Formal Specification and Verification for Automated Production Systems

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

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

genehmigte

DISSERTATION

von

Alexander Sebastian Weigl
geboren in Bad Neuenahr-Ahrweiler

Tag der mündlichen Prüfung: 27. April 2021

Referent:	Prof. Dr. rer. nat. Bernhard Beckert, Karlsruhe Institute of Technology
Korreferent:	Prof. Dr.-Ing. Stefan Kowalewski, Rheinisch-Westfälische Technische Hochschule Aachen

Acknowledgment

A Ph.D. thesis is the end of a long journey that is accompanied by several people and institutions. This section acknowledges their support, although, I know it may not be sufficient in comparison to their efforts.

First, I need to thank my supervisor Prof. Dr. Bernhard Beckert, for the opportunity to follow my research in his group—without his support nobody could read this doctoral thesis.

Also, I want to thank my colleagues over all the time in my research group for their discussions and support (in alphabetical order): Dr. Lionel Blatter, Dr. Thorsten Bormer, Dr. Daniel Grahl, Dr. Sarah Grebing, Dr. Simon Greiner, Dr. Mihai Herda, Dr. Markus Iser, Michael Kirsten, Jonas Klamroth, Dr. Tianhai Liu, Jonas Schiffl and Annika Vielsack. A special thanks to my supervising post-docs: Dr. Mattias Ulbrich and Dr. Vladimir Klebanov for their help and teaching. Further, thanks go to our administrators Simone Meinhart and Ralf Koelmel.

Thanks to Prof. Dr. Stefan Kowalewski (RWTH) for participating as the second reviewer on my way to the doctoral degree. Moreover, I need to thank Dr. Dimitri Bohlender (RWTH) for the fruitful discussion on PLC topics.

Also, I supervised students in the field of automated production systems. Some of their work can be found in a refurbished and updated form in this thesis. But regardless of this circumstance, I thank my students who worked with me in this field, namely, Anja Blechinger, Matthias Gorenflo, Daniel Lentzsch, Augusto Modanese, Andreas Wieland, and Moritz Baumann.

During my doctoral research, I was funded and supported by several research projects: First, *IMPROVE APS* project was part of the DFG Priority Program 1593 (Design for Future – Managed Software Evolution), which gave a fruitful environment for the discussion on software evolution. Especially, I thank my immediate research partners; in person: Suhyun Cha, Dr. Sebastian Ulewicz and Prof. Dr. Birgit Vogel-Heuser. The results of *IMPROVE APS* provide the main part of this thesis. Second, the *DeduSec* project and DFG Priority Program 1496 (Reliable Secure System) was my introduction to the academic world, and also to secure information flow and security analysis. Third, *KASTEL*, the Competence Center for Applied Security Technology in Karlsruhe was a steady company on my journey and the environment on my work on verified security of production systems. In this context, I want to thank Fraunhofer IOSB for the industrial case study.

Last, but not least, a special thanks to my parents for their steady and long and steady support through all the years of my under-, graduate, and doctoral studies.

Summary

Motivation

Complex industrial control software often drives safety- and mission-critical systems, like automated production plants or control units embedded into devices in automotive systems. Such controllers have in common that they are *reactive systems*, i. e., that they periodically read sensor stimuli and cyclically execute the same program to produce actuator signals.

The correctness of software for automated production is rarely verified using formal techniques. Although, due to the Industrial Revolution 4.0 (IR4.0), the impact and importance of software have become an important role in industrial automation.

What *is* used instead in industrial practice today is testing and simulation, where individual test cases are used to validate an automated production system. Three reasons why formal methods are not popular are: (a) It is difficult to adequately formulate the desired temporal properties. (b) There is a lack of specification languages for reactive systems that are both sufficiently expressive and comprehensible for practitioners. (c) Due to the lack of an environment model the obtained results are imprecise. Nonetheless, formal methods for automated production systems are well studied academically—mainly on the verification of safety properties via model checking.

Contribution

In this doctoral thesis we present the concept of (1) *generalized test tables* (GTT), a new specification language for functional properties, and their extension (2) *relational test tables* (RTTS) for relational properties. The concept includes the syntactical notion, designed for the intuition of engineers, and the semantics,

#	ASSUME			ASSERT		⏲
	mode	learn	I	Q	W	
1	Active	—	—	0	TRUE	—
2	Learn	TRUE	q	0	FALSE	1
3	Learn	TRUE	p	0	FALSE	1
4	Active	—	$[p, q]$	$[p, q]$	FALSE	—
5	Active	—	$> q$	q	FALSE	5
6	Active	—	$< p$	p	FALSE	5

Figure 1: A GTT for the MinMaxWarning function block Appendix B.1.

which are based on game theory. We use RTTs for a novel confidential property on reactive systems, the provably forgetting of information. Moreover, for regression verification, an important relational property, we are able to achieve performance improvements by (3) creating a decomposing rule which splits large proofs into small sub-task. We implemented the verification procedures and evaluated them against realistic case studies, e. g., the Pick-and-Place-Unit from the Technical University of Munich.

Generalized Test Tables. We present the concepts and logical foundations of GTTs, a specification language for reactive systems. GTTs extend the concept of (concrete) test tables, which are already frequently used in the quality management of PLC systems. The main idea is to allow more general table entries, thus enabling a table to capture not just a single test case but a family of similar behavioral cases. Nonetheless, of the added syntactical notions, we try to preserve the intuitiveness and comprehensibility of the concrete test tables. In particular for system design engineers who are experts in test case specification but are not familiar with the formal temporal specification.

Figure 1 shows a typical GTT which is the specification for a diagnosis module. During a training mode, this module learns an expected range of a sensor variable. And later in operation mode, the module normalizes the given sensor value and also signals an error if the given value is out of the learned range. Typically, a GTT has two sides: one for the input variables and one for the output variables. Each cell contains constraints on a designated column variable. The rows form the steps of the defined (test) protocol. They are consecutively applied from top to bottom, where a row or a group of rows can be skipped or repeated depending on the given constraints in the duration column (⏲). If multiple successor rows are possible, the protocol branches nondeterministically.

In this work, we formally define the syntax and semantics of GTTs. The

#	CTRL		ASSUME		ASSERT	⏲
	a	b	**I**	b»$SFCReset$	**O**	
1	▶	▶	=	a»$SFCReset$	=	≥ 1
2	⏮$_0$	▶	=	TRUE	=	1

Figure 2: This RTT specifies that resetting with $SFCReset$ results into the same behavior, as running the system from its initial state. The program runs a and b are from the same program.

semantics of GTTs defines the conformity of a system to a GTT, which is based on a two-party game over infinite words between the environment and the challenger. This game can be encoded into a model which can be efficiently validated by state-of-the-art model-checkers. We demonstrate the applicability of the language with realistic examples from the automation industry and show the feasibility of the verification. Aside from the verification with a model checker, we show how GTTs can be used for runtime verification by generating monitoring software modules.

Relational Test Tables. Relational test tables (RTTs) are an extension of GTTs to allow the specification of relational properties [Wei+20]. A functional property only specifies the behavior of one program run, e. g., an invariant that needs to hold in every reachable state, whereas a relational property talks about multiple program runs. Relational properties enable us to use existing software as a functional specification. Thus, the remaining relational specification only needs to define the relation between the used program versions. A typical application scenario is regression verification—used to ensure that no unintended change of the behaviors was introduced during the software evolution.

RTTs (Figure 2) extend the syntax of GTTs to cope with the multiple program runs. First, program variables are qualified with the identifier of the corresponding program. Secondly, we introduce a user-defined projection function in the column header, which maps the current states of the k program runs to an n-tuple on which appropriate predicates can be applied (e. g., $=$ for $n = 2$, Figure 2). Third, control commands (▶, ⏸, ⏮$_r$) allows the manipulation of the program execution. They allow us to break up the synchronous (or lock-step) execution of the program runs.

The semantics of RTTs is defined by reduction to GTTs. The control commands are handled by program transformation of the original software before we supply the product program to the GTT verification engine. We show the applicability and feasibility of the specification and verification of RTTs on typical examples from software engineering.

Modularization. A further contribution is a sound and complete modularization approach for regression verification [WUL20]. With regression verification, we prove the absence of unintended changes between software revisions. For long-deployed automated production systems, regression verification helps to carry earned trust in operation to the next software version. The idea behind regression verification is that structural and semantic differences between the software revisions are limited, for example only a few procedures are changed. The modularization approach allows us to decompose the proof obligation into smaller sub-goals on smaller program pieces.

The main goal of modularization is to increase the verification performance, which is mainly carried by using a simpler and faster regression verification algorithm. Employing the simpler algorithms becomes possible because the modularization decomposes large programs into smaller program parts. A behavioral difference of these program parts tends to be small (or even non-existent). For example, checking equivalence by comparing the abstract syntax trees is fast, and sufficient for an unchanged procedure like used common library functions. The performance evaluation of the modularization shows that the decomposing enables the regression verification of large evolution scenarios.

The main difference to common modularization or abstraction techniques for functional verification is that the modularization needs to be done coherently applied in two programs. For the coherence, our approach needs markings of the module boundary in the program code, whereby these boundaries can be user-defined, or automatically inferred from the software structure. For each pair of corresponding modules from both programs, the user needs to specify a regression contract, which has three parts: (1) a condition, when the contract is applicable, (2) the assumed relation of the input variables and (3) the guaranteed relation of the output variables between both modules.

Closing The presented contribution follows the idea of lowering the obstacle of verifying the dependability of reactive systems in general, and automated production systems in particular for the engineer either by introducing a new specification language (GTTS), by exploiting existing programs for the specification (RTTS, regression verification), or by improving the verification performance.

Zusammenfassung

Motivation

Komplexe industrielle Steuerungssoftware bestimmt die Abläufe von sicherheits- und unternehmenskritischen Systemen wie sie in automatisierten Produktionsanlagen oder in eingebetteten Steuergeräten, wie z. B. in Automobilen, zu finden sind. Solche Steuerungen sind *Reaktive Systeme*, d.h. sie werden periodisch ausgeführt. Dabei reagieren sie auf die Sensorwerte der überwachten Umgebung, führen das hinterlegte Programm aus und bestimmen die nächsten Aktionen der Aktuatoren. Die Korrektheit von Software für automatisierte Produktionsanlagen wird selten mit Formalen Methoden überprüft, obwohl aufgrund der Industrie 4.0, die Auswirkungen und Bedeutung von Software eine zunehmende wichtigere Rolle in der industriellen Automatisierung spielt.

Stattdessen setzt die industrielle Praxis auf das Testen und die Simulation, bei der nur einzelne Testfälle zur Validierung eines Produktionssystem überprüft werden. Formale Methoden sind u. a. aus folgenden drei Gründen nicht verbreitet: (a) Die Formalisierung von (den richtigen) temporalen Anforderungen ist schwierig. (b) Es mangelt an geeigneten Spezifikationssprachen für Reaktive Systeme, die hinreichend ausdrucksstark und für Praktiker verständlich sind. (c) Ohne ein Umgebungsmodell sind die erzielten formalen Ergebnisse ungenau. Unabhängig davon sind Formale Methoden für automatisierte Produktionssysteme akademisch gut untersucht—hauptsächlich zur Überprüfung der *Safety*-Eigenschaften mit Model-Checker.

