\rightarrow^*}\ s$, int workerNumber), 138
DFS_{FIFO}, **125**
DFS_{FIFO}, vii, 5
$DFS_{incremental}$, **122**
DHCP, **116**
$\Diamond$ operator, **73**
diamond operator, **73**
dilemma between software complexity and quality, **1**
DIR, 110
directed automated random testing, **174**
directed test generation, **23**
directed, acyclic graph, 28, 203
directly reachable, **34**
discharge function, **260**
discharge function for o, **264**, 310
discharge proviso, **293**
dischargeable, 260
discharged, 254, **260**
discharging, 7, **293**
disclosure of the system, 14
discrete system, **186**
discrete-time Markov chain, **113**
disjunctive Büchi automaton, **79**
disk I/O contention, 354
distance, 312
$distance_o$, 312
distorted weights, 314
distributed LazyOTF, vii, 6, **275**, 280
distributed lazyotf, 351
distributed architecture, 52
distributed assertions, 176
distributed component, **48**
distributed computing, **48**, 272
distributed executor service, **53**, 278, 281, 363
distributed hash table, 53

distributed model-based testing, **274**
distributed node, **48**
distributed shared memory, **48**
distributed shared memory system, **48**
distributed system, 155
distributed systems, 252
distributed tree, **273**
divergent trace, **131**
DiVinE, 44, 101, 110, **112**, 145
DiVinE modeling language, **112**
division by zero, 173
do, **46**
document late, **55**
DoD, **343**
does not suppress TOs, **306**
Dolev-Yao threat model, 176
domain-specific language, 233
dominance algorithm, 157
dormant, **19**
DPLL, 27
driver, **250**
DSE, **173**, 176
$D_{\Sigma_{FOL}}$, **30**
DSL, 233
DSM, **48**
d_step, **47**
DTMC, **113**
Dumont language, **325**
DVE, 110, **112**
dynamic, **2**
dynamic bound, **265**
dynamic bound heuristics, **285**
dynamic coverage, **24**
dynamic coverage level, **24**
Dynamic Host Configuration Protocol, **116**
dynamic information, 6, 243, **254**, 264, 284
dynamicInfo, **254**
dynamic load-balancing, **57**, 112
dynamic polymorphism, 252
dynamic reordering, 28
dynamic scheduling, 57, 61
dynamic symbolic execution, **173**, 335
dynamic testing, 13, **14**, 155
dynamic verification, **25**
dynamicInfo, 267

E

E, **70**
m), **86**
E□, 74
E◊, 74
E¬ F, **74**
E¬ G, **74**
E¬ X, **74**
ECTL*
– semantics, 86
eager, **54**, 270
eager encoding, **55**
eager micro-traversal sub-phase, 318, **319**, 336, 339
eager micro-traversal sub-phase optimization, 318, 353
eager micro-traversal sub-phases, 339
eager SMT, **55**
eager SMT solving, **31**
earliest counterexample, 131
earliest NPC w.r.t. progress, 127, 136
early backtracking, **144**
early bug detection, 2
early termination, **56**
EclEmma, 328
Eclipse Modeling Framework, 252
economics, 1
ECTL∗, **85**
ECTL*, **85**
even(*p*), **86**
EF, **74**
efficiency
– parallel, **49**
EFSM, **246**
EG, **74**
elastic, 53
else (PROMELA), **45**
embedded, 53
embedded device, 161, 163
embedded software, 2
embedded system, 1, **13**, 155, 157, 161
EMC, 173
EMF, 252
empircal, 348
emptiness check, **98**
emptiness proviso, **105**
empty string, **10**
enabled, **35**
– state, **199**
enabled(*s*), **35**
$enabled_{\mathcal{S}}(s)$, **35**
enabledness, **104**
encapsulated, 250
encoding for LTL BMC, 106
end, **46**
end label, 46
end state, **71**
end state validity, **109**

ENDFS, 111
energy, 163
energy consumption, 18, 162
Enhanced Intel SpeedStep, 388
enumeration (PROMELA), **44**
enums in Dumont, 325
environment, 1, **60**, 61
environment abstractions, 61
environmental data, **161**
environmental influences, 184
epistemological frame problem, 184
epistemological restrictions, 194
ϵ, **10**
equality, 189
equality in Dumont, 325
equally expressive, **80**
equidistribution, **307**
equivalence check, 179
equivalence of property descriptions in temporal logics, **69**
equivalent mutant, **24**, 179
ER, 74
Erlang, 204
Erreichen gewünschter Zielzustände, viii
error, 19
error path, **90**, 117
error screen, 184
ESAWN, **161**
ESAWN packet, **162**
$\mathcal{ET}$ (exhaustiveness threshold), 222
ETF, **40**
EU, 74
Euler diagram, 68, 70
evaluation function, **69**
evaluation function for FOL, **30**
evaluation function for PROP, **26**
even out, 303, 308
$even(p)$, **80**
event driven, 163
event-driven, 157, 176
eventually invariant, **129**
eventually operator, **73**
EW, 74
EX, **74**
ex., **10**
exception handling, **46**, 328, 349, 385
exceptional, 195
exceptions, 184
exclusive privilege, 47
EXE tool, **173**
executable, **44, 45**
executable sequence, **45**
executable specification languages, 274
executable test case, **207**
executable test steps, **241**
execution sub-phase, **253**
execution-based model-checking, 173, 275
exh⇒disch proviso, **294**
exhaustive, 183, **209**
– model-based testing with execution, **239**
– model-based testing without execution, **239**
exhaustive model checking, **93**
exhaustive test generation, **210**
exhaustive test generation gen_{exec}, **210**
exhaustive test suite, **210**
exhaustive test suite for $\mathbb{S}$, **210**
exhaustiveness, 7
exhaustiveness for an ioco variant, 228
exhaustiveness of genTC, 213
exhaustiveness of genTS, 216
exhaustiveness threshold, 6, **222**, 288
exhaustiveness thresholds, vi
$\text{P}_{exh \Rightarrow disch}$, **294**
m), **70**
m), **86**
exists, 10
$\exists$ (as path quantifier for ECTL$*$), **86**
$\exists$ (as path quantifier), **70**
exit criteria, 226
exit criterion, **23**, 24, 223, 245
exit criterion for a sprint, 343
exit criterion for an agile task, 343
exitCriterion , **253**
expansion, 333
expansion of an STS, **41**
expected, 184
expected value, **355**
experiment, 203, **391**
experiment archive, 328, 354
experiment sample, **391**
experiment sample set, **391**
experimental attrition, 370
experimental mortality, 370
Experimente, viii
experiments, 7
explicit enumeration, **92**
explicit nondeterminism, **157**, 172
explicit path MC, 174
explicit path model checking, **174**
explicit reset, **186**
explicit state MC, 241, 251, 252
explicit state model checking, **92**

explicit symbolic path model checking, **174**
exploration, 325
exploratory testing, 2
explored, 140
explorer, **250**
exponential decrease of POR, **134**
expressible, **80**
extended Büchi automaton, **77**
Extended Computation Tree Logic∗, **85**
extended FSM, **246**
Extended Secure Aggregation for Wireless sensor Networks, **161**
Extended Table Format, **40**
Extended-Büchi, **77**
extends (test cases), **214**
external discharge, **276**, 281
external nondeterminism, **194**
Extreme Programming, 342

F

F (function symbols), **29**
F flag, **331**
F operator, **73**
F-Soft, 177
f.a., **10**
f_{agg} (aggregation function), **162**
fail
– of a test case, **206**
– of a test run, **206**
– of a test suite, **206**
fail fast, 58, **233**
failure, **19**
failure analysis problem, **256**
failure detection, 13, 20
failure recovery, 204
fairness, 95, 133
fairness constraint, 6
fairness of the SUT, **186**
fairness proviso, **295**
$\text{fairness}_{\mathbb{S}\approx_{\mathcal{S}}}$, **222**
fairness_{strong}, 109
$\text{fairness}_{strong}(a)$, **95**
fairness_{weak}, 109
$\text{fairness}_{weak}(a)$, **95**
faithful, 59
fake progress cycle, **132**
false (PROMELA), 44
false, **9**, 9
false alarms, **25**
false negative, **25**
– for a test generation, **210**
false positive, **25**
– cycle warning, 316
– for a test generation, 210
false positives, 158
false sharing, **51**, 143
fancy TO, 352
fancy_{linear}, **314**
$\text{fancy}_{nonlinear}$, **314**
fast heuristics, **268**
fast-path-slow-path, 142
fault, **19**, 176
fault finding effectiveness, **20**
fault handling, 349, 385
fault injection, **23**
fault tolerance, 328
fault-based testing, **23**
fault-tolerance, 43, 53, 179, 186, 204, 280
fault-tolerant, 272, 273, 277
fault/failure model, **19**
faultable(·) (for *Straces*), 203
faultable(·) (for states), 202
faultable reduced suspension traces, **203**
faultable states, **202**
FAuST, **173**
$f_{distr}(\cdot)$ (coverage), **302**
$f_{distr}(\cdot)$ (for nondeterminism on output), **303**
feasibility, **4**
feature, 1, 260, 329
feature change, **346**
Feature Driven Development, 342
feature interaction problem, **163**, 176
feature oriented software development, 176
feature requirement, 176
FeaVer, **176**
feedback, 23, 331
fi, **45**
F_i^b, **106**
FIFO, **125**
FIFO buffer, 41
FIFO queue, **125**
files, 184
final state, **37**
Findbugs, 4
FindBugs, 344
finingPath2W, **302**
finite, 36
finite ∪ infinite trace semantics, **73**
finite automaton, **37**
finite graph, 28
finite path, 33
finite state machine, **37**, 176
finite system, 3
finite trace semantics, 72