Wissenschaftlicher Beitrag

In dieser Doktorarbeit stellen wir das Konzept der (1) *Generalisierten Testtabellen* (GTT) vor. Sie sind eine neue Spezifikationssprache für funktionale Eigen-

schaften von Reaktiven System. Mit ihrer Erweiterung, den (2) *Relationalen Testtabellen* (RTT), können auch relationale Eigenschaften spezifiziert werden. Beide Konzepte bestehen aus der intuitiven (informellen) Beschreibung und sowie der formalen Definition von Syntax und Semantik. Mithilfe von RTTs, haben wir eine neuartige Vertraulichkeitseigentschaft für Reaktive Systeme formalisiert, das beweisbare Vergessen von Information. Für die Regressionsverifikation, eine wichtige relationale Eigenschaft, die die Abwesenheit von neuen Fehlern in der Softwareevolution spezifiziert, konnten wir eine erheblich Leistungsverbesserung erzielen. Diese basiert auf einer (3) neuen Zerlegungsregel, die komplexe Regressionsbeweisziele in kleinere und einfachere Unterbeweise zerlegt. Neben den theoretischen Grundlagen meiner Ansätze entstanden auch Implementierungen, die anhand von realistischen Fallstudien bewertet werden; z. B. die *Pick-and-Place-Unit* der Technischen Universität München.

Generalisierte Testtabellen. Wir präsentieren das Konzept und die logischen Grundlagen von GTTs, eine Spezifikationssprache für Reaktive Systeme. GTTs erweitern das Konzept der (konkreten) Testtabellen, die bereits in der Qualitätssicherung von Software für Speicherprogrammierbare Steuerung (SPS) verwendet werden. Die grundsätzliche Idee besteht in der Verallgemeinerung der Tabelleneinträge. Dadurch beschreibt nun eine Tabelle nicht nur einen einzelnen Testfall, sondern eine Familie von ähnlichen Testfällen. Trotz der neuen Syntax versuchen wir die Selbsterklärbarkeit und Verständlichkeit der konkreten Testtabellen zu bewahren. Besonders im Hinblick auf Ingenieure, die Übung mit der Spezifikation von Testfällen haben, aber eben nicht mit formaler temporaler Spezifikation vertraut sind.

Abbildung 1 zeigt eine typische GTT für die Spezifikation eines Diagnosemoduls. Während des Trainingsmodus lernt dieses Modul einen erwarteten Wertebereich einer Sensorvariablen. Später im Betriebsmodus normalisiert das Modul die gegebenen Sensorwerte und signalisiert einen Fehler, wenn der gegebene Sensorwert zu lange außerhalb des erlernten Wertebereiches liegt. Typischerweise besteht eine GTT aus zwei Seiten: die *linke* Seite mit den Spalten für die Eingangsvariablen und die *rechte* Seite für die Ausgabevariablen. Jede Zelle enthält Einschränkungen, die sich auf die Variable der aktuellen Spalten bezieht. Die Zeilen bilden die Schritte des spezifizierten Test-Protokolls. Sie werden nacheinander von oben nach unten angewendet, wobei eine Zeile oder eine Zeilengruppe übersprungen oder wiederholt werden kann, abhängig von den angegebenen Wiederholungsbedingung (🕒). Wenn mehrere Nachfolgerzeilen möglich sind, verzweigt sich das Test-Protokoll nicht-deterministisch.

In dieser Arbeit definieren wir die formale Syntax und Semantik von GTTs. Die Semantik definiert dabei die Konformität eines Systems bezüglich einer

GTT und basiert auf einem Zwei-Parteien-Spiel über unendliche Worte zwischen dem System und seinen Herausforderern. Dieses Spiel kann über Model-Checking effizient statisch überprüft werden. Wir demonstrieren die Umsetzbarkeit und Anwendbarkeit von GTTs anhand realistischer Beispiele aus der Automatisierungsindustrie. Neben der statischen Überprüfung zeigen wir, wie GTTs auch zur Laufzeitüberprüfung eingesetzt werden können.

Relationale Testtabellen. Relationale Testtabellen (RTTs) sind eine Erweiterung von GTTs. Sie ermöglichen die Spezifikation von relationalen Eigenschaften. Während eine funktionale Eigenschaft nur das erlaubte Verhalten eines Programmlaufes beschreibt, beschreibt eine relationale Eigenschaft das erlaubte Verhalten von und zwischen mehreren Programmläufen. Mit RTTs können wir daher vorhandene Software zur Spezifikation einsetzen. In dem wir, z. B. in der Regressionsverifikation, das Verhalten der neuen Softwarerevision in Beziehung setzen zur vorherigen Revision. Damit stellen wir sicher, dass keine unbeabsichtigten Änderungen oder Verhalten in der neuen Revision eingeführt wurden.

RTTs (vgl. Abbildung 2) erweitern die Syntax von GTTs, um effizient über mehrere Programmläufe zu sprechen. Zunächst müssen die Programmvariablen immer vollständig qualifiziert werden, im Beispiel b»$SFCReset$, d.h. neben dem Namen der Variable $SFCRecet$ wird der zugehörige Programmlauf b mit angegeben. Mittels benutzerdefinierten Funktionen können die Zustände der k Programmläufe auf n-Tupel abgebildet werden, die wir mit geeigneten n-stelligen Prädikate überprüfen können, z. B. die Gleichheit $=(n = 2)$ in Abbildung 2 auf die Abbildung aller der Eingaben **I** oder Ausgaben **O** in den Läufen a und b). Steuerbefehle (▶, ▮▮, ⏮$_r$) erlauben uns die Ausführungen der Programmläufe zu manipulieren. Dadurch können wir simultane Ausführungen aller Programmabläufe aufbrechen, um eben auch relationale Eigenschaften zu spezifizieren, in dem die Programmläufe auch zeitlich auseinander driften.

Die Semantik von RTTs ist durch Reduktion auf GTTs definiert. Dabei werden die Steuerbefehle durch Programmtransformation der ursprünglichen Software abgehandelt. Anschließend wird das k-fache Produktprogramm (der transformierten Programmen) zusammen mit der, bei der Reduktion entstandenen, GTT an die Verifikationspipeline übergeben. Auch hier demonstrieren wir die Anwendbarkeit und Durchführbarkeit der Spezifikation und Verifikation anhand typischer Beispiele aus der Softwareentwicklung.

Modularisierung. Ein weiterer Beitrag ist die Modularisierung für Regressionsverifikation. Über die Regressionsverifikation kann die Abwesenheit von unbeabsichtigter Änderungen zwischen Softwareversionen bewiesen werden.

Für automatisierte Produktionssysteme, die über Jahrzehnte eingesetzt werden, hilft diese Verifikation bereits verdientes Vertrauen in den Anlagenbetrieb auf die nächste Version zu übertragen. Häufig sind die strukturellen und semantischen Unterschiede zwischen den Software-Revisionen eher klein, bspw. werden nur einige lokale Fehler behoben. Der Modularisierungsansatz ermöglicht es uns, die Beweisverpflichtung in kleinere Teilziele zu zerlegen. Dabei nutzen wir die interne Struktur der Software aus.

Das Hauptziel der Modularisierung ist die Verbesserung der Verifikationsdauer. Dies wird hauptsächlich durch die Verwendung einfacher und schneller Algorithmen zur Regressionsverifikation erreicht. Der Einsatz dieser Algorithmen wird erst durch die Modularisierung ermöglicht, da dadurch kleinere Programmfragmente und einfachere Beweisziele entstehen. Der strukturelle und semantische Unterschied in diesen Fragmenten ist i. d. R. gering bis nicht vorhanden. Beispielweise kann die Überprüfung der semantischen Äquivalenz zwischen zwei Programmfragmenten schnell über den Vergleich der Syntaxbäume erfolgen. Dies ist ausreichend für unveränderte (und häufig verwendete) Prozeduren aus vordefinierten Bibliotheken. Die Evaluation zeigt, dass die Modularisierung die Regressionsverifikation von massiger industrieller Software ermöglicht.

Der Hauptunterschied zu gängigen Modularisierungs- oder Abstraktionstechniken für die funktionale Überprüfung besteht darin, dass die Modularisierung kohärent in zwei Programmen erfolgen muss. Für die Kohärenz benötigt mein Ansatz Markierungen an der Modulgrenzen im Quelltext (wobei diese Grenzen benutzerdefiniert automatisch aus der Softwarestruktur abgeleitet werden können). Außerdem ist für jedes Modulpaar aus beiden Programmen ein benutzerdefinierter Regressionsvertrag nötig. Der Vertrag ist dreiteilig: (1) eine Bedingung zur Anwendbarkeit des Vertrages, (2) die angenommene Beziehung der Eingabe und (3) die garantierte Beziehung der Ausgabe jeweils zwischen den beiden Modulen.

Fazit. Alle Beiträge meiner Dissertation verfolgen die Idee, die Hürden für die Anwendung der formalen Verifikation für Reaktive Systemen im Allgemeinen und automatisierten Produktionssystemen im Speziellen zu verringern. Dies erfolgt durch Einführung einer neuen Spezifikationssprache (GTTS), durch Ausnutzung bestehender Programme für die Spezifikation (RTTS, Regressionsverifikation) und durch Verbesserung der Verifikationsperformanz.

Contents

Summary **iii**

Zusammenfassung **vii**

Contents **xi**

1 Introduction **1**
1.1 Contributions 5
1.2 Outline 7
1.3 Previously Published and New Material 8

2 Preliminaries **11**
2.1 Reactive Systems 11
2.2 IEC 61131-3: Software for Automated Production Systems 12
2.3 Model-Checking 15
2.4 Regression Verification 19

3 Related Work **21**
3.1 Functional Verification of PLC Software 21
3.2 Specifications for Reactive Systems 23
3.3 Relational Verification 32

I Generalized Test Tables **37**

4 Towards Generalized Test Tables **39**
4.1 Concrete Test Tables 39
4.2 Generalization of the Syntax 41

4.3 Examples . . . 47
4.4 Semantics: Conformance . . . 52

5 Formalization of Generalized Test Tables 57
5.1 Reactive Systems . . . 57
5.2 Syntactical Representation of Tables . . . 59
5.3 Semantics . . . 65
5.4 Properties . . . 70

6 Decision Procedures 79
6.1 Model-Checking for Conformance . . . 79
6.2 Horn-based Verification via C-program Verifier . . . 86
6.3 Implementation of the Verification Pipeline . . . 94

7 Evaluation 103
7.1 Built-Ins of IEC 61131-3 . . . 103
7.2 Industrial Examples . . . 108
7.3 Plant-Specific Function Blocks . . . 112
7.4 Verification . . . 118

8 Runtime Verification with Generalized Test Tables 123
8.1 Introduction . . . 123
8.2 Monitor Generation . . . 125
8.3 Application Scenarios . . . 132
8.4 Discussion . . . 135
8.5 Related Work . . . 136
8.6 Closing . . . 138

9 Conclusion and Outlook 139
9.1 Weaknesses and Strengths . . . 140
9.2 Meshed Generalized Test Tables . . . 143
9.3 Generalising the Game . . . 148

II Relational Verification 153

10 Relational Test Tables 155
10.1 Syntax . . . 157
10.2 Decision Procedure . . . 163
10.3 Conformance of RTTs . . . 167
10.4 Application Scenarios . . . 167
10.5 Conclusion . . . 174

11 Provably Forgetting of Information **177**
11.1 Confidentiality in Automated Production Systems 177
11.2 Related Work 180
11.3 Forgetting of Information 181
11.4 Experiment 185
11.5 Discussion 190
11.6 Conclusion 192

12 Modular Regression Verification **193**
12.1 Formal Equivalence Relations 194
12.2 Modularization 195
12.3 The Algorithm 203
12.4 Evaluation 210
12.5 Conclusion 213

13 Conclusion **217**
13.1 Summary of the Thesis 217
13.2 Future Work 218
13.3 Follow-up Projects 220

A Glossary **223**

B Source Code **227**
B.1 Function Block `MinMaxWarning` 227
B.2 Function Block `LinRe` 229

List of Figures **233**

Listings **237**

List of Tables **239**

Bibliography **241**

Companion material for this thesis is permanently available under [Wei21].

Chapter 1

Introduction

The software development of automated production system (aPS) is undergoing a radical transformation. This transformation arises the need for new methods for quality assurance of the required control software. In this thesis, we mainly present new languages for the formal specification of functional properties and the specification of the relationship to other programs, with the goal, to provide an accessible and powerful specification and verification for engineers. The need is expressed by the following quotation:

> *Software plays an ever-increasing role in industrial automation.*
>
> Eelco van der Wal
> Managing Director PLCopen
> PLCOpen Newsletter of November 2017

Industrial Revolution Increases Complexity. The Fourth Industrial Revolution (or Industry 4.0) is the current trend in manufacturing systems. This notion encapsulates miscellaneous goals and measurements to enable more productive and more automated manufacturing. The common ground of this revolution is the increased information processing and integration.

For example, a main idea of this revolution is the support for individual products which can be customized by each customer and order. This requires that information flows between the webshop of the producer, via the enterprise resource planning system (ERP) to the control software of the production system. An information flow also exists in the opposite direction, as customers like to

be notified of their order state. In this new scenario, the plant needs to adapt itself to produce the different product combinations. No changeover times or manual interventions should be required. The idea of a flexible and adaptive factory is driven even further, to such a degree, that it is not only possible to automatically produce different combinations of a product, but also to switch the production to completely different products. In this case, the hardware of the production systems will be generic, and only the control software defines the manufactured production.

The industrial revolution offers new possibilities for the producer to be more responsive to and more capable on the market. Under this impression, we can attest, that aPS and their engineering, especially their control software, become increasingly complex, following current trends such as, e. g., increasing customer flavor variety [Vog+15], and increasing system functionalities realized by software [Thr10].

Criticality of Software. The control software is the central player in a production system, and therefore it is an important part of the safety of the entire system. A production system is safe, when (under any circumstance) it does not endanger the human operators or itself. Safety is the most important requirement for a production system, but not the only one. These systems are a central investment for a company, and the irreplaceable option to fulfill the orders efficiently. A standstill or non-correct working manufacturing is a mission-critical risk, which needs to be highly avoided. Considering both aspects, the safety requirement and their criticality, a rigorous validation of the production system is required before operation.

Formal Methods. In academia, the application of formal methods to ensure the reliability of aPS is well-researched (Section 3.1). Formal methods can appear in different shapes, e. g., formal verification and specification, symbolical or model-based testing, model-based design. They have in common that they have a rigorous mathematical background, which allows unambiguous reasoning and validation of the systems. Formal verification offers techniques to prove the reliability of systems in all possible scenarios and circumstances. To achieve this, the formal verification requires a formal description of the system, formal requirements, and rigorous (formal) procedure to reason about the adherence. Note that source code is already a sufficient formal description of the control software if it comes along with a proper interpretation.

Formal Methods in Industry Standards. Moreover, formal methods have been implemented in the standards for functional safety. The IEC 61508 for

"functional safety of electrical/electronic/programmable electronic safety-related systems" mentions semi-formal and formal methods in "Part 3: Software requirements". For example, explicitly stated are Petri-Net, Sequence Diagram (UML) as semi-formal techniques, and symbolical execution [IEC61508, Table B.7, Table B.8]. Additionally, "Part 7: Overview of techniques and measures" defines formal methods similar to our perspective. Moreover, the standard ISO 13849:2015 defines "safety of machinery" and is part of Machinery Directive (Directive 2006/42/EC) of the European Parliament and of the Council. Surprisingly, the institute of the German Social Accident Insurance stated following over this standard in connection to formal methods.

> *Der Einsatz rechnerunterstützter Spezifikationswerkzeuge und formaler Methoden zur Spezifikationserstellung ist möglich, wenngleich unüblich.*
>
> *The use of computer-assisted specification tools and formal methods for the specification creation is possible, although uncommon.*[1]
>
> Institut für Arbeitsschutz der Deutschen Gesetzlichen Unfallversicherung [Hau+17]

Industrial Practice. In today's industrial practice, formal methods have not arrived. The software quality is assured with manually or automatically executed tests, with coding guidelines [Böm+20; RV17; PLC18; PLC16] (Good Automated Manufacturing Practice (GAMP)), and mainly on the development process level of the software project [IEC61508]. It seems that the situation was much worse around 2012: Kormann et al. [KTV12] attest "an unrepresented consideration of testing. Currently, testing is reduced to spare manual developer checks by randomly manipulating parameters of the control software."

The main weakness of traditional testing is that one test case covers only a single, particular run of the control software; many scenarios remain uninvestigated during testing. Full test coverage can rarely be achieved. Systematic testing is a solid utility for detecting typical and expected faults, but unpredictable and rare malfunctions (which also can have severe consequences) are less likely to be discovered. For documentation of the test cases, *test tables* are widely used by the industry [Rös+14]. Each table consists of a sequence of sensor inputs with their expected software responses (table columns), and its rows denote the successive test steps with the specified inputs and outputs for/of the control software. Test tables are commonly written using spreadsheet software and executed within test automation tools like the CODESYS Test Manager.

In contrast to testing, formal verification achieves full coverage and provides mathematical proof of correctness. Also, a benefit of formal verification is that it can be applied early in the software development process, thus, potential faults can be early discovered and removed. As analyzed by Pakonen et al. [Pak+16], one reason for lack of formal methods in aPS domain is that adequate formal specifications are not easily obtained and require a deep understanding of the underlying formal concepts. This makes the application of formal methods often unduly labor-intensive. Also, Ovatman et al. [Ova+16, 10.1 Open Challenges] state that "Specifying Properties to be checked" is an open challenge. In particular, they attest that "temporal logic [..] requires expertise in formal methods and mathematical modeling area" and they also have foreseen our test tables ("conversion from a tabular format to extract specifications in LTL or CTL").

Summary and Look Ahead. Although the industry standards allow and promote the use of (semi-)formal methods, they have not reached the daily practice of the aPS software development yet. With our contributions (Section 1.1), we want to bring formal methods into the software development of automated productions. First, by the introduction of a new specification language, which is derived from a currently used language for describing test cases. Second, by the extension of our specification language (and verification) for *relational properties* of control software. Traditionally, functional correctness, e. g., the adherence to safety requirements, is a property of a single program run. In contrast, the relational properties are able to express relations between multiple program runs. These program runs can be defined by the execution of different programs or the same program. Thus, using relational properties, we are able to use (existing) software for the specification, and minimize the size formal specification—leaving the application engineer in their accustomed environment. Two prominent applications of relational verification are the *regression verification* (proving conditional program equivalence), and *non-interference* (proving the absence of information flow). For the regression verification, we propose a novel modularization approach, which enables the comparison of large control software. Also, we use our relational specification to formalize a novel security property to identify systems which eventually forget secret information. With relational verification, we refer to the verification of relational properties.

Scope of this Thesis. We limit the scope of this thesis to the domain of automated production systems, their prevailing programming language, and their computer systems – programmable logic controllers (PLC). This is mainly motivated by the accessibility to case studies and software. But our considerations and contributions are directly applicable to a wider range of systems, known

as *reactive systems*. We characterize them by two facts: They are periodically executed and operate in a feedback loop with a physical environment. We fully introduce them in Section 2.1 (informally) and Section 5.1 (formally). From our theoretical perspective, we do not distinguish the reactive system and the PLC and use these terms synonymously.

1.1 Contributions

In this doctoral thesis we present the following contributions:

Generalized Test Tables To lower the threshold of applying formal verification, we propose generalized test tables (GTTs), a novel formal specification language which is derived from the existing concrete test tables.

By using the industrial used concrete test tables, we hope to transfer their comprehensibility and intuitiveness into a formal specification, which gives an increased validation coverage. The degree of generalization can be individually decided by the engineer by using (or not using) the feature of GTTs. This allows for gradual progress from specification-by-example (starting with a concrete test table) to a fully systematic specification. GTTs support static and dynamic verification.

Relational Test Tables Using relational properties, we can specify requirements by using (existing) software, and so decrease the size formal specification. Also, relational properties cover a wide range of important requirements, like secure information flow. The verification of relational properties are hardly accessible to engineers.

We introduce *relational* test tables (RTTs) – an extension of GTTs for the specification of relational properties. Relational test tables support the specification of relational properties between $k \geq 2$ program runs (also known as k-hypersafety properties). This is the first (dedicated) specification language for relational properties of reactive systems.

Forgetting Information During the manufacturing process, confidential information is generated and aggregated that constitute business secrets; therefore they are part of the attacker's focus and require rigid protection. Hence, it is a valuable target to prove the absence of business secrets.

For this, we present a novel notion of information forgetting in reactive systems. This is a relational property describing that a reactive system forgets the specified information within a certain amount of execution cycles. This property limits the amount of historical information an attacker

can learn by observing a manufacturing system. Information forgetting is a relational property formalized upon RTT.

Modular Regression Verification Regression verification, a relational property, helps to prevent the introduction of unintended, faulty behavior during the software evolution, and to transfer the earned trust of operationally well-tried systems to the next revision [Cha+19].

We present a novel approach for modular regression verification proofs for reactive systems based on the idea of relational regression verification contracts. The approach allows the decomposition of a larger regression verification proof into smaller proofs on its sub-components. We embed the decomposition rule in a new algorithm for regression verification, which orchestrates several light- and heavyweight techniques.

Each contribution is evaluated to show its practicability and feasibility. For this an additional contribution has arisen besides this thesis: our verification library VERIFAPS.

Verification Library We contribute VERIFAPS, an open source library for the verification of aPS software after IEC 61131-3 standard (Section 2.2).[2] The library provides the basic functionality for parsing and transformation of the programming languages, syntactical analyses, accessibility to logical models, symbolical execution engines, and support for the communication with verification backends. Also, all our presented approaches are implemented with this library, and are available in the same repository.

The contributions of this thesis follow the idea of lowering the obstacle of verifying the dependability of reactive systems in general, and automated production systems in particular for the engineer by introducing a new specification language (GTTS), exploiting existing programs for the specification, or improving the verification performance.

The presented work was mainly done within the IMPROVE APS project (as part of DFG Priority Program 1593 –"Design for Future"), in cooperation with Suhyun Cha, Dr. Sebastian Ulewicz and Prof. Birgit Vogel-Heuser of the Technical University Munich (TUM). Whereas this thesis concentrates on the formal foundations, the verification tooling, and their feasibility of the presented approaches, the companion doctoral thesis of Suhyun Cha focus on the automation engineering perspectives, including the application concept (under consideration of mechanical requirements), use cases, and empirical evaluations of GTTS and regression verification.

[2] https://github.com/VerifAPS/verifaps-lib

1.2 Outline

This thesis is split into two parts. The first part (Part I) is dedicated to the functional verification with GTTs and the second part (Part II) to the relational verification of reactive systems.

Before we dive into our contributions, the Chapter 2 presents preliminary work, which is required to understand content chapters. In particular, we give an informal definition of reactive systems (Section 2.1), the programming languages of aPS defined by IEC 61131-3 (Section 2.2), an overview on Model-Checking for Linear Temporal Logic and IC3 in Section 2.3, and our previous work on regression verification (Section 2.4).

The Chapter 3 gives the combined general related work on the functional verification of aPS, and the relational verification. Additionally, related works are also given in Sections 8.5 and 11.2 if they are too specific.