finitely branching, **204**
finitely monotonically increasing $path2W_o$ towards TO, **312**
finitely monotonically increasing sequence, **305**
finitization, **157**
first come first serve, 127
first order formula, **29**
first order logic, **28**
first order structure, **30**
fitness evaluation, 288
fitness function, 305
fix-point, **105**
– greatest, **105**
– least, **105**
fixed-point decimals in Dumont, 325
FixedSizeBitVectors theory, 31
$\mathcal{F}_L$ (forgetful transformation), **35**
flags, 331
flexible, **344**
$F^b_{\neg L,i}$, **106**
floating point values, 113
FLTL, **72**
fluent interface, 233
$\mathcal{F}_{mod}$ (forgetful transformation), **206**
FMs, **2**
$f_{new}(\cdot)$ (coverage), **302**
$f_{new}(\cdot)$ (for nondeterminism on output), **303**
FOL, **28**
for all, 10
$\forall$ (as path quantifier), **73**
forgetful transformation for labels ($\mathcal{F}_L$), **35**
forgetful transformation for model component ($\mathcal{F}_{mod}$), **206**
forgetful transformation for TC component ($\mathcal{F}_{TC}$), **206**
forgetful transformation for weights ($\mathcal{F}_w$), **297**
fork, 174, 278
FORM, **29**
$\text{FORM}_{\Sigma_{FOL}}$, **29**
$\text{FORM}_{\Sigma_{FOL}}(Var)$, **29**
– semantics, 30
formal description, 2
formal interface, 39
formal method, **25**
formal methods, **2**, 240
formal parameter, 44
formal specification, 17
formal type parameter, 327
formal verification, **25**
Fortran, 172
Fortschritt mittels Transitionen, viii
forward traceability, 17
fragment of CTL, 153
$free(\cdot)$, **29**
free variables, **29**
frequency, **163**, 388
F_{S^0}, **106**
FSHELL, **175**
FSM, **37**
$\mathbb{S}_{FSM}$, **37**
FSM-based testing, 186, 218
FSMGen, **176**
FSP language, 247, 248
F^b_T, **106**
$\mathcal{F}_{TC}$ (forgetful transformation), **206**
full fence, **51**
full parallel composition, **42**
full path sequence, **265**
full symmetry, **107**
full TC seq, **265**, 301
full test case, 265
full test case sequence, **265**
full trace sequence, **265**
full weighted test case sequence, **301**
full WTC seq, **301**
fully expand, 137
fully expanded, **103**
function parallelism, **50**
function symbols, **29**
functional, **18**
functional abstraction, **59**
functional behavior, 59
functional requirement, **2**, 14
functional requirement specification, **18**
functional testing, **14**, 183
functional verification, **25**
future, **50**
fuzzy traceability, **333**
$\mathcal{F}_w$ (forgetful transformation), **297**

G

G operator, **73**
game, 251
game theory, 307
gap, 60
garbage collection, 354
garp, **116**
gate, **40**
G_b, **122**
gc, 354
gcc, 51

gen, 209
general coverage criterion, **330**
General Inter-Orb Protocol, **116**
generalized NPC conjunctive Büchi automaton, **129**
generalized theoretical foundation, 6
generateLicense, 349, 385
genetic algorithm, **288**
Genetic Network Analyzer, 110
gen_{exec}, 209
genTC($\mathcal{S}, \ddot{s}$), **211**
genTS($\mathcal{S}, \ddot{s}, b$), **215**
genWTS($\mathcal{S}, \pi, b$), **299**
Gewichte, viii
giop, **116**
glass-box testing, **14**
global LTS, **46**
global model checking, **93**
global progress state, **115**
global property, 176
global state, **46**
global variable (PROMELA), 44
globally operator, **73**
GLSL, 172
GNA, 110
GNU Compiler Collection, 51
GNU Unix to Unix Copy, 116
Go, 172
goal proviso, **295**
good adversary, 307
GO packet, 163
goto, **46**
graph
  – single-rooted, connected, directed, acyclic, 203
  – single-rooted, connected, finite, directed, acyclic, 28
graphical user interface, 233
GraphML, 249
GraphViz, 249
gray-box testing, **15**
greatest fix-point, **105**
greedy, **54**, 263, 296
Grid broker scheduler, 273
grid computing, 273
GridUnit, **273**
Group Address Registration Protocol, **116**
guarantee of discharging, 293
guarantee of discharging all test objectives, **293**
guard, 43, 249
  – in PROMELA, **45**
  – in STSs, **40**
guard (POR), **103**
guard-based POR, 103, 137
guarded command, **43**, 112
guarded command language, **43**
GUI, 325
guidance, 3, 23, 24, **256**
guidance heuristic, **264**
guidance heuristics, 7, 223, 254, 259, **286**, 336
guides, **249**

H

Hadoop, 52, 273, 275
HadoopUnit, **273**
halt, 89
hang, 115
happy path, **19**, 195
hardware, 388
Hardware Lock Elision, 51
hardware raid, 354
hardware transactional memory, 50
hardware-in-the-loop, 237
hash collision, 108
hash compaction, **108**, 112
hash function, 108
hash table, **99**
Haskell, 172, 274
hazard, 179, 228
Hazelcast, 48, **53**, 276, 278, 336
Hazelcast Management Center, 278
heap, 94
heap symmetry reduction, 112
heavyweight formal method, **2**
heavyweight specification, 3, 218
help command, 332
helping pattern, 142
hesitant alternating automaton, 101
heuristics, 6, 159, **283**
  – guidance, 286
  – meaningfulness, **283**
  – phase, **258**, 284
  – test purpose, **286**
  – test selection, **283**, 286
hidden I/O between components, **189**
hide an action, 189
hierarchical refinement, 229
high performance computing, 49
high performance verification, 109
high volume automated testing, **274**
HIL, 237
history, 266, 267

HLE, 51
Hoare triple, 39
hold, 66
homing sequences, 204
homomorphic, 310
horizontal scalability, **49**
horizontal tracing, 18
hot spot, 316, 368
human communication, 2
human error, **19**
hybrid systems, 186
hyper-threading, 275

I

I (interpretation function), **35**
I flag, **331**
i-prot, **116**
i-Protocol, **116**
I/O contention, 388
I/O-bound, 389
IDE, 343
i_δ, **204**, 336
i_δ caching, **336**
idempotence, 3
IdGenerator, 53
idle mode, 370
idleness, 186
if, **45**
IF, 248
if and only if, 10
IF specification, 247
iff, **10**
ignoring problem, **104**
$I^{lazy}(\cdot)$, **263**
illegal pointer access, 158, 169, 173
ILR, **172**
image finite, **197**
immutability, 328
imperative, 44
imperative languages, 43
implementation, **18**
implementation, 276
implementation independent, 22
implementation of parallelism, 111
implementation relation, **202**, 229
implementations, 7
Implementierung, viii
implicit enumeration, **92**
implicit nondeterminism, **157**, 172
implicit output transition to fail , **326**
implicit precedence, 26, 30
implicit reset, **186**
implicit scheduling, 57
implicit state model checking, **92**
implicit synchronization, 57
$in(\cdot)$, 194
in-memory, 53
in-memory database, 53
inactive, **260**
$in^{\cap}(\cdot)$, 228
$in^{\bigcap}_{\mathcal{S}_{\tau^*}}(\cdot)$, 228
incident, 2
incomplete, 60, 291
incomplete reduction, **108**
inconsistent state, 204
incremental depth-first search, **122**
incremental depth-first search with FIFO, **125**
incremental hashing, 111
incremental process, 19
incremental solving, 334
incremental tree compression, 112
independent, **104**
independent observation, 355
independently, 21
indeterminism of nature, 194
indivisibly, **47**
INDUCING, **263**
inducing state, **258**, 285
inducing states, 254
inducing superstate, **258**
inducingness, 258
inequalities between $p2W^{TO}$ and $p2W^{\mathcal{T}\mathrm{O}}$, **306**
inequality in Dumont, 325
inert process, **41**
infinite data structure, 67
infinite data structures, **54**, 110
infinite model checking, **94**
infinite state model checking, **94**
infinite system, 2
infinite systems, 186
infinite trace semantics, **71**
infinitely branching, 212, 290
infix, 194
influencing parameter, 370
informationally unencapsulated, 184
informed compression, 112
inhibited, 184
init, 188
init, **33**
init , **44**
init process, **44**
initial state, **32**, 37

initial states, 40
initializer, **44**
$init_{\mathcal{S}}$, **33**
$init_{\mathcal{S}_{det}}$, 196
$init_{\mathcal{S}_{udet}}$, 228
inlined, **156**
inlining, 21
input, **184**, 187
input action, 42
input completion, **199**
input coverage criteria, 22
input gate, **234**
input output symbolic transition systems, 40
input prefix, **42**
input-complete $\mathcal{S}$, 199
input-complete initialized STS with I/O, 234
input-enabled
– $\mathcal{S}$, 199
– state, **199**
$in_{\mathcal{S}_{\tau^*}}(\cdot)$, 194
Insense, **177**
instantiated state (for STSs), **41**
instantiator, **250**
instruction coverage, 328
instruction reordering, 51
instruction sets, 172
instrumentation, 14, 173
integer overflow, 158, 173
integer underflow, 158, 173
integers in Dumont, 325
integrated, 18
integration test, 328
integration testing, **18**, 189
Intel, 50
Intel Core i3-2100 processor, 275
Intel Xeon E5430, 354
intelligent code completion, 343
interaction, 325, 329, 331
interaction coverage, 334
interaction points, 41
interaction variable, **40**
interactive, 250
interactive theorem proving, 3
interactive verification, **25**
interface abstraction, **184**, 207
interface automata, 251
Interface based on a Partitioned Next-State function, **110**
interleaving, **42**
– in SPIN, **47**
intermediate language, 172
intermediate logic representation, **172**
intermediate representation, **157**, 173, 177
– LLVM, 172
internal, **188**
internal discharge, **276**, 353, 371
internal nondeterminism, **194**
internal transition, **188**
Internet, 53
Internet Protocol, 52, 116
– Version 4, 52
– Version 6, 52
interpretation
– first order logic, **30**
– Kripke structure, **35**
– propositional logic, **26**
– STS, **41**
interpretation function, 35
interprocedural, 175
introspection window, 339
intrusion detection, 161
intrusion model, 176
Ints theory, 31
INV, 39
invalid bit shift, 173
invalid frees, 173
invalid memory access, 173
invariant, 4, **39**
– of $\mathrm{DFS}_{\mathrm{prune,NPC}}$, 128
invariant under stuttering, **76**
invisibility proviso, **104**
invisible, **105**
IOchooser, **250**
ioco, 202
ioco theory, **183**
iocoChecker, **233**, 251, 261
$ioco_{\mathcal{F}}$, 227
IOSTS, 40
$\mathcal{IOTS}_A(L_I, L_U)$, **199**
$\mathcal{IOTS}_A(L_I, L_U, x_1, \ldots, x_n)$, **199**
$\mathcal{IOTS}_A(L_I, L_U, x_1, \ldots, x_n, *)$, **199**
$\mathcal{IOTS}_A(L_I, L_U, x_1, \ldots, x_n, +)$, **199**
IP, 52
IPv4, 52
IPv6, 52
IR, **157**
irregular topology, 374
irregular underspecification of input, **228**, 350
is within, **93**
$I_{\Sigma_{FOL}}$, **30**
ISIS, 345
ISO 26262, 4
isomorphic, 28

iteration modeling, **55**
iterative deepening, 122
iterative process, 19

J

Jambition, 252
Jararaca, 248, 249
Java, **44**, 325
Java bytecode, 172, 173
Java Modeling Language, 3
Java Pathfinder, 173
Java Virtual Machine, 173
JBoss, 371
Jenkins, 343
JIT compiler, 173
JML, 3, 39, 44
job queue, **163**
job scheduler, 370
Joshua, **273**
JPF, **173**
JTorX, viii, 7, 110, **249**, 325
JTorX GUI, **250**
JTorX Guides, 249
jtorx log, 249
Julia, 172
JUnit, 243, 273, 275
justice, **95**

K

keyword, 60, 208
keyword driven testing, 60, 208
kill a model-based mutant, 261
kill a mutant, **23**
kind of testing, 14
KLEE, 176, 316
KLEE tool, **174**
Kleene closure operator, **10**
Kleene's closure operators, **33**
KleeNet, **176**
Komposition, viii
Kripke frame, **32**, 36
Kripke structure, **34**
Kripke structure with state vector semantics, **91**
kripke structures, 34
$\mathbb{S}_{Kripke,<\omega}$ (Kripke structures for finite trace semantics), 71
$\mathbb{S}_{Kripke,\geq\omega}$ (Kripke structures for infinite trace semantics), 71
$\mathbb{S}_{Kripke,\Sigma_{sv}}$, **91**

L

$\xrightarrow{l}$, **35**
L, **35**
L (for STSs), **40**
label, **35**
– in SPIN, **46**
label (for STSs), **40**
labeled Kripke structure, **36**
labeled kripke structures, 36
labeled transition, **35**
labeled transition system, **35**
labeled transition system with inputs and outputs, **187**
labeled transition system with inputs and outputs and internal transition, **188**
labeled transition systems, 35
lambdas, 55
Landau notation, **119**, 196, 197, 241, 242, 244, 267, 268, 299
language containment check, **98**
lasso, **33**
lasso, LTS, **35**
lasso,FSM, **38**
latency, **48**, 367
lazily, 51, 94, 223, 229, 230, 248, 253, 278, 280, 338, 345
lazy, vi, 5, **54**
– cache removal, 338
– SMT, 55
– tabu search, 338
– variable reordering, 28
lazy encoding, **55**
lazy evaluation, **54**
lazy full expansion due to progress, **137**
lazy initialization, **54**
lazy model checking, 56
lazy on-the-fly MBT, vii, 6, **253**
lazy progress traversal, **121**
lazy scheduling, **57**
lazy SMT, **55**
lazy SMT solving, **31**
lazy specification, **55**
lazy store operations, 113
lazy systematic unit testing, **55**
lazy task creation, 57
lazy technique, 6
lazy techniques, 6
lazy threads, 57
lazy traversal sub-phase, 336, 339
lazy traversal sub-phase optimization, **318**, 353

lazy traversal sub-phases, 317, 339
lazy visibility proviso
– $C2^{\mathfrak{T}}_{\text{lazy}}$, **138**
– $C2^{S}_{\text{lazy}}$, **138**
$L_{\delta\epsilon}$, 191
leader, **116**
leader election protocol, **116**
leader_{DKR}, **116**
leaf node, **162**
least fix-point, **105**
LED, 163
length of a path in a TS, **33**
length of a path in an LTS, **35**
length of a trace, **35**
$length_{discharge}$), 320
L_{ϵ}, 188
$L_{\epsilon\delta}$, 191
level 0 OTF, **56**
level 1 OTF, **56**
level 2 OTF, **56**
License Central, **231**
lifting, 234
lightweight, 175, 343, 344
lightweight formal method, **3**
lin-lin plot, xx, 356
linear, **67**
linear Kripke structure with finite length, **67**
linear refinement, 208
linear scaling, **49**
linear speedup, **49**, 151
linear test case, **195**
linear time formula, **70**
Linear Time Logic, **74, 75**
linear time property, 66, **67**
link mobility, **41**
list collection, 53
list modulo a given depth, 262
lists in Dumont, 325
livelock, **115**
livelock detection, 89, 109
livelocks, 89, 95, 142, 189
liveness, 95
liveness check, **109**
liveness properties, 109
liveness property, **68**
LL/SC/VL, 51
LLBMC, **172**
LLDET, 131
$L_{livelock}$, **117**
LLVM, 112, **172**
LLVM compiler infrastructure, **172**
LLVM Compiler Infrastructure, 174
LLVM intermediate representation, **172**
LLVM IR, **172**, 174
LNDFS, 111
load, 388
load balancing, **50**, 143, 278
load balancing, 143
Load-Linked, Store-Conditional, Validate, 50
local LTS, **46**
local progress state, **115**, 133
local properties, **167**
local property, 167
local queue, 143
local stack, 138
local state, **46**, 115
local transitions, **111**
local variable (PROMELA), 44
localBackup, 349, 385
location, **40**
location variable, **40**, 314
lock-free, 50, 142
lock-free hash table, 141
lock-free program, **142**
lockless, **142**
lockless shared hash table, 111
locks, 111
lockstep, 47, 242, 256
log-lin plot, xx, 356
logging, **328**
logic, **31**
logical encoding, 157
logical input map, 175
loop body execution, 156
loop unwinding, 288
lossless abstraction, **59**, 207
lossy abstraction, **59**, 155, 207
lossy hashing, **108**
LOTOS, 43, 247, 248
low coupling, 273
low energy, 162
Low-Level Bounded Model Checker, **172**
low-level logging, **329**
$l(s)$, 36
L_{τ}, 188
$L_{\tau\delta}$, 191
LTL, **74, 75**, 109
LTL_{-X}, **76**
LTS, **35**
– deterministic, 36
– nondeterministic, 36
LTS semantics of PROMELA, 46
LTS with I/O, **187**
LTS with I/O and τ, **188**

LTS$_{\text{MIN}}$, 249
$\mathcal{LTS}(L_I, L_U)$, **187**
$\mathcal{LTS}(L_I, L_U, *)$, **187**
$\mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n, *)$, **187**
$\mathcal{LTS}(L_I, L_U, +)$, **187**
$\mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n, +)$, **187**
$\mathcal{LTS}(L_I, L_U, , \delta, \epsilon)$, **191**
$\mathcal{LTS}(L_I, L_U, \delta, \epsilon, *)$, **191**
$\mathcal{LTS}(L_I, L_U, \epsilon)$, **188**
$\mathcal{LTS}(L_I, L_U, \epsilon, *)$, **188**
$\mathcal{LTS}(L_I, L_U, \epsilon, \delta)$, **191**
$\mathcal{LTS}(L_I, L_U, \epsilon, \delta, *)$, **191**
$\mathcal{LTS}(L_I, L_U, \tau, *)$, **188**
$\mathcal{LTS}(L_I, L_U, \tau, \delta)$, **191**
$\mathcal{LTS}(L_I, L_U, \tau, \delta, *)$, **191**
$\mathcal{LTS}(L_I, L_U, x_1, \ldots, x_n)$, **187**
LTSmin, 101
LTS$_{\text{MIN}}$, **109**
Lua, 172

M

macro mechanism, 60
maintenance, 15, 237, 287
make progress, **115**
malicious, 162
manager, **250**
Manifesto for Agile Software Development, **342**
manual test case generation, **15**
manual testing, **15**
manual verification, **25**
manually, **155**
map, 52
map collection, 53
MAPLE, 110
MapReduce, **52**, 278
MapReduce Master, 52
MarginSafetyMiniTT, **318**
Markov decision process, **113**
masked, 21
master, **52**
master/slave network, **52**
Max, **308**
maximal bound, **258**
maximal depth bound, **285**
maximal environment, **61**
maximal paths, **33**
maximal test run, **206**
maximal test run path, **206**
maximal test run trace, **206**
maximal value, **355**
MaxMax, **308**
$max(p2W, max(\ldots))$ (aggregation), 307
$max(p2W, min(\ldots))$ (aggregation), 307
$max(x)$, 355
MBT, 4, **238**
MBTAD, **345**
MC, 3, **90**
MC/DC, 21
mCRL2, 43, 110, 111, 249
MDD, **107**
MDP, **113**
mean adversary, 307
mean value, **355**
meaningful, **20**, 203, 213, 283
meaningfulness, vii
meaningfulness heuristics, **283**
meaningfulness weight heuristics, 297
median, **355**
$median(x)$, 355
medicine, 1
memory, 184
memory barrier, **51**
memory block, 51
memory bound, **49**
memory consistency, **51**
memory consistency model, 51, 113
memory errors, 158, 173
memory fence, **51**
memory model, 51
  – bit-precise, 172
memory requirement, 371
memory requirements, 353
Mersenne Twister, 351, 371
message (process algebra), **41**
message passing, **41**, 48
message passing implementation, 52
message passing interface, **48**
message sequence chart, 250
meta level, 6, 7
meta-mutant, 179
meta-state, **196**
Metaebene, viii
metaheuristic, 290, 337
metric, 310
metric-driven, **289**
minimal bound, **258**
minimal depth bound, **285**
minimal value, **355**
minimization, 111
minimized automata compression, **107**
minimized automaton compression, 92
minimum robustness specification, 232
Minisat2, **27**, 158

min(*x*), 355
MISRA C standard, 157
misses, **25**
mistake, 19
mitigation, 296
mitigation of LazyOTF's guidance, 292
mitigation via weights, **302**
mobility, **41**
MOD, **185**
model, 34
– behavioral property, **66**, 66
– FOL, **30**
– for abstraction, **59**
– for ioco, **185**
– propositional, **26**
– temporal logic, **69**
model checking, v, 3, **90**, 240
– bounded, **93**
– conditional, **91**
– exhaustive, **93**
– explicit state, **92**
– implicit state, **92**
– infinite state, **94**
– offline, **93**
– on the fly, **93**
– software bounded, **94**
– symbolic implicit via BDDs, 105
model nondeterminism of the LTS, 189
Model Programs, **251**
model-based mutant, 261
model-based mutation testing, **261**
model-based testing, 3, **238**
model-based testing (abstr. def.), **237**
model-oriented testing, **238**
modeling language, 177
Models, **69**
$Models_{lin}$, **69**
Modified Condition/Decision Coverage, 21
modularization, 43, 47, 60
monitor, 161
monitoring, **14**, 271
monitoring application, 161
monitoring execution step, 44
monotonic POR, 103
Moore's law, **47**
more expressive than, **80**
mortality, 370
move towards TO in the current traversal sub-phase, **312**
mpC, 273
MPI, **48**
MTBDD, **113**
mtype (PROMELA), **44**
Muller automaton, 77
multi-core, 274
multi-core BFS, 146
multi-core computing, **48**
multi-core DFS, 147
multi-core multiprocessing, 48
multi-terminal binary decision diagram, **113**
multi-threaded, viii, 142, 252
multi-threaded DFS_{FIFO}, 6
multi-threading, **48**, 138, 272, 278
multi-valued decision diagram, **107**
multicast, **52**, 116
multicast group, 52, 116
multiple instruction streams, multiple data streams, 48
multiprocessing, **48**, 274
multitasking, **169**
MurPHI, 112
mutant, **23**
mutation, 288
mutation kill ratio, **24**
mutation operator, **23**
mutation score, **24**
mutation testing, **23**
MySQL, 371