Part I. In the first part we introduce the GTTs. We start with an informal engineering perspective in Chapter 4 in which we try to give a phenomenological view upon GTTs, covering the syntax and semantics and giving examples. This chapter targets engineers which want to understand and use GTTs. The following chapters are more formal. Chapter 5 presents the mathematical foundations of GTTs, including the formal definition of reactive programs and systems, the syntax and mathematical structure of GTTs (Section 5.2) and also their formal game-based semantics. We distinguish between the weak, strict, and cooperative conformance (Section 5.3). At the end (Section 5.4), we state and prove properties of GTTs. The formal foundation is exploited in Chapter 6 to build two decision procedures using model-checker (Section 6.1) and Horn-based C-program verifier (Section 6.2). Both procedures decide whether a given program is valid in respect to a GTT. Our approach is evaluated in Chapter 7 by the specification and verification of examples from the IEC 61131-3 standard, industrial practice, and a demonstrator plant of the TUM. These chapters focus on the static verification of GTTs. In Chapter 8, we investigate the dynamic verification of GTTs by generating runtime monitors. The generated monitors close the gap between the static verification and the operation, e. g., checking of observance of the verification assumption. We close our considerations on GTTs in Chapter 9 with the discussion of strengths and weakness of GTTs (Section 9.1), more powerful "meshed GTTs" (Section 9.2), and a generalization of the underlying two-party game semantics (Section 9.3) to allow other specification notions than GTTs.

Part II. Chapter 10 presents the RTTs as an extension of GTTs. Therefore, we focus on the difference to GTTs. Moreover, this chapter also includes the decision

procedure for RTTs (by reduction to GTTs), and four realistic application scenarios. The RTTs are also used to define the property of forgetting information in Chapter 11, which motivates, explains, and formalizes this property for reactive systems. Also, a large part (Section 11.4) is dedicated to the application of this property on a demonstrator provided by the Fraunhofer IOSB and developed by a third contractor. The decomposition rule for the regression verification is presented in Chapter 12, along with a new algorithm for regression verification and the evaluation.

1.3 Previously Published and New Material

Parts of this thesis were published in similar or different shapes. We dedicate this section to clarify the origin of the chapters.

Part I. GTTs were covered in [Cha+18b; Wei+17; Bec+17; Bec+19]. These publications are included and extended in the Chapters 4 to 6. In particular, Weigl et al. [Wei+17] gives an informal view on GTTs and provides the linear interpolation example in Section 7.2.1. In the companion paper [Bec+17], we presented the syntactical and semantical formal foundation (evaluation of expression, mathematical structure two-party game, etc.) and also the "min-max" example in Section 7.2.2. With [Cha+18b], most of these parts became deprecated by the introduction of GTT new features: row groups, strong repetition and state variables. Also, Section 7.3 origins from [Cha+18b]. In this thesis, we have revisited, extended and updated the texts and the mathematical definitions. Especially, the following (larger) points are new and unpublished in the Chapters 4 to 6.

- Two separate informal explanations of the GTT semantics (Chapter 4).
- A new (additional) conformance definition: the *cooperative* conformance (Chapter 5).
- A new decision procedure exploiting C-program verifier. This idea was originally investigated in a bachelor thesis [Wie20]. For this doctoral thesis, the presentation is completely rewritten, the evaluation is extended to our examples, and we also use SeaHorn [Gur+15].
- The evaluation was completed repeated (Chapter 7). Also, new cases are added from the standard IEC 61131-3, and an additional table for "min-max" example (Section 7.2.2).

Note that Beckert et al. [Bec+19] is a revisited presentation of the material in [Bec+17] and [Bec+15].

The Chapter 8 bases on the recently accepted paper [Wei+21]. We updated this chapter to be aligned to the prevalent notions in this thesis and added a new section on the implementation (Section 8.2.4).

The last chapter in this part, Chapter 9, is completely unpublished bases on personal notes.

Part II. The presented work regarding the area of relational verification consists of three chapters which origins from three publications.

Weigl et al. [Wei+20] is the source of Chapter 10. The texts are revisited and adapted to fit into this thesis. These adaptations include the use of the common notions of this thesis, and the removal of GTT explanations in favor of Part I. Also, the idea of projection function in the table column (Section 4.2.2.2) is backported to GTTs.

Chapter 11 is not peer-reviewed published. A technical report [Wei19] exists with a stronger focus on the verification process and the software. This chapter is rewritten to bring more attention to the novel property of the forgetting of information. Note that this work was already publicly presented at the annual meeting of the FoMSESS 2019.[3]

Chapter 12 was previously published in [WUL20]. The idea is further research of our master thesis [Wei15] and was later topic the master thesis of Daniel Lentzsch [Len18]. In contrast to [WUL20] and besides notion adaptions, this chapter receives a larger explanation for the "conformance by symbolic execution" (including the algorithm) in Section 12.3.2, more detail on the soundness, and we elaborate the use RTTs and multiple contracts.

Also, there are papers related to topics of this thesis but not covered. In [Cha+19], the authors explain how and when trust in an aPS is preserved from the old revision to the new revision of the system. And Cha et al. [Cha+18a] proposes a simple specification inference of GTTs from Sequential Function Charts. The idea is to use the generated GTTs as regression test cases for the new revision.

[3] FoMSESS is a special interest group under the umbrella of the German Informatics Society and dedicated to Formal Methods for the Software Engineering, Safety and Security.

Chapter 2

Preliminaries

2.1 Reactive Systems

We already mentioned the term "reactive systems". Now, we want to introduce it in more detail. For our section, we use [Hal98]. Reactive systems are characterized by the two facts:

- They are designated to run forever.
- They need to react to their environment.

The first fact distinguishes reactive systems from simple batch programs. A batch program starts with an input and terminates with an output. In contrast, a reactive system starts and continuously interacts with its environment. Both facts are also suitable for *interactive systems*. In comparison to reactive systems, interactive systems can synchronize with their environment [Hal98], i. e., blocking any further progress in it. For example, communicating systems are typically interactive. As long as a communication party does not emit the next message, the other parties (environment) pauses and wait for it. A reactive system cannot block or pause its environment, because its environment consists of physical processes, which run unblockable and in parallel with the system.

We can distinguish between two operation modes for reactive systems: *event-driven* and *sampling*. In event-driven mode, the system waits for certain events to occur, which triggers the program execution. In sampling mode, the system periodically (1) reads the input values, (2) executes the program, (3) writes output values, and (4) waits for the next cycle. This procedure is also known as the "scan cycle" in the terminology of aPS. For our thesis, we focus on the sampling mode.

Usually, software for reactive systems is limited in the used (or allowed) programmatic constructs because they have to ensure real-time guarantees with deterministic runtimes. The scan cycle has to be finished before the next cycle starts. Halbwachs [Hal98] already pointed out that reactive systems are "[..] intended to be deterministic" and "[..] are submitted to critical reliability requirements". The formal definition follows in Section 5.1. Note that in this thesis we only consider deterministic reactive systems and software.

2.2 IEC 61131-3: Software for Automated Production Systems

In this section, we clarify the notions of the IEC 61131-3 standard [IEC61131-3], which defines the software and programming languages of aPS.

The techniques in this thesis are applicable to all kinds of reactive software systems. However, we put a special focus on Programmable Logic Controllers (PLC). PLCs are computing units that are used to drive and control automated production systems. Thus, they are reactive real-time systems, and are usually in operation for a long time. Due to the massive amount of I/O, and the real-time and safety requirements, PLCs are specialized computing components that are programmed differently from standard PC hardware. In a PLC, the program code (logic) is repeatedly executed once every few milliseconds. The constant time between two runs is called the *cycle time*. There are also different execution modes for PLCs (event-driven, continuous, ...) that we do not consider in this thesis.

Program Organization Units. A family of five programming languages for PLCs is defined in the standard IEC 61131-3 [IEC61131-3]. These programming languages are the body of Programming Organization Units (POUs). POUs are the concept of structuring a PLC software into multiple reusable components. A POU can occur in different shapes as a program, a function block, a function, an interface, and a class. Interfaces and classes are introduced in the latest version of IEC 61131-3 to allow object-oriented programming. In this thesis, we concentrate on the first three POU kinds. Functions are callable stateless procedures, which take input arguments and compute a return value. Functions can be called as a single statement or inside an expression. In contrast, a function block can have state variables and multiple output variables. A function block needs to be instantiated (similar to classes in object-oriented languages) inside any other stateful POU (function block, program). The number of instances is not limited. A function block offers only one function which can be invoked on

the state of the instance. As it has multiple output variables, an instance of a function block cannot be invoked inside an expression. Programs are similar to function blocks, as they are also stateful. But in contrast, programs cannot be instantiated inside the PLC software, instead, they are instantiated from the PLC system during the bootstrapping of the application. Programs are the entry point for the PLC to invoke the logic. The execution (frequency and priority) is defined in a configuration description. You can consider programs as singleton function blocks. Also, we consider the input and output variables of the programs as the incoming sensor and outgoing actuator signals.[1] Additionally, to the state of the defined programs, we can define global variables, which can be accessed from every POU. In this thesis, the PLC software in the examples and experiments has one program and no global variables. We use the notion "PLC program" to refer to the complete software and not the program POU.

All considered POUs (program, function block, function) are defined by two parts: the signature and the implementation body. The signature defines the variable with their names, datatype, and category (input, output, local, etc.). The datatypes are either built-in (various integers, float double, strings, time, datetime, etc.) or user-defined (structures, arrays, etc.). The implementation body is written in one of the five languages: Structured Text, Sequential Function Chart, Function Block Diagram, Ladder Diagram, and Instruction List. We give a brief overview:

Structured Text Structured Text (ST) is a textual imperative programming language, similar Pascal. It has the typical statement constructs: loops (for, while, repeat), selection (if, case), limited jumps (exit, return, continue), and assignments.

Sequential Function Chart Sequential Function Chart (SFC) are a graphical programming language, which is derived from Grafcet (EN 60848) and Petri-Net. In the core, an SFC is an automaton with steps (states) and transitions between the steps. A transition has a guard, and can connect multiple predecessors with multiple successors, to allow a fork and join similar to Petri-Net. With each step, we associate a list of actions, which are triggered when the step is active. The special feature is that these actions can be triggered depending on a given qualifier. For example, the timed qualifier `D T#1s` triggers an action after the step was active for one second.

Function Block Diagram Function Block Diagram (FBD) are graphical language. An FBD is a list of networks. A network is a graph consisting of

[1] Normally, this is given in a configuration description, which describes a mapping between the state space and the I/O bus.

computation nodes (operators, functions, function block instances) and variable dependency (transitions). A network is executed by invoking the computation nodes in topological order defined by the variable dependency. This representation allows capturing the information flow from the input and state variables to the output variables easily. A seldom-used feature is the goto-node, which allows jumping between graphs inside an FBD.

Ladder Diagram Ladder Diagram (LD) is the third graphical language, which emulates an electrical circuit. An LD consists out of multiple networks. Each network consists out of circuit diagrams (coils, contacts, and connecting lines), which are limited by the left and right vertical power lines. The evaluation of an LD is a simulation of the current flow from left to right, where connection lines and coils always pass the flow and contacts only let it pass if their associated variable is true. Additionally, coils store the incoming flow into a variable.

Instruction List Instruction List (IL) is a textual assembler-like language. An IL body consists of a sequence of commands, which are evaluated against an accumulation register. Conditional and unconditional jumps exist. This language is marked as deprecated and will be vanished in the future version of the standard.

In this thesis, we concentrate on Structured Text. You find examples for Structured Text throughout this thesis (Listings 4.4, 4.6, 7.1 and 7.3 and Appendices B.1 and B.2). In previous work [Wei15] we gave a deeper introduction and formal semantics to Structured Text. A formal definition for SFCs is given by Bauer et al. [Bau+04b], and a former Instruction List semantics is in [Wan+13]. A reduction of SFCs to ST is given by bachelor thesis [Gor19], which we supervised. In Section 6.3.1, we introduce ST_0 a simplified version of ST and explain how we handle the languages in our verification pipeline. Note that there are vendor-specifics extensions to the IEC 61131-3 languages, e. g., goto-statement in ST, pointer with pointer arithmetic, conditional assignments. We do not consider these extensions in this thesis.