N

n (sample size), 355
named arguments, 40
named channel, **41**
NameNode, 52
NDFS, **99**
negation normal form, **75**, 91
negative test, **19**
nesC, **163**
nested depth-first search, **99**
nested dfs, 98
nested parallelism, **57**
nested type, 177
network, 52
– master/slave, 52
– peer-to-peer, 52
network simulator, 167
never, **44**
never claim, **109**, 118
never claim process, **44**
next time operator, **70**
next-free LTL formulas, **76**
next-state function, **111**
next-state interface, **110**
nightly tests, 266, 272

no operation, **188**
node, **48**
node in a sensor network, **161**
non-emptiness game, 101
non-functional, **18**
non-functional abstraction, **59**
non-functional requirement, **2**, 14
non-functional requirement specification, **18**
non-functional testing, **14**
non-functional verification, **25**, 113
non-maximal paths, **33**
non-normal distribution, 371
non-parameterized statistical tests, 371
non-progress cycle, **117**
non-progress cycle check, **117**
non-progress cycles, 109
non-redundant TS, **215**
non-structural coverage criteria, 22
non-uniform memory architecture, 48
nondeterminism, 155, 157, 185, 186, **194**
nondeterminism (abstr. def.), **60**
nondeterminism of the LTS, **195**, 334, 350
nondeterminism on output, **195**, 350
nondeterministic branching, **195**, 350
nondeterministic choice point, 194
nondeterministic choice points, **60**
nondeterministic choices, **60**
nondeterministic LTS, 36
nondeterministic resolution, **60**, 284
nondeterministic system, **195**
nondeterministic systems, 186
nondeterministic transition system, **34**
nondeterministic, static coverage criteria, **216**, 216
nonfancy, **314**
NOP, **188**
normal form, 28
normalized, 303
notInStack proviso, **105**
np_, **46**
NP, 284
NP hard, 283
NP-complete, 27
NPC, **117**
NPC check, **117**
NPC_ϕ, 129
NS-2, 167
null, 41
NULL platform, **163**
nullness, 328
NUMA, 48
number of test steps, **265**
NuSMV, 97

O

ω, 9
o (test objective), 260, 264, 310
$\omega + 1$, 9
OBDD, **28**
object-based DSM system, 48
Objective-C, 172
Observable MC/DC, 21
observable nondeterminism, **194**
observation, **184**
observation objective, **248**
observer, 53
observer pattern, 329
occam, 43
od, **46**
o_{dec}TO, 352
$\ddot{o}$ (active test objectives), 260
$\ddot{o}_{cov}$ TO, 351
$(\ddot{o}_i)_{i\in[1,\ldots,p]}$, **315**
$\ddot{o}_{init}$, **310**
$\ddot{o}_{S_{isol}}$TO, 352
off-the-fly MBT, **243**
offline, 242
– model checking, **93**
– tableau-based LTL model checking, 97
offline MBT, **243**
offline MBT (abstr. def.), **242**
offline technique, **56**
offline test selection, **289**
OMC/DC, 21, 179
Omega constraint solver, 251, 335
ω-automaton, **76**, 76
ω-regular language, 129
OMG, 116
omission failure, 277
OMNeT++, 167
on-the-fly, 210
on-the-fly determinization, 197
on-the-fly MBT, **245**
on-the-fly MBT (abstr. def.), **242**
on-the-fly model checking, **93**
on-the-fly performance, 117
on-the-fly technique, **56**
on-the-flyness, vii, **56**, 93, 131, 150
on-the-flyness for MBT, **259**
one pass, **130**
one pass algorithm, 131
One-Way-Catch-Them-Young, **112**
o_{NFRSML} TO, 352
online MBT, **245**

oom, 145, 354, 355, 370
Opaal, 110
open, **61**
open addressing, **111**
open architecture, 331
open source, 53, 163
open system, 16
OPEN/CAESAR, 39, 110, 248
OpenMP, **48**
operating software, 163
operator basis, 26, 29, 73
operator precedence, 26, 30
optimal test suite, 283
optimistic, 307
optimization, 6, 157
optimization for MC, **111**
optimizer, 172
oracle, **16**
OracleSafetyTree, **318**
ordered
– binary decision diagram, **28**
ordered binary decision diagram, **28**
ordinal, 9
– smallest infinite, 9
ORDINARY, **263**
original encoding of LTL BMC, 106
o_{RSML} TO, 351
OTF, **56**, **245**
OTF limit proviso, **291**
$out(\cdot)$, 194
out-of-memory, 354, 355, 370
out-of-order execution, **51**, 113
output, **184**, 187
output (PROMELA), **45**
output action, 42
output gate, **234**
output prefix, **42**
output-complete $\mathcal{S}$, 199
output-enabled, 203, 309
– $\mathcal{S}$, 199
– state, **199**
$out_{\mathcal{S}_{\delta\tau^*}}(\cdot)$, 194
$out_{\mathcal{S}_{\tau^*\delta}}(\cdot)$, 194
over-approximation, 228
overall number of test steps, **353**
overall worst case space complexity of all traversalSubphase, **268**
overall worst case space complexity of determinization, **197**
overall worst case time complexity of all traversal Subphase, **268**
overall worst case time complexity of determinization, **196**
overflow, 310
OWCTY, 112, 145

P

p, **21**
P, **138**, 275
P (number of processes), **49**
P (predicate symbols), **29**
$P_{vanishing}$, **293**
p_+ (depth bound threshold), **285**
p_- (depth bound threshold), **285**
packages for transmission, **349**, 385
packet fragmentation, 164, 176
padding, **51**
pagination, 232, 264, 349, 385
pan.c, **109**
parallel, 108, 110, 272
parallel composition, **42**
– asynchronous, 42
– full, 42
– restricted, 42
– synchronous, 42
parallel composition operator, 42
parallel computing, **47**
parallel efficiency, **49**
parallel instance, **48**
parallel interaction, 48
parallel interleavings, 195
parallel language mpC, 273
parallel model-based testing, **274**
parallel overhead, 49, 147
parallel performance metrics, 49
Parallel QuickCheck, **274**
parallel scalability, **49**, 53, 138, 277
parallel schedulings, 50
parallel speedup, **49**, 147
parallel test automation, 252
parallelization, 6, 47
– in search-based software testing, 317
parallelized, **47**, 137
parameterized transition system, 334
parantheses in plots, 355, 370
partial coverage, 333
partial function, **34**, 36
partial knowledge, 333
partial order, 230
partial order reduction, vi, 6, **103**, 190
partial specification, **228**
Partition tolerance, 278
partition tolerant, 278

partitioner, **250**
pass
– of a test case, **206**
– of a test run, **206**
– of a test run path, **206**
– of a test suite, **206**
pass, 326
passendste Abstraktionsgrad, viii
passive nodes, **116**
passive testing, **14**
path coverage, 173
path divergence, **241**
path explosion, **174**
path in a TS, **33**
– length, **33**
path in an FSM, **38**
path in an LTS, **35**
– length, **35**
path quantifier
– existential, **70**, 86
– universal, **73**
path reduction, 112
path sequence, **265**
path2W(π), 298
path2Wprob, 321
pathological, 291, 306
paths pruned at depth b, **215**, 299
$paths_{\text{fail}}(\mathcal{S}_{det})$, 213
$paths(\cdot)$ (finite paths set, LTS), **35**
$paths(\cdot,\cdot)$ (paths set, LTS), **35**
$paths_{<max}(\cdot)$ (non-maximal paths set, LTS), **35**
$paths_{<max}(\cdot,\cdot)$ (non-maximal paths set, LTS), **35**
$paths^{fin}(\cdot)$ (finite paths set, LTS), **35**
$paths^{fin}(\cdot,\cdot)$ (finite paths set, LTS), **35**
$paths^{fin}_{max}(\cdot)$ (maximal finite paths set, LTS), **35**
$paths^{fin}_{max}(\cdot,\cdot)$ (maximal finite paths set, LTS), **35**
$paths_{max}(\cdot)$ (maximal paths set, LTS), **35**
$paths_{max}(\cdot,\cdot)$ (maximal paths set, LTS), **35**
$paths^{\omega}(\cdot)$ (infinite paths set, LTS), **35**
$paths^{\omega}(\cdot,\cdot)$ (infinite paths set, LTS), **35**
$paths_{max}(\cdot,\cdot)$ (maximal paths set, TS), **33**
$paths(\cdot)$ (paths set, TS), **33**
$paths(\cdot,\cdot)$ (paths set, TS), **33**
$paths_{<max}(\cdot)$ (non-maximal paths set, TS), **33**
$paths_{<max}(\cdot,\cdot)$ (non-maximal paths set, TS), **33**
$paths^{fin}(\cdot)$ (finite paths set, TS), **33**
$paths^{fin}(\cdot,\cdot)$ (finite paths set, TS), **33**
$paths^{fin}_{max}(\cdot)$ (maximal finite paths set, TS), **33**
$paths^{fin}_{max}(\cdot,\cdot)$ (maximal finite paths set, TS), **33**
$paths_{max}(\cdot)$ (maximal paths set, TS), **33**
$paths^{\omega}(\cdot)$ (infinite paths set, TS), **33**
$paths^{\omega}(\cdot,\cdot)$ (infinite paths set, TS), **33**
$paths_{\mathbb{V}}(\mathcal{S}_{det})$, 213
PB, **145**
PBS, 363
$pc\_$, 46
PCOs, **16**
$\mathrm{P}_{coverage}$, **292**
$\mathrm{P}_{coverViaTOs}$, **293**
PCTL, **113**
p_{curr} (current phase number), 265
$\mathrm{PDFS}_{\mathrm{FIFO}}$, **138**
$\mathrm{P}_{discharge}$, **293**
peephole POR, 103
peer-to-peer, 53
peer-to-peer network, **52**, 273, 276
performance, 14, 18, 59, 186, 240
performance of interaction, **48**
performance quantity, 353
period $(\ddot{o}_i)_{i\in[1,\ldots,p]}$, **315**
persistence, 53
persistence property, **129**, 153
persistent set POR, 103
peterson mutual exclusion, 112
P_{fair}, **295**
PGA, **275**
P_{goal}, **295**
phase heuristics, 6, 254, **258**, 284, 299
phase of LazyOTF, **253**
phase proviso, **295**
Phasen-Heuristik, viii
phases of the software development process, 17
phaseVariant, 295
physical limitations, 47
physical system, 183
$(\pi_i)_{i\in[1,\ldots,1+p_{curr})}$ (path sequence), 265
$\mid \pi \mid$, 33
π-calculus, 41
pid (PROMELA), **44**
π^{full} (concatenated full path), 266
$(\pi^{full}_i)_{i\in[1,\ldots,1+r_{curr})}$ (full path sequence), 265
piggyback (SPIN), 145
$\pi_{\geq k}$, 33
$\pi_{\leq k}$, 33
PInputDefault, **309**