Restrictions. While the languages are Turing-complete, PLC programs hardly ever contain general while-loops. If they contain loops, they have a known fixed upper bound on the number of iterations since PLC code has to meet strict real-time conditions. For the same reason, recursion is avoided. Additionally, the state space of PLC programs needs to be bounded at compile time because the complete memory is allocated during the bootstrapping of the PLC program

which makes them more predictable. This forbids dynamic memory allocation and also recursive datatypes (a record cannot have a variable of its datatype). Both limitations make the correctness theoretical decidable.

2.3 Model-Checking

Model-Checking is the discipline of evaluating that a given property P holds in a given model $\mathcal{M}$, often written as $\mathcal{M} \models P$.

Let AP be a finite set of automatic propositions. The atomic propositions represent the value of program variables. We use Kripke structure as the model of our systems (cf. [GGS21]):

Definition 2.1 (Kripke structure)**.** *A Kripke structure $K = (S, I, R, L)$ is a 4-tuple, where 1. S is a finite set of states, 2. $I \subseteq S$ is the set of the initial states, 3. $R \subseteq S \times S$ is a total transition relation between states, and 4. $L \colon S \to 2^{AP}$ is the labeling function.*

A Kripke structure is a graph in which the states S are connected via R. The label function L determines the variable assignment in each state. We say a proposition p is true in state $s \in S$ if and only if $p \in L(s)$.

For the verification of linear temporal logic (LTL) properties, we are interested in the linear infinite paths starting at an initial states. A path $p = s_0, s_1, s_2, \ldots$ over a Kripke structure K is an infinite sequence of states ($s_i \in S \wedge (s_i, s_{i+1}) \in R \wedge s_0 \in I$). Normally, we interest for the value assignments along a path: a trace $t = L(s_0), L(s_1), L(s_2), \ldots$ is infinite sequence of the corresponding labels for a given path p. Hence, a trace t is a word in 2^{AP^ω}. With $L(K) \subseteq 2^{AW^\omega}$ we denote the set of all possible traces over a Kripke structure K. The model-checking problem for LTL is stated as follows:

Definition 2.2 (Model-Checking of LTL)**.** *An LTL property $P \in Fml_{LTL}^{AP}$ holds in a Kripke structure K if and only if the property holds on all traces over K:*

$$K \models P \textit{ iff } \forall t \in L(K).\, t \models P \ .$$

The set of all LTL Fml_{LTL}^{AP} is defined in Definition 2.3.

From Büchi [Büc90], we know that model-checking of LTL is decidable by reduction to the emptiness problem of Büchi automata, but this suffers under state explosion as explicit automata for the system and the LTL formula are constructed. In modern tools, more advanced techniques are used, either by the reduction from LTL to Computation Tree Logic (CTL) [CGH97], which can also be checked symbolically [Bur+92]. Or by translation to an invariant [Bra+11; HBS12] and using IC3 (and derivatives).

Linear Temporal Logic (LTL). LTL model-checking depends on the evaluation of the LTL property P on each trace. The set Fml_{LTL}^{AP} denotes the set of LTL property over the propositions AP. We define similar to [BK08, Definition 5.1.]:

Definition 2.3 (LTL formulas). *The set Fml_{LTL}^{AP} of LTL formulas is defined as*

- $\texttt{true} \in Fml_{LTL}^{AP}$ *and* $a \in Fml_{LTL}^{AP}$ *(for each* $a \in AP$*) are LTL formulas.*
- *Let* $\phi_1, \phi_2 \in Fml_{LTL}^{AP}$*, then*

$$\begin{aligned} \phi_1 \wedge \phi_2 &\in Fml_{LTL}^{AP} \\ \mathbf{X}\phi_1 &\in Fml_{LTL}^{AP} \\ \phi_1 \ \mathbf{U} \ \phi_2 &\in Fml_{LTL}^{AP} \\ \neg\phi_1 &\in Fml_{LTL}^{AP} \end{aligned}$$

This is a minimal definition for LTL, consisting of two temporal operators until (**U**) and next (**X**), negation, and the conjunction. By DeMorgan's law, we also have `false` and the disjunction. Additionally, we define the two operations finally and globally:

$$\mathbf{F}\phi \equiv true \ \mathbf{U} \ \phi \qquad\qquad \mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$$

The semantics define when given a trace t at a starting position $n \in \mathbb{N}$ satisfy an LTL formula ϕ, denote as $t, n \models \phi$.

Definition 2.4 (LTL semantics). *Let* $t = s_1, \ldots$ *be a trace over* 2^{AP} *and* $\phi \in Fml_{LTL}^{AP}$*, then satisfaction of* $t, n \models \phi$ *is defined as*

$$\begin{aligned} &t, n \models \texttt{true} && \\ &t, n \models a && \textit{iff } a \in s_n \\ &t, n \models \phi_1 \wedge \phi_2 && \textit{iff } t, n \models \phi_1 \textit{ and } t, n \models \phi_2 \\ &t, n \models \neg\phi_1 && \textit{iff } t, n \not\models \phi_1 \\ &t, n \models \mathbf{X}\phi_1 && \textit{iff } t, n+1 \not\models \phi_1 \\ &t, n \models \phi_1 \ \mathbf{U} \ \phi_2 && \textit{iff } \exists j \geq n.\ (\forall n \leq i < j.\ t, i \models \phi_1) \textit{ and } t, j \models \phi_2 \end{aligned}$$

Note that the default starting is zero, hence $t \models \phi \equiv t, 0 \models \phi$. From the semantics, we see the next-operator ($\mathbf{X}\phi$) is satisfied if ϕ is satisfied in the successor state $(n + 1)$. The until-operator (**U**) requires an upcoming point in

time (j) in which ϕ_2 is satisfied, and before j, each point (i) satisfies ϕ_1. For globally- and finally-operator following semantics are derived:

$$\begin{aligned} t, n &\models \mathbf{F}\phi_1 && \text{iff } \exists j \geq n.\ t, j \models \phi_1 \\ t, n &\models \mathbf{G}\phi_1 && \text{iff } \forall j \geq n.\ t, j \models \phi_1 \quad , \end{aligned}$$

which are rather simpler than the until-operator. Finally-operator states that a ϕ_1 must be satisfied eventually in the future, and globally-operator requires ϕ_1 is satisfied in all upcoming states ($j \geq n$). For our formalization, we use the combination of both $\mathbf{GF}\phi$, which can be interpreted as ϕ is "infinitely often" satisfied.

IC3. Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3) is a relatively new technique for checking invariants in a Kripke structure. An invariant is a state formula which needs to be valid in all reachable states of a system. Coming from LTL, an invariant is describable $\mathbf{G}\phi$, where ϕ is formula free of any temporal operator. Bradley and Manna [BM07] firstly formulated this technique, which later targeted in various publications, e. g., [SB11; Bra12; Bra11].

We reuse our Kripke notions: Let $I \subseteq S$ be the initial states, and $R \subseteq S \times S$ the transition relation between states. Also, let $P \subseteq S$ be our property, given as the set of states which fulfills the given invariant ϕ

$$P := \{s \mid s \in S \wedge s \models \phi\} \quad .$$

Note that normally these sets are described by CNF formulas. For our explanation, we use the unusually (but simpler) notion with explicit sets.

The idea IC3 is to find a new inductive invariant $Inv \subseteq S$ with following properties:

- the inductive invariant is stronger than our property:

  $$\forall s \in S.\ s \in Inv \rightarrow s \in P$$

- the inductive invariant is indeed inductive:

  $$\forall s, s' \in S. s \in Inv \wedge (s, s') \in R \rightarrow \wedge s' \in Inv \quad , \text{and}$$

- it is valid for the initial state:

  $$\forall s \in I. s \in Inv$$

If such an inductive invariant Inv is found, we have shown that our property ϕ holds in all reachable states.

Bradley [Bra12] gives a detailed explanation. The algorithms are in [Bra11]. Here, we give a brief explanation of the procedure, and try to abstract the sophisticated SAT encodings. For more details refer to the publications.

In the core, IC3 is a forward search (similar to bounded model checker) and a backward propagation of occurring counter-examples. Counter-examples can be spurious because IC3 uses internally an over-approximation of the reachable states. This approximation is strengthened by the spurious counter-examples. In the remaining section, we use $s \in S$ and $s' \in S$ as universal quantified variables in each formula.

IC3 starts with two simple checks to ensure that the property is valid in the initial states, and the states reached after one step, formally, $s \in I \rightarrow s \in P$ and $s \in I \wedge (s, s') \in R \rightarrow s' \in P$. Afterward, it starts with the forward search. Let $F_k \subseteq S$ be a set of states, s.t. all states reachable within k transitions are (at least) included in F_k. Hence, F_0 are the initial states ($F_0 = I$). F_1 is set to all states which adhere to the property. Note that the initial definition of F_1 ("$F_1 = P$") is the over-approximation. The forward search extends F_1 to F_2 by applying one transition step:

$$F_2 := F_1 \cup \{s' \mid s \in F_1 \wedge (s, s') \in R\}$$

Now, we need to check if we have reached states in F_2, which violates our property. If $F_2 \setminus P = \emptyset$, then P is the searched inductive invariant, and we terminate. Otherwise, we need to check each violating state $s \in F_2 \wedge s \notin P$, whether it is spurious or indeed reachable (and therefore a valid counter-example). Reachability needs to be checked by applying R reversely, for F_2:

$$\exists s, s', s'' \in S.\ s'' \in F_2 \wedge s'' \notin P \wedge (s', s'') \in R \wedge (s, s') \in R \wedge s \in I$$

If no such s exists, we can subtract s' from F_1 as s' is not reachable in one step from F_0 and part of the over-approximation. Note that we know that s'' is reachable via s'. In general, for arbitrary k, the counter-example s'' is back-propagated until F_0. If it is spurious, then there exists $k' < k$ for which the transition cannot be applied reversely. Then, all $F_{k''}$ ($k' \leq k'' \leq k$) are strengthened by subtracting the corresponding falsely assumed reachable state. If F_1 became strong enough (no counter-examples are reached in F_2 anymore), there are two options: On $F_k = F_{k+1}$, IC3 terminates and the inductive invariant is the state-formula which describes the set F_k. Or $F_k \neq F_{k+1}$, then we proceed with the forward search by expanding F_{k+1} to F_{k+2}.

It may be surprising, that F_k becomes the inductive invariant. But this is just a consequence of the k-induction: A formula, which describes all reachable

states, is inductive on the transition relation. If $F_k = F_{k+1}$, we reached an over-approximation of the reachable states F_k, s.t. no new states can be reached with one additional transition application ($k + 1$), and by construction every state $s \in F_k$ adheres the property P. IC3 always terminates for finite state spaces, as there exists a maximal amount steps in which all states are reachable.

In implementations, the sets F_i are encoded using a set of clauses s.t. their models are the set of included states. A huge performance impact has the generalization of the counter-examples. In our explanation, we only back-propagate one state at a time. Due to a symbolical encoding, it is also possible to propagate a set of violating states. These violating states are found by generalization in the last (violating) transition. One simple generalization is to start with a concrete counter-example, given as a model from the underlying SAT-solver, and drop literals from this model as long as all states (described by reduced model) violate the property.

2.4 Regression Verification

Beckert et al. [Bec+15] applies regression verification to PLC software and is the base for our work of RTTs and modular regression verification.

Motivation. Throughout their lifetime, systems have to adapt to new situations (bug fixes, hardware replacements, new function requirements, etc.) and many system changes will also incorporate changes in the software. Each software modification potentially introduces incorrect behavior as a side effect. To avoid the effect, *regression tests* are widely used in the industry as they yield good results and can easily be extended to the new functionality of the software. However, software testing cannot guarantee correct behavior since there will always be scenarios that are not covered by the test suite. Functional verification can help to overcome this problem: A formal specification describes the expected behavior of the software and a verification system analyzes whether the specification holds in all possible scenarios. But, the specification must be user-provided and is, in most cases, not trivial to find, especially when developers are less experienced with formal specification. For the verification in an evolutionary environment, two specifications are required: one for the existing software revision, and one for the new revision.