PINS, **110**
pins, 110
PINS2PINS wrappers, **111**
PLASTIC Framework, 252
plug-in, 252
pluggable type checking, 4, 328
plural observation objective, **287**
$+(p2W, max(\dots))$ (aggregation), 307
$+(p2W, mean(\dots))$ (aggregation), 307
$+(p2W, min(\dots))$ (aggregation), 307
$+(p2W, +(p \cdot z_0, (1-p) \cdot mean(\dots)))$ (aggregation), 308
$+(p2W, +(\dots))$ (aggregation), 307
point of control and observation, **16**
point of entry, 21
point of exit, 21
pointer analysis, 112
pointer dereference error, 171
pointerless, 142
polyadic communication prefix, **42**
polymorphic, **125**
polymorphic method, 122, 134, 253, 276, 299
population, 355, 374
POR, **103**, 112
– cartesian, 103
– guard-based, 103
– monotonic, 103
– peephole, 103
– persistent set, 103
– probe set, 103
– sleep set, 103
– stubborn set, 103
Portable Batch System, 370
positional arguments, 40
possibilistic, 60, 186
POST, 39
postcondition, 4, **39**
– of $\text{DFS}_{\text{prune,NPC}}$, 128
– of model checking, 90, 92
– of model-based testing, 238
– of NPC checks, 117
– of test case generation, 209
potential state, 196
potentially infinite data structure, 54
potentially most meaningful, 283
$\text{P}_{\rightarrow OTF}$, **291**
power consumption, **49**
power-management technology, 388
PowerPC, 172
P_{phase}, **295**
practical application, 7
pragmatic approach of quantifying probability, **308**
praktischer Einsatz, viii
PRE, 39
precedence, 26, 30
precondition, 4, **39**
– of $\text{DFS}_{\text{prune,NPC}}$, 128
– of model checking, 90, 92
– of model-based testing, 238
– of NPC checks, 117
– of test case generation, 209
predicate, **21**
predicate abstraction, **102**, 176, 177
predicate symbols, **29**
preemption, **89**
prefix, **33**
prefix, LTS, **35**
prefix, trace, **36**
prefix,FSM, **38**
prefix,trace in an FSM, **38**
preprocessing, 159
Presburger arithmetic, 31
primer, **250**
primitive type, 44
primitive types, 44
printf(), **45**
prioritization of faults, 299
priority of operators, **26**, 30
PRISM, 44, **113**
privilege, 47
probabilistic behavior, 113
probabilistic concast, **161**
probabilistic CTL, **113**
probabilistic mitigation, **292**
probabilistic model checker, 113
probabilistic property, 113
probabilistic structures, 113
probabilistic weights, **320**
probability, **186**
probability distribution, 113, 371
probability of a path, **113**
probe set POR, 103
procedural programming languages, 43
process, **47**
process (abstr. def.), **41**
process (software development), **17**
process algebra, **41**, 110
process algebraic parallel composition operators, **42**
process algebraic specifications, 189
process calculus, **41**
process declaration (PROMELA), 44

process starvation, **109**
process type declaration, **44**
processor load, 388
proctype, **44**
product backlog, **343**
product lines, 176
product release date, 272
program counter, **46**, 94
program monitoring, **14**
program slicing, 3, 72, 102, **107**
programmatic, 233
progress, vi, **115**, 115, 142
  – label, 46
  – state, **46**
progress, **46**
progress (abstr. def.), **89**
progress cycle, **126**
progress state, **115**, 115
  – global, **115**
  – local, **115**, 133
progress state semantics, **132**
progress transition, 6, 7, **132**
progress transition semantics, **132**
progress_counter, **122**
Prolog, 249, 326
PROMELA, **44**, 110, 176, 248, 249
promise, **50**
promise broken, **50**
promise kept, **50**
promise pending, **50**
promote memory to register, 172, 178
Prop, **69**
PROP, **26**
PROP_{Σ}, **26**
PROP_{Σ}
  – semantics, 26
propagate, 19
property, **34**
  – behavioral, **66**
  – temporal, **65**
property check, 112
property description, **89**
property descriptions in temporal logics, **68**
property relevance, 333
property-based random testing, 274
property-based testing, 3, 274
$Prop_{lin}$, **69**
propositional formula, **26**
propositional logic, **26**
propositional variable, **34**
propositional variables, **26**
protocol initialization, 161
proviso
  – ample decomposition, **103**
  – cover via TOs ($\mathrm{P}_{coverViaTOs}$), **293**
  – coverage ($\mathrm{P}_{coverage}$), **292**
  – cycle, **104**
  – cycle implementation, **104**
  – cycleClosing, **105**
  – dependency, **105**
  – discharge ($\mathrm{P}_{discharge}$), **293**
  – emptiness, **105**
  – fair (P_{fair}), **295**
  – goal (P_{goal}), **295**
  – invisibility, **104**
  – lazy visibility $\mathrm{C2}^{\mathfrak{T}}_{\mathrm{lazy}}$, **138**
  – lazy visibility $\mathrm{C2}^{S}_{\mathrm{lazy}}$, **138**
  – notInStack, **105**
  – OTF limit ($\mathrm{P}_{\rightarrow OTF}$), **291**
  – phase (P_{phase}), **295**
  – vanishing ($\mathrm{P}_{vanishing}$), **293**
  – visibility, **105**
  – visibility $\mathrm{C2}^{\mathfrak{T}}$, **136**
  – visibility $\mathrm{C2}^{S}$, **136**
provisos framework, **322**
prune state space, 47
pruned, 158
$P_{\ddot{s}}$ (behavioral property for a TG), 260
$P_{safety} \wedge P_{liveness}$, **68**
pseudo-random number generator, 275
pseudorandom number generator, 351, 371
$\mathbf{P}_{enabled(\ddot{s})}[\ddot{s} \xrightarrow{l} \ddot{s}']$, 320
pure, 328
Python, 172

Q

QF, 31
QML, **252**
qualification problem, 185
quality, **186**
quality management, 345
quantifier-free, 31
QuickCheck, 3, **274**
quiescence, 16, 186, **190**, 207
quiescent, **191**
quotient structure, **107**

R

$?r$ (reified implicit reset capability), 262
R, **73**, 172
Rabin automaton, 77
Rabin-Scott powerset construction, 197
race condition, 276

race conditions, 3, 195, 207
radiation, 184
RAID, 388
ramification problem, 185
random, 167, 245, 270
random receive (PROMELA), **45**
random test selection, 286
random testing, 274
random waiting, 116
random walk, **113**, 175, 255, 286, 357
randomized next-state function, 138
randomly ordered next-state function, 138
randomness, 241
rapid delivery, **344**
rate of progress, 49
RC5 cipher, 167
r_{curr} (current restart number), 265
reachability, 118
reachability properties, **240**
reachability property, **65**, 68, 260, 264
reachable, **34**
reached TG, **260**
reactive system, **157**
read matrix, **111**
read-acquire, **51**
real superstate, **196**
real-time, 273, 278
real-time behavior, **142**
real-time constraint on $\rightarrow$, **186**
real-time system, 33
reassembling errors, 164, 176
receive
  – random (PROMELA), 45
  – standard (PROMELA), 45
receive (process algebra), 41
receive (PROMELA), **45**
receive queue, 164
recomputation, 354
recomputed, 369
record (PROMELA), 44
records in Dumont, 325
recurrence, **95**
recurrent restarts, 262
recurrently, 142
recurrently increasing, **295**
recursive depth, **156**
red-green-refactor-cycle, **343**
reduce, 52
reduced exhaustive test suite, 223
reduced ordered binary decision digram, **28**
reduced traces, 234
reduction heuristic, 248, **288**, 290
reduction rate, 146
reductions, 112
reductions, 112
redundancy, 53, 208
refactor, 343
refinement, 6, **60**, 207, 229, 235
refinement hierarchy, 247, **345**
refinement relation, viii
refines, 230
refines$_{\mathcal{F}}$, 230
reflexive, 188, 230
reflexive closure, 34
reflexive, transitive closure, **36**
reflexivity, 3
refusal, 186, 207
refusal testing, **184**
register, 94
regression bug, **19**
regression test, 343
regression testing, **19**, 237
regular expression, 248, 286
reified implicit reset capability, 262
reifies quiescence, **197**
relational operators in Dumont, 325
relative speedup, **49**, 147, 366
relative standard error of means, **355**
release memory barrier, **51**
release operator, **73**
relevant behaviors, 102
reliability, 1, 13
reliable, 369
reliable reset capability, **186**, 262, 351
remoteBackup, 349, 385
removeAll, 349, 385
removeExpired, 349, 385
removeLicense, 349, 385
renaming, **42**
reordering, 51
reordering optimizations, **51**
repeatability, **319**
repetition, **46**
replication, **42**
reproducibility, 20, **318, 319**, 337
reproduction operator, **288**
REQ, **18**, 167
request-response pattern, 349, 385
requirement, 167, 329
requirements, 2, **17**, 333
  – vague, 333
requirements conformance testing, **22**
requirements coverage, **22**, 252, 271, 310
requirements phase, 18

requirements traceability, 17
requirements-based testing, **22**
reset property, **95**
resolving nondeterminism, **195**
resource, 115
resource constrained, 163
resource consumption, 59
response messages, 349, 385
response property, **129**
response time, 48
REST, 53
restart, 262, **266**
restart heuristic, **262**
restart property, 292
restart sequence, **265**
restricted parallel composition, **42**
restriction, **42**
retain moving towards a TO, 313
return on investment, **341**
reuse, 107
revealing, **20**
reviews, 14
$(r_i)_i$ (restart sequence), 265
right level of abstraction, **7**
right-associativity, **26**
right-unique, 195
right-unique relation, **34**
risk management, 13
roadmap, 1, 13, 32, 39, 65, 66, 70, 116, 121, 134, 138, 145, 156, 161, 183, 187, 208, 227, 247, 284, 325, 341, 349
ROBDD, **28**
robust, 19
robust testing process, **205**
robustness, 14, 18, 179, 240
robustness testing, 232
ROI, **341**
root cause, **19**
rotation, **107**
round, **116**
round bracket, **26**, **30**
RSEM, **355**
$RSEM(x)$ relative standard error of mean, 355
Ruby, 172
run (PROMELA), **45**
runtime conformance monitoring, 271
runtime error, **158**
runtime trace semantics, **72**
runtime verification, **14**, 96
Rust, 172

S

S, **32**
S (for STSs), 40
Σ, **34**
$\Sigma_{sv}(X)$ (variables for X), 106
$\mathcal{I}$, 40
$\mathcal{V}$, 40
$|\mathcal{S}|$, 34
S^0, **32**
safety check, **108**
Safety Integrity Level, 4
safety properties, 108
safety property, **68**
safety standards, 4
safety-critical, 161
safety-critical software, 2, 4
safety-critical systems, **1**
SAL, 101
sample, **355**, 355, 374
sample mean, **355**
sample size, **355**, 371
$\mathcal{S}_\dagger$, **199**
SAT, **27**, 177
SAT modulo theories, **31**
SAT solver, **27**, 92
SAT-solver, 106
SATABS, **177**
satisfies
– Büchi automaton, **77**
– behavioral property, **66**, 66
– ECTL$*$, **86**
– first order logic, **30**
– propositional logic, **26**
– temporal logic, **69**, 69
sawtooth phase heuristic, **326**, 361
sax, 249
SBMC, 3, **94**
SBST, **288**
Scala, 172
scalability, 147, 353
– parallel, **49**, 138
scalable, **49**
scalable up to P, **49**
scalar evolution analysis, 172, 178
scales linearly, **49**
scales well, **49**
scaling of a parallel program, **49**
SCC, **97**, 101, 262
scenario, **287**
scenario (in Spec Explorer), **251**
scheduling, **163**, 195, 273

– in SPIN, **47**
scheduling of tasks, **54**
schema-based mutation, 179
scope, 44
Scrum, 342
$\mathcal{S}_{\delta}$, **191**
$\mathcal{S}_{\delta\tau^*}$, **191**
$\mathcal{S}_{\iota}$, **199**
S_{det}, 196
$\mathcal{S}_{det}$, 196
SDL, 248
search heuristics in EXE, 174
search-based software testing, 241, **288**, 316, 337
search-space explosion, 3
secondary operator, 86
secondary operators, 26, 29, 73–75
security protocol, 176
seeds, 275
selection, **45**
selection hypotheses, 185
selective data hiding, 102, 107
Selenium, 273
selenium grid, **273**
self-clustering, 53
self-discovering, 53
self-loop, **71**
SEM, **355**
semantics, 40, 70, 74, 75, 85
semantics of CTL*, 71
semantics of ECTL*, 86
semantics of $\mathrm{FORM}_{\Sigma_{FOL}}(Var)$, **30**
semantics of PROP_{Σ}, **26**
semi-decidable, 239
$SEM(x)$ standard error of mean, 355
send
– sorted (PROMELA), 45
– standard (PROMELA), 45
send (process algebra), 41
send (PROMELA), **45**
sensor, 161
separation of mechanism and policy, 329
SEQ, 247
sequence (PROMELA), **45**
sequential, 48, 51, 127, 147
sequential composition, **42**
sequential composition (PROMELA), **45**
sequential consistency, 51, 113
server machine, 273
service, 177
service oriented architecture, 116
service-oriented architecture, 53, 231, 349
set collection, 53
set of initially active TOs, **310**
set theoretic semantics, **71**
$SETAGG$ packet, 163
SET packet, 163
Σ_{FOL}, **29**
$(\sigma_i^{full})_{i\in[1,\ldots,1+r_{curr})}$ (full trace sequence), 265
shadow copy of the heap, 112
shape analysis, 112
shared memory, 48, 142
shared memory abstraction, **48**
shared memory architecture, **48**
shared memory programming, 48
shared reduced ordered binary decision diagrams, **28**
shared state storage, 138
shared-nothing architecture, **48**
sharing in functional programming, 54
short (PROMELA), **44**
short vector, **111**
shortest counterexample, **131**
showLicenses, 349, 385
$(\sigma_i)_{i\in[1,\ldots,1+p_{curr})}$ (full trace sequence), 265
sibling states, **34**
sibling transitions, **34**
side effect, **32**, 157, 174
side effects, 184
sifting algorithm, 28
SIGAR framework, **371**
σ^{full} (concatenated full trace), 266
Σ_{sv}, **91**
signature, **34**
signature for first order logic, **29**
signature for propositional logic, **26**
SIL, 237
silent prefix, **42**
simplifications, **159**
simulated parallelism, 47
simulated SUT, 210, **220**
simulation, **165**
simulation, 165
simulation API, 247
simulative execution, **173**
single point of failure, 52, 53
single unwinding, **33**
single-rooted graph, 28, 203
singular observation objective, **287**
sink, **162**
S_{isol}, **222**
size of $\mathcal{S}$, **34**
size of a cycle, **33**
size of a TC, **204**

size of an OBDD, 28
SKEY, 168
skip (PROMELA), **45**
$\mathbb{S}_{Kripke}$, **35**
$\mathbb{S}_{Kripke,1}$ (one initial state), **35**
$\mathbb{S}_{Kripke,finite}$, **35**
$\mathbb{S}_{Kripke,labeled}$, **36**
$\mathbb{S}_{Kripke,labeled,finite}$, **36**
$\mathbb{S}_{Kripke,linear}$, **67**
slave, **52**
Σ_{lazy}, **263**
$[[\mathcal{S}, L]]^b$, **106**
$\mathcal{S}_{LC}$, 349, 385
Slede, **176**
sleep set POR, 103
SLF4J, 329
slicing, 176
slicing rules, 159
slicing scenario, **251**
SLOC, 338
slot, **110**
$\mathbb{S}_{LTS}$, **35**
$\mathbb{S}_{LTS,finite}$, **35**
small scope hypothesis, **93**, 130, 150, 288
Smart Wireless, 161
smells, 184
SMT, **31**, 106, 241
SMT logic, **31**
SMT solver, **31**, 174, 257, 334
SMT solvers, 92
SMT-LIB, 158, 172
SMT-LIB 2.0, 334
SMT-LIB 2.0 standard, 31
SMV, 101
SNA, **48**
SOA, 231
SOAP, 53
software BMC, 94
software bounded MC, 94, 155
software bounded model checking, 3, **94**
software bug, 1
software development, 7
software development platform, 163
software development process, 17, 341
software engineering, 17
software engineering process, 17, 55
software life cycle process, 17
software requirements, **18**
Software System Award, 108
software testing, **13**
software transactional memory, 51
software verification, **25**
software-in-the-loop, 237
sometimes operator, **73**
sorted send (PROMELA), **45**
sound, 3, **25**
  – model checker, **90**
  – model-based testing, **239**
  – SAT solver, **27**
  – test case, **210**
  – test generation, **210**
  – test suite, **210**
sound abstraction, **163**
sound test generation gen_{exec}, **210**
soundness for an ioco variant, 228
soundness of genTC, 213
soundness of genTS, 216
soundness of DFS_{FIFO}, 127
soundness of DFS_{FIFO} with POR, 136
sounds, 184
source, **34**
source LoC, 338
source of a path
  – FSM, **38**
  – TS, **33**
$source(\cdot)$ (source of a path
  – LTS), **35**
  – TS), **33**
S_P (speedup), **49**
Spec Explorer, 245, 251, 287, 335
Spec#, 251
specifiable, **80**
specification, **16**
specification hierarchy, **345**
specification language, **38**
specification language front-end, 250
specification language front-ends, 110
specification mutation testing, **261**
specification-driven development, **345**
specifications (for ioco), **200**
specifications coverage, **22**
specified, **69**
speedup
  – (super-)linear, **365**
  – absolute, **49**
  – linear, **49**
  – parallel, **49**
  – relative, **49**
  – super-linear, **49**
$\mathcal{S}_\pi$, **67**
SPIN, 101, **108**, 176
SPINS, **110**
S_{POR}, 103
$\mathcal{S}_{\text{POR}}$, 103