Definitions. In regression verification, instead of using two specifications for two revisions, both revisions are compared directly to each other: The old software revision serves as a functional specification for the new one. However,

the old revision can only *partially* specify the new revision since only those scenarios (input sequences) where the behavior should not change can be checked for equivalence. The input sequences, for which the behavior has been intentionally changed, need to be verified separately using functional verification or testing, as identified in [Ule+16b]. This functional verification of the behavioral difference between the old and new system is, therefore, called *delta verification*.

Regression verification does not necessarily imply the software behaves correctly for all inputs, it rather says that the software has the same behavior as the previous revision, including all potentially undiscovered errors. Regression verification transfers "trust" (of correct functioning) of the old software to the new software. Therefore, the confidence and trust must be experienced or earned by the older software during its operation, validation, or verification. This topic is detailed elaborated in [Cha+19].

Beckert et al. [Bec+15] identifies three different kinds of equivalences between PLC software. We present the simplest and most complex formalization here.

First, the *perfect equivalence* states that under equal input both systems produce equal output. This is a typical use case when the software changes are only refactoring steps. Note that perfect equivalence is similar to bisimulation. Bisimulation [San09] states that both system K, K' (given as a Kripke structure) simulates each other: $K \sim K'$. This implies, that the languages of both structures are equal $L(K) = L(K')$. Hence, there exists no Büchi automaton which can distinguish between both structures K and K'. Whereas the bisimulation considers the complete state, the perfect equivalence only claims the equivalence for input and output variables, thus, does not enforce the equivalence for state variables. Also, for deterministic systems, the bisimulation coincides with the trace equivalent [San09]. And later formulation is similar to perfect equivalent modulo the state variables.

Second, the conditional equivalence extends the perfect equivalence in two ways: (a) the equivalence of input and output are replaced by user-defined relations, and (b) a predicate filtering out traces for which the relation is not enforced. Conditional equivalence only permits a relation over old and new states in the same cycles—relating states of different time points is not possible. Also, there exists only one relation overall time and all traces of the old and new software. For the evolution scenario of the Pick-and-Place Unit (Section 7.3) in [Bec+15], conditional equivalence was sufficient. Nonetheless, we develop RTTs which allow far more complex relational (and regression) properties.

Chapter 3

Related Work

We use this chapter to present the related work to our thesis. In contrast to the preliminaries, the related work follows goals similar to our thesis' contribution. We identified three different research fields: Section 3.1 reports previous attempts on formal verification for PLC software. In Section 3.2, we report on previous attempts for new specification languages for PLC and reactive systems. Finally, we discover the world of relational verification in Section 3.3. Also, related work which is very individual to a specific topic is given in Sections 8.5 and 11.2.

3.1 Functional Verification of PLC Software

In a broader sense, we follow with GTT the goal of functional verification. The contribution is mainly a new formal specification language, but we see the necessity to give a current overview about the state of functional verification of PLC software.

This research domain is already the subject of three surveys [Lam+99; YF03; Ova+16]. We focus on the most recent survey [Ova+16] from 2016, and extend it later with more recent publications. Ovatman et al. [Ova+16] presents a meta-study and gives interesting insights: the used verification techniques, the programming languages of the case studies, the temporal logic, and the kind of properties. For the verification techniques, they identified three categories: Timed Automaton (UPPAAL and KRONOS), symbolical (SMV-family) and explicit-state model-checker (e. g., Spin). Timed Automaton is especially useful to model real-time behavior and timer function blocks (Section 6.3.1) in IEC 61131-3, e. g., [Bau+04a]. But the most prominent techniques are the symbolical model checking ([Ova+16, Table 1]), and also the trend in the application of model-

Table 3.1: "Number of studies related to different programming languages in 5-year periods From: An overview of model checking practices on verification of PLC software" [Ova+16, Table 13]

PLC prog.	Textual	LD	SFC	FBD	Petri net	Other
−1999	1	3	1	0	1	5
2000−2004	4	3	10	0	3	3
2005−2009	4	3	0	3	2	8
2010−	0	2	0	5	2	3

checking.[1] Across the different case studies, invariants are the most prominent specification kind (used in 90% of the referenced case studies), followed by safety (81%) and liveness (43%). Moreover, in most of the case studies, CTL or TCTL (Timed CTL for Timed Automata) was used (80%), the remaining studies used LTL. The Table 3.1 shows the usage of the programming language (used in the case studies) over the years. In comparison, the input for our decision procedures is textual (i. e., Structured Text), which can be generated from a Sequential Function Chart, Function Block Diagram or Instruction List.

It seems, that [Ova+16] misses some approaches. For example, Wan et al. [Wan+09] presents a formalization of timers in the interactive theorem prover Coq. Their specialty is the analysis of timings for instructions within a scan cycle. For this, they use a control flow graph with additional annotated time delays to present Ladder Diagrams. Also, Coq, was exploited by [BB11] for the verification of PLC systems.

In the recent development, we identify two platforms for PLC verification: *PLCverif* [DFB15] developed at CERN and *Arcade.PLC* [BBK12] in RWTH Aachen. Both verification platforms have a similar structure: the PLC program is parsed into a (logical) intermediate format, which is the base for optimization and reasoning and can also be exported into different formats for model-checking.

PLCverif is presented in [DMV16b] with the extension for the programming languages of Instruction List, Function Block Diagram, and Ladder Diagram, better scalability, and equivalence checking. The programming languages are handling by the reduction to the *STr*, a dialect of Structured Text, where the implicit register of Instruction List is made explicit with program variables. Their case study contains 120,000 STr instructions generated from 9500 Instruction List instructions. Due to the reduction method of the intermediate representation (automata-based), the final size is 29 locations (from original 123,346), and

[1] In [Ova+16]: "It can be seen that in industry current trend in the application of model checking PLC programs is using model-based development tools on symbolic model checkers like NuSMV."

366 bits (from 3105 bits). The requirements of the case study were compiled from interviews, and expressed either as CTL properties or in their behavior specification-based approach PLCverif. More details on the used specification approach *PLCspecif* are below (and in Figure 3.2). The verification took 53 minutes for all 25 requirements.

Bohlender and Kowalewski [BK20] (and with their earlier publications [BHK18; BK18]) exploits state-of-the-art solvers for *constraints horn clauses* (CHC). CHC are a restricted first-order logic where only Horn clauses are allowed (clauses with at most one positive predicate). Horn clauses can be efficiently generated from programs (via weakest-precondition [Bjø+15]), and solved efficiently with SMT-based solvers. Such solvers allow using theories for linear integers, real arithmetic, or arrays.

In [BHK18], the authors provide an SMT-based bounded model-checker with the specialty of *dynamic large-block encoding*. Bounded model-checking is a bug finding technique, where prefixes up to a length k are investigated for safety properties. The dynamic large-block encoding describes the symbolic execution technique, which encodes a complete scan cycle as a single state transition.[2] The dynamic aspect states, that only feasible paths of the current cycle are encoded into the formula. Therefore, the symbolic execution starts with a specified context of variable assignments. The approach of [BHK18] outruns its competitors on the verification of invariants.

Bohlender and Kowalewski [BK20] propose compositional reasoning for PLC software along with *mode abstraction*. The first one allows replacing a call of function block instance with an over-approximate summary of the called body. Candidates for the summary are computed via the mode abstraction, which is a static analysis that computes an abstract transition relation, similar to predicate abstraction, where the state spaces are partitioned by the predicates evaluation (on the states) and the new abstract transition relation is derived from the equivalence classes. Mode abstraction is similar, but a mode variable is given, on which the predicates should are inferred.

3.2 Specifications for Reactive Systems

This thesis is not the first initiative to build a proper specification language for PLC programs, or reactive systems. We start the specific PLC approaches, followed by more generic approaches to reactive systems or temporal logics. We compare the related work with GTTS. Note that we anticipate the content of

[2] In contrast, in [DMV16b] a single instruction is a transition. The end of scan cycles is indicated with a special variable.

the upcoming chapters. Hence, not all comparisons might be understandable without further reading of Part I.

PLC Specifications. Darvas et al. [DBM15] presents a specification approach *PLCspecif* specific to PLC. An example of their approach is presented in Figure 3.2 showing the different parts: the signature (input and output variables and definitions), defined events (transition guards), a state machine, output functions, and invariants on the internal state and signature. Note that the specification is operational by defining an executable Mealy automaton.[3] The conformance is defined via equivalence of the output variables between the specification and the verification subject—similar to perfect equivalence (Section 3.3). This specification format is similar to the used automaton-based specification used in [Tec06], and the format of the defined output functions are similar to the Parnas tables [PMI94]. The output is determined by evaluating the conditions on the column headers and searching for the matching table row.

In contrast, GTTs allow defining the observational behavior without writing a concrete state and transitions. Due to their nondeterminism by row-selection and global variables, GTTs describe a family of allowed behaviors, e. g., "the output can be 1 or 2", where this approach only allows checking against a concrete value.

In [DMV16a], this specification is further discussed, and includes a comparison between programs. Therefore, we discuss this in the last section with the specification of the relational verification. In [DVM16], they show to synthesize programs using their specification approach. Due to the used encoding as a Mealy automaton, the program synthesis is straightforward.

Ljungkrantz et al. [Lju+10] introduce a new dialect of LTL: *ST-LTL*. All (future) temporal operators ($\mathbf{X}$, $\mathbf{G}$, $\mathbf{F}$, and $\mathbf{U}$), and typical logical combinators of LTL are present. The changes are:

- The usage of Structured Text syntax for the operators. This also allows to express the ST-LTL formulas as Ladder Diagrams or Function Block Diagrams.
- Access the previous value of a variable via the variable suffix `_previous`.
- Detection of rising and falling edges on variables (suffix: `_risingEdge` and `_fallingEdge`).
- Access to input and output value of "inout" variables (variable suffix `_in`, and `_out`).

[3] A Mealy automaton is categorized that the output depends on the current state and the inputs.

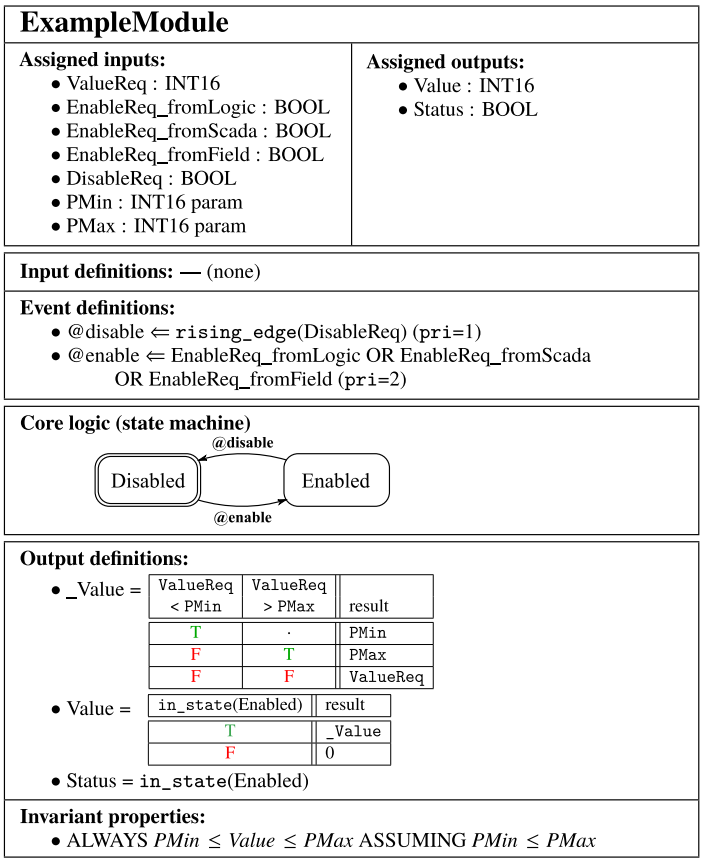

Figure 3.2: Example specification from [DBM15, Figure 3] describing a Mealy automaton.

Let us consider an example of ST-LTL from [Lju+10]:

```
1 Spec6 := ALWAYS( Run_risingEdge ONLY_IF
2    NOT EStop & NOT Error & Reset_risingEdge );
```

First, we notice that the LTL operator have full names, and negation is expressed as `NOT`, like in Structured Text. "`ONLY_IF`" is the logical implication. This specification is an invariant, stating that for every state: When `Run` has a rising edge (`NOT Run_previous AND Run`), `EStop` and `Error` are false, and `Reset` has a rising edge, too.

The extensions of [Lju+10] tries to make LTL more appealing to engineers

by using Structured Text notions and adding abbreviation for previous value and edge detection, but LTL's problem remains. LTL formulas become incomprehensible with an increasing amount of nested temporal operators.

In GTTs, we also have an easy access to previous variable values (Section 4.2.1). The edge detection can be modeled by using the projection function (Section 4.2.2.2).

Later, Ljungkrantz et al. [Lju+12] "proposes a systematic work procedure that can be used as a first step of developing formal specifications of safety PLC programs in industry" reusing the specification in [Lju+10]. The procedure is only applicable to safety requirements in PLC programs.

Xiong et al. [Xio+20] propose a user-friendly verification approach which bases upon specification-mining of LTL properties and invariants. The idea is that a tool analyses a PLC program and generates (useful) properties, which are then reviewed by the engineers. Internally, this approach uses the specification miner *Taxado* [LPB15]. The specification miner requires three inputs: a set of finite traces (which are generated from test cases), templates of LTL property, and a threshold (a confidence value and a support value). In [Xio+20] following patterns were supplied

$$\mathbf{F}(p) \qquad \mathbf{G}(!p) \qquad \mathbf{G}(x \rightarrow \mathbf{X}y)$$
$$\mathbf{G}(x \rightarrow \mathbf{G}\neg y) \qquad \mathbf{G}(x \rightarrow \mathbf{XF}y) \qquad \mathbf{F}y \rightarrow (\neg y \ \mathbf{U} \ x)$$

This idea founds on the assumption, that the engineers can understand generated specifications. Xiong et al. [Xio+20] state the properties for two examples: a toggle flip-flop (one input and one output variable) with four two-states invariants ($\mathbf{G}(x \rightarrow \mathbf{X}y)$), and six invariants for a two digit-sorter (two inputs and two outputs).

Bitsch [Bit01] follows a similar target of decreasing the obstacles of existing temporal logics. Instead of generating LTL properties, they provide a catalog and categorization of patterns. The catalog provides specification patterns in different specification languages LTL, CTL, and μ-calculus along with an explanation in natural language.

Kormann et al. [KTV12] proposes a fragment of UML sequence diagrams for the specification of test cases for PLC programs. This fragment is a well-defined executable subset, enriched by new constructs, e. g., state invariants (that are checked during the tests), timing information on executions, special messages. A sequence diagram is compiled into a testable executable program, which calls and supervises the specified module. Such a sequence diagram is a bit more powerful than a concrete test table. A sequence diagram describes multiple concrete test protocols. The variation arises due to the flexibility of the given timing constraints. Many of the sequence diagram features are not exploited or forbidden, e. g., multiple lifelines or asynchronous messages, by [KTV12].

Reactive Systems. We focus on a much broader spectrum: specification languages of reactive systems.

Software Cost Reduction (SCR) [Hei+05; Hei+98] is a formal requirements method, that was applied to mission-critical systems by NASA [HJ07]. SCR uses synchronous state machines to describe the behavior of a system. State machine specifications use a "user-friendly" table-based notation for the transition relation and the output relation. SCR provides various tools for the simulation and validation of specifications, the generation of system invariants and source code, and the formal verification of application properties. There are similarities to the approach of [DBM15].

CocoSpec [Cha+16] is a specification language for reactive programs that are written in the Lustre programming language. Similar to GTTS, CocoSpec is based on an *assume-guarantee* paradigm using constraints on input values (assumptions) and output values (assertions) in every time step. The constraints are Boolean expressions following the semantics of Lustre. Using a state variables, assumptions and assertions become time-dependent in CocoSpec. In GTTS, in contrast, assumptions and assertions are always time-dependent, i. e., they depend on the table-rows.

The example in Listing 3.3 shows a Lustre program with CocoSpec from [Cha+16]. The first module (node) is used to delay the input values. If $x_1, x_2, x_3, \ldots$ are the incoming values for x, `previous` emits the values delay by one time step $0, x_1, x_2, x_3, \ldots$. The module `stopwatch` is a simple stop watch which increments the `count` when `running` is true. The variable `running` can be toggled, and the `count` can be reset. This module is coupled with the contract. A CocoSpec contract is similar to a module, in this example, it has inputs (e. g., `tgl` and `rst`), and outputs (`c`). Internal helper variables are allowed (`on`). The contract has global assumptions (`assume`) and guarantees (`guarantee`), that must always hold, or they are always given. Additionally, the contract has modes. Every mode is a tuple of two lists: additional assumptions (`require`) and guarantees (`ensure`).

CocoSpec semantics is defined by reduction to LTL. A module satisfies a contract if $\mathbf{G}A \rightarrow \mathbf{G}G'$ is valid, where A stands for the global assumptions and $G' = G \cup \{R_i \rightarrow E_i\}$ refer to the combined guarantees (global guarantees G and the ith requires clauses implies the ith ensures clauses). A contract is well-defined if and only for every state there is at least one active mode [Cha+16, Eq. 1].

Timing diagrams were analyzed for the specification around the millennium. Fisler [Fis99] formalizes timing diagrams and investigate the theoretical properties. A timing diagram is a three tuple $\langle P, N, O \rangle$ where P is a ordered sequence of time points, function N maps (variable) names to *waveforms* at specific time points $p \in P$, and O a ternary relation capturing temporal order of events, syn-

```
1  node previous ( x : int ) returns ( y : int )
2  let
3    y = 0 -> pre x;
4  tel
5
6  node stopwatch ( toggle, reset : bool ) returns ( time : int );
7  (*@contract import stopwatchSpec(toggle, reset )
8           returns (time); *)
9    var running : bool;
10 let
11   running = (false -> pre running) <> toggle;
12   count = if reset then 0
13   else if running then previous(count) + 1
14   else previous(count);
15 tel
16
17 contract stopwatchSpec ( tgl, rst : bool ) returns ( c : int );
18 let
19   var on: bool = tgl -> (pre on and not tgl)
20                       or (not pre on and tgl);
21   assume not (rst and tgl); guarantee c >= 0;
22   mode resetting ( require rst; ensure c = 0; );
23   mode running ( require not rst; require on;
24                  ensure c = (1 -> pre c + 1); );
25   mode stopped ( require not rst; require not on;
26                  ensure c = (0 -> pre c); );
27 tel
```

Listing 3.3: Example specification of a stop watch in CocoSpec.

chronization and time bounds. Allowed waveforms are: H (high level), L (low level), F (falling edge), R (rising edge). Considering the example in Figure 3.4, P contains the time points $p_1, \ldots, p_6$, N contains for each variable a, b, c the waveform for each $p_1, \ldots, p_6$, e. g., $N(a, p_1) = R$ and $N(b, p_5) = L$, O contains the relation of the events, e. g., "$= n$" states that a rising edge on a is followed by a falling edge within n time units ($((a, p_1), (a, p_2), [n, n]) \in O$) , and the double bar denotes rising edges on a and c at the same time ($((a, p_5), (c, p_5), [0, 0]) \in O$). Note that the time points P are generated by the tool and are not part of the user input. Intuitively, a trace satisfies a timing diagram, if we can assign each time point $p \in P$ to a position of the trace, s.t. the constraints (waveforms and time constraints) are met.

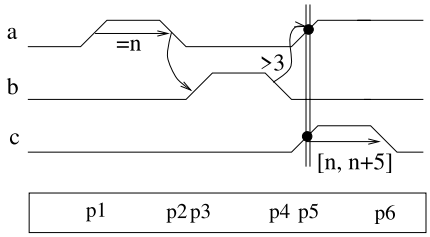

Figure 3.4: An example of a timing diagram from [Fis99, Figure 1.].

Fisler [Fis99] states that timing diagrams are incomparable to LTL, some LTL formulas are not expressible. On the other hand, timing diagrams can express some non-regular words. Timing-diagrams are decidable by the reduction to deterministic counter machines (in particular 1-2DCM).

To some extent, timing diagrams are similar to GTTs. In both specification notions, we describe the expected value at certain time points. In GTTs, the specification is dense: the time points are successive and compact, whereas in timing diagrams the time points are sparse: Between the specified time points arbitrary time can pass depending on the time bounds on the relations. Moreover, in GTT we decide between assumptions and assertions.

Vyatkin and Hanisch [VH01] investigate timing diagrams for the use in control engineering for the verification of distributed controllers (after standard IEC 61499, extension of IEC 61131). Their notion of timing diagrams distinguishes between input variables and output variables. The specification of input variables is translated into finite-state models, and the specification of output variables is translated into extended CTL formula. The finite-state models describe the valid input sequence, for which the CTL formula needs to be adhered to by the system. Their notions of timing diagrams are similar to the one of [Fis99]: A timing diagram is the asynchronous product of specified waveforms for the signals, which is restricted by timing bounds between these events. A timing diagram specifies a finite behavior. They propose different semantics, e. g., "there exists specified behavior in the original system", or "all specified scenarios exists in the system model". The "conditional existence" semantics is similar to the GTT semantics, as it states if a behavior of the system matches the input side, the behavior needs to match the complete timing diagram. Our semantics for GTT is quite more complex, mainly induced by the variable length of the described words of a GTT. Thus, the GTT notion needs to be robust on prefixes of the system behavior.

Schlör et al. [SJW98] uses (RT)STD, an interpretation of timing diagrams, for the specification in an industrial context. Their semantics are defined in [FJ97]. In contrast to [Fis99], they see the waveform as a sequence of expression,

allowing to specify allowed values symbolical, instead of a fixed set of waveforms. In additional, they have weak and strong time constraints between time points. Strong time constraints are needed to be fulfilled by the system, and weak constraints are assumptions on the environment. A timing diagram has two modes: *initial* denotes the start of the timing diagram in the first state, and *invariant* timing diagram should be satisfied again and again. The semantics are defined by the construction via timed Büchi automaton and timed propositional logic.

In timing diagrams, the time points of interest for which the specification defines the expected values (waveforms) of the variables, are finite and fixed. The flexibility in timing diagrams is in the timing constraints between these time points. In the example (Figure 3.4), the variable n is instantiated depending on a given word and can be used in multiple time bounds. If the time between rising and falling of a takes n units, rising and falling on c takes at least n units up to $n + 5$ units later. Such constellations are not expressible in GTTs, because our time constraints are rigid intervals. On the other hand, GTTs are more flexible on the specification of the signals. Timing diagrams only allow a fixed range of specified waveforms, limiting them to Boolean variables, or state formulas [SJW98]. GTTs allow arbitrary constraints, which also can depend on the previous value of variables, which can be asserted at (finite or infinite) arbitrary many points in time.

To summarize, GTT describe finite or infinite behaviors with the flexibility on the data constraints, whereas timing diagrams describe finite behaviors and are flexible on the timings between events.

Temporal Logic for Specification. We present three "general-purpose" temporal logics (FTL, ITL, and GTL) which aims provide a better specification experience than LTL or CTL.