sprint, 260
sprint backlog, **343**
sprints, **343**
$\mathcal{S}_{\rightarrow^*}$, **34**
$S_{\rightarrow^*}$(state space), **34**
SSA, **157**
SSA-based, 172
$\mathbb{S}_{sim\,\mathcal{S}}$ (simulated SUT), 220
$\mathbb{S}_{STS}$, **40**
stable, 272
stack buffer overflows, 173
standard configurations, **329**
standard deviation, **355**
standard error of mean, **355**
standard receive (PROMELA), **45**
standard send (PROMELA), **45**
starvation-freedom, 142
starve, **89**, 115
starving, **95**
starving process, **109**
state, **32**, 37
– accepting, **37**
– final, **37**
– global, 46
– initial, **32**, 37
– local, 46
state change, 43
state coverage in isolation, 222
state coverage of $\mathcal{S}_{det}$, **216**
state coverage of $\mathbb{M}_{det}$, **216**
state machine, 346
state space, **34**
state space bottleneck, 374
state space compression, **107**
state space explosion, v, 3, 59, **91**, 155
state space partitioning, 112
state space reduction, **107**
state space reduction technique, 102
state transition system, **32**
state variable, **40**
state vector, 46, 91
state vector semantics, **91**
statement, 35
statement merging, 103, **107**, 112
states of *faultable*(S) in isolation, **222**
static, **2**
static analysis, 3, 4, **14**, 157, 172, 343
static code analysis, **14**
static coverage, **24**
static coverage level, **24**
static defect, 19
static dependency matrix, **110**
static polymorphism, 252
static single assignment form, **157**
static testing, 13, **14**, 155
statistically verified, 177
STATUS packet, 162
$\mathcal{S}_{\tau^*\delta}$, **191**
stdin pipe, 250
stdout pipe, 250
steep memory hierarchy, 51
steepest ascent hill climbing, 305, 317
STG, **251**, 335
stimulus, **184**
storing communication between JTorX and LazyOTF, **328**, 354
STP, 172, 174
STP SMT solver, 55
Straces$(\cdot)$, 194
Strace equivalence, **194**
Strace equivalence on sets, 194
$Straces_{\mathcal{S}_{\delta\tau^*}}(\cdot)$, 194
$Straces_{\mathcal{S}_{\tau^*\delta}}(\cdot)$, 194
straight-line instruction throughput, 47
Streett automaton, 77
stress tests, 271
strict on-the-fly, **242**
strict $\text{PDFS}_{\text{FIFO}}$, **143**
strictly monotonically increasing $path2W_o$ towards TO, **312**
strictly monotonically increasing sequence, **305**
strings in Dumont, 325
strong (A-)fairness, 133
strong bisimulation, 111
strong exception safety, 204
strong fairness, **95**, 101, 133
strong mutation testing, **23**
strong scaling, **49**
strong trace link, **346**
strongly connected components, **97**, 137
struct, 177
struct (PROMELA), 44
structural coverage criteria, **22**
structural health monitoring, 1, **161**
structure the LTS, 189
STS, **40**, 233, 234, 249
$\mathbb{S}_{TS}$, **32**
STS with I/O, **234**
STSExplorer, **334**
$\mathbb{S}_{TS,finite}$, **32**
STSimulator, **325**, 325
stubbed, 367
stubborn set POR, 103

stutter, **76**, 312
stutter equivalence, **76**, 136
stutter invariant, **76**
stutter-closed, **76**
stuttering, **71**, 76
stuttering equivalent, 104
stuttering in a finitely monotonically increasing sequence, **305**
sub-graph, 254, 284
sub-linear, **147**
$subF$, **81**
$subF_{lin}(\cdot)$, 83
subsystem, 18
S_{udet}, 228
$\mathcal{S}_{udet}$, 228
suffix, **33**
suffix, LTS, **35**
suffix, trace, **36**
suffix,FSM, **38**
suffix,trace in an FSM, **38**
sum, **42**
super-linear speedup, **49**, 151, 365
super-linear speedup of meaningful test execution, 365
superstate, **196**
$supp(f(\cdot))$, **9**
$supp(I(\cdot, s))$, **35**
$supp(I(p, \cdot))$, **35**
support, **9**, 35
suppress TOs, **306**, 307
SureCross, 161
survival selection, **288**
suspension, **190**
suspension automaton, **191**, 196
suspension automaton after τ abstraction, **191**
suspension trace coverage of $\mathcal{S}$, **216**
suspension traces, **194**
SUT, **16**
SUT, 16
SUT, **183**
sut, 350
SUV, **25**, 158
$[[\mathcal{S}_{\mathcal{V}^d_{LC}}]]$, 350
swarm verification, 111
Swift, 172
switch, **40**
switch relation, **40**
switch restriction, **40**
symbolic, 92, 102, **104**, 173, 255
symbolic branch point, 174
symbolic constant (PROMELA), 44
symbolic execution, 173, **174**, 176, 252, 275, 334
symbolic extended traces, 234
symbolic implicit state model checking via BDDs, **105**
Symbolic Java Pathfinder, **173**
symbolic name, 108
symbolic path, **174**
symbolic paths, **334**
symbolic test generation tool, **251**
symbolic transition system, **40**
symbolic transition system with inputs and outputs, **234**
symbolically, 212
symmetric in its parameters, 310
symmetry reduction, **107**, 112
SymToSim, **325**
synchronization, 48
synchronization primitive, 53
synchronization primitives, 142
synchronize, 42
synchronous communication, 41
synchronous parallel composition, **42**
synchronous product, **42**
synchronous random polling, **57**, 112
synchronous transmission, **41**
synergetic, 255
synergetic effects, 287
synergetic TOs, **260**, 357
syntax, 40, 70, 74, 85
syntax error, 178
system, 1
  – open, 61
system architecture, **18**
system design, **18**
System Information Gatherer and Reporter, **371**
system integration testing, **18**
system requirements, **18**
system requirements specification, **18**
system specification, **18**, **89**
system specification description language, **38**
system specifications (for ioco), **200**
system structure, 18
system testing, **18**
system tests, 328
system under test, **16**
system under verification, **25**, 161

T

T, **32**
$\mathfrak{T}$, **35**

T flag, **331**
T-Check, **175**
t-test, 371
TaaS, 272
tableau-based offline LTL model checking, 97
tabu search, 317, 337
Tarjan's DFS, **97**, 190, 248
task parallel language, 61
task parallelism, **50**
task queue, 164
task-parallel language, **57**
τ, **188**
τ (internal transition), 188
τ as unobservable communication, 189
τ abstraction, **188**
τ-closed, **188**
τ-closure, **188**
τ-cycle, **189**
taxonomy of MBT, **240**
taxonomy of testing, **14**
TC, **204**, 265
TC (abstr. def.), **16**
TC seq, **265**, 301
TCI, 275
t_{curr}, 241, 242
t_{curr}(number of test steps), 265
t_{curr}^{max}(number of overall test steps), 353
t_{curr}^{max}, **348**
t_{curr}^{TO}, 267
TDD, **343**
TDD cycle, **343**, 343
telecommunications systems, 176
temporal logics, **68**
temporal operators of CTL, **74**
temporal property, **65**
term, **29**
TERM, **29**
$\text{TERM}_{\Sigma_{FOL}}$, **29**
$\text{TERM}_{\Sigma_{FOL}}(Var)$, **29**
terminate, 46
termination, 115
termination, 71
termination detection, **143**, 174, 276
termination function, 295, 296
Terracotta, 371
test, 203
TEST, **185**
test adapter, **207**
test adapter (abstr. def.), **16**
test case, **204**, 265
test case (abstr. def.), **16**
test case complexity, 255
test case contamination avoidance, 277
test case coverage, 216
test case execution, **205**, 237
test case execution (abstr. def.), **19**
test case explosion, 2, **239**, 255
test case generation, 237
test case generation algorithm, 208, **209**
test case prioritization, 297
test case refinement, **208**
test case sequence, **265**
test directive, **286**
test driver, 3, **205**, 271
test driver (abstr. def.), **16**
test engineer, **13**
test execution, 205, **206**
test execution control, 278
test execution engine, **249**
test execution step, **19**
test generation, 173, **208**
Test Generation V, **247**
test goal, **260**
test load distribution, 277
test manager, 273
test objective, 7, 253, 260, **264**, 283
– active, **260**
– basic, **310**
– composed, **310**
test objective (abstr. def.), **260**
test objective (for weight heuristics), **310**
test objectives, 351
test objectives tab, **332**
test oracle, 3, 275
test oracles, **205**
test purpose, 243, 245, **286**
test purposes, 283
test run, **206**
test run (abstr. def.), **19**
test run path, **206**
test run trace, **206**, 260
test selection, 3, 7, **283**, 289, 297
– offline, **289**
test selection (abstr. def.), **259**
test selection directive, 240, 248, 249, 251, 252, **286**
test selection heuristics, **286**, 286
test step, **16**
test step during execution, **19**
test suite, **204**
test suite (abstr. def.), **16**
test suite integrity, 278
test suite maintenance, 15

test suite reduction, **23**, 24, 223
test WC time, **353**, 371
test-driven development, 15, **343**
Test-Selektion, viii
Testen von Software mit BMC, viii
tester process, **131**
testExecutionSubphase, **254**
TESTGOAL, **263**
testing, **13**
Testing and Test Control Notation version 3, 275
testing approach, **155**, 155, 167
Testing as a Service, 272
testing automata, 131
testing hypothesis, 6, **185**, 188, 190, 248, 249
testing on the cloud, 272
testing with SBMC, **155**, 155
testing with software bounded model checking, 7, 14
Testobjekte, viii
testware, **16**
text mode, 250, **333**
$(\mathbb{T}_i^{full})_i$ (full TC seq), 265, 301
TG, **260**
TGV, **247**, 289
the free lunch is over, 47
the null platform, 163
the rest of weights being aggregated, **306**
the values of AD, **342**
theory, **30**
theory of uninterpreted functions, 55
theory testing, 369
thread, 1, **48**
thread pool, 50
threats to construct validity, 370
threats to external validity, 372
threats to internal validity, 369
threats to statistical conclusion validity, 371
threats to validity, **369**, 369
throughput, **49**
$(\mathbb{T}_i)_i$ (TC seq), 265, 301
time and space complexities of ltl mc, 101
time and space complexities of ndfs, 101
time deadlocks, 112
time region, 33
timed automata, 110
timeout, **186**, 188, 262
timeouts, 207, 355, 370
timing constraint, 252, 262
timing values, **186**
TinyOS, **163**
TO, **260**, 260
TO coverage level, **310**
TO editor, **330**, 339
TO management, **329**
tool qualification, 4, 237
tool-independent foundation, 183
top-level, 87
topological sorting, **112**
topology, 107
topology of a WSN, 161
TorX, **248**
TorX Explorer programs, 249
TOs, 253
TOSSIM, 167, 175
TosThreads, 177
total memory use, 149
$\mathfrak{T}_{\text{POR}}$, 103
träge Techniken, viii
$trace(\cdot)$ (trace of a path), **35**
trace back, 343
trace in an LTS, **35**
trace length, **36**
trace links, **17**
trace of a path, **35**
trace pattern, **287**
trace semantics
– finite, **71**, 72
– finite ∪ infinite, **73**
– infinite, **71**
– runtime, **72**
trace sequence, **265**
trace/tree explorer, **331**
traceability, **17**, 333, 343
– backward, 17
– forward, 17
traceability tab, 333, 339
$traces(\cdot)$ (traces set), **36**
$traces(\cdot,\cdot)$ (traces set), 36
traces,FSM, **38**
$traces_{<max}(\cdot)$ (non-maximal traces set), **36**
$traces_{<max}(\cdot,\cdot)$ (non-maximal traces set), 36
$traces^{fin}(\cdot)$ (finite traces set), **36**
$traces^{fin}(\cdot,\cdot)$ (finite traces set), 36
$traces^{fin}_{max}(\cdot)$ (maximal finite traces set), **36**
$traces^{fin}_{max}(\cdot,\cdot)$ (maximal finite traces set), 36
$traces_{max}(\cdot)$ (maximal traces set), **36**
$traces_{max}(\cdot,\cdot)$ (maximal traces set), 36
$traces^{\omega}(\cdot)$ (infinite traces set), **36**
$traces^{\omega}(\cdot,\cdot)$ (infinite traces set), 36
$traces_{\mathcal{S}_{\tau^*}}(\cdot)$, 194
transaction, 207
transaction primitive, 273
transactional memory, 50

– hardware, 50
– software, 51
Transactional Synchronization Extensions, 51
transactions, 53
transceiver chip, 164
transform pass, 172, 179
transformation
– forgetful for labels, **35**
– forgetful for model component, **206**
– forgetful for test case component, **206**
– forgetful for weights, **297**
transformational system, 115
transistor, 47
transition, **32**
transition coverage, 252
transition coverage of $\mathcal{S}_{det}$, **216**
transition grouping, 111
transition relation, **32**, 37
transition system, **32**
– deterministic, **34**
– nondeterministic, **34**
transition system (abstr. def.), **36**
transition systems, 32
transition tasks, **55**
transitive, 188, 230
transitive closure, 34, **36**
transparent and automatic distribution, 277
transparent POR, **104**, 137
transportation, 1
trap property, **90**, 270
trap variable, **241**, 248
traveling salesman problem, 284
traversal sub-phase, **253**, 284
traversalSubphase, **254**
traverse progress, **116**
$T_{\rightarrow^*}$, **34**
$\mathfrak{T}_{\rightarrow^*}$, **35**
tree compression, **107**, 145
tree shaped computations, 57
treeSolver, 249, **326**, 335
trema, 10
triangle phase heuristic, **326**
trip count, 178
true (PROMELA), 44
true, **9**, 9
truly OTF, **56**
truly underspecified, **60**, 158
trustworthiness, 369
truth value, **9**
$T(s)$, **34**
TS, **204**
TS (abstr. def.), **16**
TS (transition system), **32**
TSX, 51
TTCN, 248
TTCN-3, 275
TTCN-3 Control Interface, **275**
$\mathcal{TTS}(L_I, L_U, \delta)$, **204**
tuple space model, 273
Turing complete, 41
Turing machine, 25
two literal watching, 55
two person game, 307
two-player game, 251
type(·), 40
type system, 328
typedef (PROMELA), **44**

U

U, **70**
U_{weak}, **73**
U.S. National Center for Health Statistics, 372
UART, 163
UDP, 52
UDP broadcasts, 276, 336
UDP multicasts, 276, 336
uioco, **228**
uioco (w/o using $\mathcal{S}_{udet}$), **229**
UML, 248, 346
UML sequence diagram, 252
UML statechart, 252
UML Testing Profile, 238
UML use case, 252
unary modality, 70
unbuffered channel, **41**
uncontrollable nondeterminism, **194**
undecidable, 3
under the SUT's control, **184**
under the tester's control, **184**
under-approximation, 228
$\mathrm{underspec}_U$, **220**, 221, 228
underspecification, 157
underspecification of input, **228**
underspecification of output, 218, **220**
underspecification of output (abstr. def.), **228**
underspecified, **228**
underspecified suspension traces, **229**
underspecified variable values, 157
unexpected, 184
uniform, **59**
uniquely defined relation, **34**
unit, **18**

unit clause, 55
unit design, **18**
unit test, 328
unit testing, **18**
universal constructors, 142
universal path quantification, 75
universal path quantifier, **73**
universal synchronization primitive, **50**
universe, **30**
unlabeled, 36
unlabeled Kripke frame, **32**
unlabeled Kripke structure, **34**
unlabeled state transition system, **32**
unless, **46**
unless operator, **73**
unobservable communication, **189**
unquantified, 31
unreachable code, **109**
unreliable, **52**
unrolled, **156**
unsharable resource, 89
unsigned (PROMELA), **44**
unsound, 60, 108, 158
until operator, **70**
unwind, 203
unwinding (code), **156**
unwinding (specification), **67**
update, **40**, 249
update mapping, **40**
update of the state vector, 46
UPPAAL, 110–112, 249
UPPAAL timed automata, 112
urefines, 231
usability, 7, 186
usage profile, **287**
use case, 346
use-definition chains, 157
User Datagram Protocol, 52
user interaction, 241, 249, 256, 292, 296
user requirements, **17**
user story, **343**
user-supplied, 24, 94, 172, 248, 258, 283, 285–288, 290, 316, 320, 321
user-supplied assertion, 47
user-supplied assertions, 109
Utraces(·), 194, 228
UUCP, 116

V

$\mathbb{V}$, **20**
V-model, **17**, 247, 343
V-Modell XT, **19**
vague requirements, 333
$val_{\mathcal{D},\beta}$, 30
val_I, 26
valid end state, **46**
– label, 46
valid FOL formula, **30**
valid propositional formula, **26**
validation, 18, 341, **343**
valuation, **30**
valuation tab, **332**
value space of a type, 326
vanish, 302
vanishing proviso, **293**
Var (for STSs), **40**
variable
– global (PROMELA), 44
– local (PROMELA), 44
– propositional, 34
variable assignment (first order logic), **30**
variable assignment (PROMELA), **45**
variable declaration, 44
variable declaration (PROMELA), 44
variable initialization for an STS, **41**
variable order, 28
variable parameters that are measured, 353
variables, **29**
VCC, **173**
$\mathcal{V}^d$ (default variable initialization), 41
VDM-SL, 274
verbose logging, **328**, 354, 370, 388
verdict, **20**, 203
verdict leaf, **204**
verdict of a maximal test run path, **206**
verdict states, **204**
verdicts of a concrete test case, **207**
verdicts of an abstract test case, **206**
$verd_{\mathbb{M}}(\mathbb{T})$, **206**
$verd_{\mathbb{S}}(\mathbb{T})$, **207**
verification, 18, **25**, 343
verification engineer, 65, 96, 115, 186
verification platform, **179**
Verifier for Concurrent C, **173**, 178
verify, 17
Verisoft, 173
Verkürzungsmerkmal, 59
version number, **157**
versioned, **106**
versioned assignment, **157**
vertical traceability, 346
vertical tracing, 19
VirtualBox, 354
visibility proviso, **105**

– C2$^{\mathfrak{T}}$, **136**
– C2^{S}, **136**
visitor, 332
Visual Studio 2010, 251
Visual Studio 2012, 251
visualization, 325, 329, 331
Vollständigkeit, viii
voltage, 388

W

ω, 9
W, **73**
$\omega + 1$, 9
wait-free, 50
wait-free program, **142**
waiting, 207
wallclock time, **353**, 371
waterfall, 19
ω-automaton, **76**, 76
WC time, **353**, 367
WC time on $\mathbb{S}$, **353**
weak Büchi automata, **113**
weak fairness, **95**
weak fairness (SPIN), 133
weak guidance, 245
weak LTL, **113**, 144, 153
weak mutation testing, **23**
weak next operator, **76**
weak scaling, **49**
weak trace link, **346**
weak until operator, **73**
web service, **53**, 252, 349, 385
Web Services Description Language, 53
weight function on $\mathcal{WTTS}$, **297**
weight function on S, **297**
weight heuristics, 7, **297**
weight of $\mathbb{W}$, **297**
weighted test case, **297**
weighted test case sequence, **301**
weighted test suite, **297**
weights, 174
weights editor, **330**, 339
$(\mathbb{W}_i^{full})_i$ (full WTC seq), 301
white-box, 173
white-box testing, **14**, 155
$(\mathbb{W}_i)_i$ (WTC seq), 301
wireless, 161
wireless sensor network, 155, **161**
within, **93**
witness, **68**
witnesses, **162**
wlog, **10**
work distribution, **50**, 57, 277
work duplication, 144, 277
work pruning, 144, 277
work stealing, **57**, 112, 143
worker thread, 138
workload, **49**
worst case latency, 142
worst case space complexity
– of on-the-fly LTL model checking, **101**
worst case space complexity of genTS, **215**
worst case space complexity of genWTS, **300**
worst case space complexity of LazyOTF, **269**
worst case space complexity of CTL model checking, **97**
worst case space complexity of offline MBT, **244**
worst case space complexity of OTF, **245**
worst case test case complexity, **241**
worst case test case complexity of LazyOTF, **269**
worst case test case complexity of offline MBT, **244**
worst case test case complexity of OTF, **245**
worst case time complexity
– of offline tableau-based LTL model checking, **101**
– of on-the-fly LTL model checking, **101**
worst case time complexity of genTS, **215**
worst case time complexity of genWTS, **300**
worst case time complexity of LazyOTF, **269**
worst case time complexity of CTL model checking, **97**
worst case time complexity of offline MBT, **244**
worst case time complexity of OTF, **245**
wrapper, 302
write matrix, **111**
write-release, **51**
WS, **53**
WSDL, 53
WSDL-S, 252
WSDLs, 15
WSN, **155**
WTC, **297**
WTC seq, **301**
WTS, **297**
$\mathcal{WTTS}(L_I, L_U, \delta)$, **297**

X

X, **70**
x86/x86-64, 172

XP, 342
xr assertion, 109
xs assertion, **109**
Xtext, 233

Y

yEd, 385
Yeti, 275
YETI on the cloud, **275**
Yices, 158

Z

Z3, **32**, 158, 334, 335
Zielsuch-Heuristik, viii
Zobrist hashing, 111
Zuul, **334**