The ForSpec Temporal Logic (FTL) is an extension of LTL developed at Intel [Arm+02]. In addition to LTL operators (next, until, always, eventually), it supports the corresponding past operators (yesterday, since, historically, once). Moreover, FTL adds some features that are of interest w.r.t. GTTs. For example, FTL supports the specification of time windows, in which certain events need to occur (bounded LTL operators). FTL allows the description of *regular events*, which are sets of finite state sequences described by regular expressions.

The description of regular events follows the syntax of regular expression from the automata theory, e. g., α^* is the zero-or-more repetition of α; and α, β the concatenation of α and β. Instead of alphabet symbols, FTL uses state formulas. For example $(send, (\neg ack)^*, send)$ describes a sequence, in which there is ack between two send, in which the formula $send$ represents the set

of all states in which $send$ is true. Such regular events can be combined with temporal operators.

Similar to the regular events is *Interval Temporal Logic* (ITL) of [Mos85], designed for the hardware specification. The notion in ITL speaks of intervals, which are (finite or infinite) sub-words of traces. ITL has two temporal operators $f\,;g$ (chop) and f^*. The chop-operator $f\,;g$ is satisfied on an interval, if the interval can be split into two intervals, such that the first interval satisfies f, and the second satisfies g. f^* is satisfied on an interval if the interval can be split into a finite number of subintervals, and each subinterval satisfies f. The special predicate **skip** is fulfilled on unit intervals.

It should be obvious that ITL mimics the operator on regular expressions in formal languages: We have the sequential composition (chop-operator), the union (disjunction), and the repetition (f^*). Therefore, ITL is ω-regular [Mos85].

GTTs are similar to ITL if we consider the mathematical structure (Definition 5.7). Either the assumption or the assertion can be directly rewritten in ITL using the chop-operator and repetition to match the time constraint. But encoding both sides of the GTTs into ITL requires more effort. Let us consider a simple GTT with $\langle(\phi_1, \psi_1), (\phi_2, \psi_2)\rangle$, with two rows. A table row is an implication: if the assumption $\phi_i$is fulfilled, then the (corresponding) assertion ψ_i needs to be adhered. Moreover, if the assumption is violated, the test is aborted. Our little example can easily encoding into ITL:

$$\phi_1 \rightarrow (\psi_1 \wedge \mathbf{skip}\,;\mathbf{skip} \wedge (\phi_2 \rightarrow \psi_2)\,;true^*$$

If the ϕ_1 is fulfilled, ψ_1 needs to adhere in a unit interval, and further the second row is satisfied in the next interval. Behind this coding, we assume the local interpretation of ITL, where the interpretation of variables depends only on the first state in an interval. Normally, the interpretation considers the complete interval. In our encoding scheme, we need to take care of multiple successor rows and row (group) repetitions. If we consider these, we see the ITL capturing a GTT can be become exponential in the size of the table rows. To reproduce this, consider a table where all rows except the first row are skippable. Encoding the first row requires that we encode all rows (except the first one) on the right side of the implication; encoding the second row requires all rows (except the first and second rows), and so on. The size of the encoding is defined in the recursion formula $T(n) = \sum_i^{n-1} T(i) + 1$ in the amount of implications for n table rows, which is bounded exponential.

In contrast to ITL, GTTs offers a simpler specification w.r.t. to the assumption and assertion pairs, and provide a more structured view of variable constraints and repetitions.

Dillon et al. [Dil+94] proposes graphical language for interval logic (GIL). A specification consists out of multiple drawn intervals and searches. The

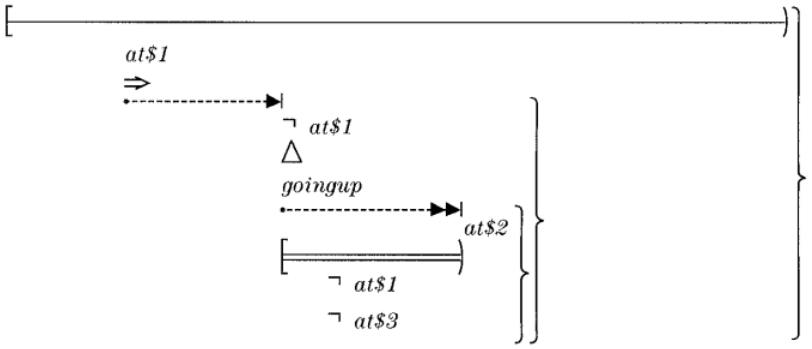

Figure 3.5: A specification for an elevator: $at\$i$ is true if the elevator is at the ith floor.

alignment of the graphical elements determines the time dependency and their meaning. Figure 3.5 presents a specification for an elevator which specifies: If the elevator is on the first floor and goes up, it arrives the second without visiting any other floors. The first graphical notion is the interval over the complete time. Under this interval, a formula is given, which starts with an implication (assuming the elevator is on the first floor $at\$1$). The position of this formula marks it as an invariant over the drawn interval. Vertical formulas are conjunctively connected. The braces help to determine which formulas or constructs belong together. The arrows are *search* operators. The first arrow in the conclusion searches for the time point in which "$\neg at\$1 \Delta goingup$ holds. The point-operator $\alpha \Delta \omega$ is similar to a reverse until-operator. It is fulfilled if there exists a time point in which α holds and ω holds for each following time point (until the end of the interval). The next-arrow (with the double arrow) indicates a strong-search indicating that a failing search is an error. This arrow specifies the second floor is eventually reached. The interval below this arrow applies an invariant during the search, that the elevator is not on the first or third floor. In the example, we have expressed that whenever the elevator is on the first floor, then there exists a time point in which the elevator is not on the first floor anymore and it is going upwards, thus it must finally reach the second floor without reaching the first or third floor.

3.3 Relational Verification

Specification. Darvas et al. [DMV16a] carries a little bit further the idea of the equivalence between specification and implementation in *PLCspecif* [DBM15] and also consider equivalence checks between two implementations or two specifications. Moreover, they introduce two new notions of equivalence besides

their strict equivalence (similar to the perfect equivalence of [Bec+15]). The equivalence notions allow a time delay between the two compared artifacts (specification or implementation). The first notion requires a constant delay, the second one has a (constant) time window in which the variables are compared for equivalence. Darvas et al. [DMV16a] considers in-perfect timers in the PLC program which can vary by one millisecond. Hence, the same programs are not required to equivalent in their definition. This nondeterminism is motivated by industrial practice in which the cycles times are not always adhered to. This time modeling permits the detection of critical timings, that may resolve into different behavior on real-world hardware. These equivalence notions are expressed as CTL formulas to be model-checked including helper models for capturing the previous values.

Comparing to RTT, we can cover their specification of strict and (constant and variable) delayed equivalence completely. The delayed equivalence can be expressed with the past-references (for the variable delay we need to disjunctively enumerate the possible equivalence in the specific time window). Moreover, RTT permits to express arbitrary large delays by using PAUSE-command—only the events, when such a delay starts and stops, need to be expressible in the table. Currently, we have not considered non-perfect timers, because our use cases are mainly driven by software evolution. During the software evolution, we want to find unintended behavioral changes introduced by the engineer and not by the PLC system. Of course, these models can be applied for RTT-based verification, but require an adaption of the underlying models for the timer function blocks.

Clarkson et al. [Cla+14] propose an extension of LTL and CTL* for the specification of hyperproperties on a Kripke structure. Both temporal logics are extended by existential and universal quantification over traces. We use the notion "program run" synonymously in our thesis. All propositions are explicitly denoted to a particular trace. In comparison, due to the existential quantification, HyperLTL and HyperCTL* allow the specification of a superset of our notions of relational properties. For example, they can express the refinement relationship: for every trace of the old system, exists a trace in the new system. Note that hyperproperties are defined on a single Kripke structure, but due to the construction of product structure, this limitation can be lifted.

In a recent publication, Goudsmid et al. [GGS21] introduce the notion of multi-properties and MultiLTL. The main difference, regarding to HyperLTL, is that a multi-property is evaluated against a multi-model. Such a multi-model is a tuple of (different) Kripke structures. MultiLTL extends LTL with universal and existential quantifiers $\forall^i, \exists^i$ over the traces of the Kripke structures, where i denotes the corresponding Kripke structure. Let us consider an example given in [GGS21]. Let C be the model of the client and S to be the model of the server, then we specify that every request from the client is finally received by the

server with:

$$\langle C, S\rangle \models \forall^C \pi_1 . \forall^S \pi_2 .\ \mathbf{G}(r\_sent_{\pi_1} \rightarrow \mathbf{F} r\_received_{\pi_2}) \ .$$

Note that $\langle C, S\rangle$ is a multi-model, and the right side is a multi-property. This property can also be specified with GTTs, due to the use of universal quantifiers. The model-checking problem with MultiLTL is reducible to HyperLTL in polynomial time. From the engineering perspective, they inherit the problems from LTL and CTL* for the practical use (cf. [Pak+16]).

Barthe et al. [Bar+19] uses first-order logic (FOL) to express hyperproperties on traces. The FOL universe consists of the integer and traces. Also, the signature includes symbols and predicates denoting specific time-points, last iterations of loops, and program variables. Note that FOL is more powerful than LTL, but only covers star-free fragment of the ω-regular languages.

There are several extensions for specifications approaches in the deductive verification domain. Yi et al. [Yi+15] propose change contracts as an extension to the Java Modeling Language (JML) which allows the description of the behavioral as well as structural changes between two methods. In principle, such a contract expresses a relation between input parameters and output parameters of the old and new version, such like *"whenever* $in > 0$ *holds,* $out' == out + 1$ for in, in' the old and new input parameter, and out, out' the old and new return value. Syntactical, a change contract adds new clauses to the method specifications. Consider the template of [Yi+15]:

```
1 /*@ changed_behavior
2   @ when_required φ; when_ensured ψ; when_signaled (T1x) θ;
3   @ requires φ'; ensures ψ'; signals (T2x) θ';
4   @*/
```

for predicates $\phi, \phi', \theta, \theta', \psi, \psi'$ and types T_1, T_2. The new `when`-clauses are the requires (pre-condition), ensures (post-condition), and signals (post-condition in exceptional case) of the previous version. Additionally, we can use `prev`-operator to access to values of the previous version. A change contract is evaluated against the parallel execution of the old and new versions of a method. It is fulfilled when the old version fulfills the `when`-clauses, and the new version fulfills the default JML-clauses. The relation between the execution of the old and new version needs to be established by using `prev`-operator. A special feature of these change contracts is the handling of object equality, which is defined by the isomorphism of the underlying graph structure. Also in the JML domain,

Scheben and Schmitt [SS14] presents a JML extension for the specification of secure information flow.

Blatter et al. [Bla+17] presents an extension for ASCL, a specification language for C programs. ASCL is similar to JML and describes the pre- and post-condition of methods. This ASCL extension introduces a specification clause for relational properties, including `\call` operator, which is used to refer to the result of function invocations (program runs in our notion). In contrast to change contracts, this notion is quite flexible, and can express various relations to other program runs, e. g., monotonicity, equivalence, or secure information flow.

Both JML and Frama-C extensions are not directly applicable for reactive systems; they target for a relational specification of the pre- and post-state of methods, rather than on infinite traces over time.

Modular Regression Verification. Modularization for regression verification is covered in [GS13] which serves as a second basis to our work in Chapter 12. Godlin and Strichman [GS13], who also coined the term "regression verification" exploit both regression verification and decomposition to prove the equivalence between similar programs. They can handle programs with recursive function calls and unbounded loops, both paradigms are not common in software for reactive systems. Nevertheless, their work does not cover our topic completely: They only consider functions that do not have an internal state and require them to be perfectly equivalent. Therefore, they do not need any regression specification, but they also require a mapping that determines function pairs of the old and new programs, which are factored out simultaneously. Moreover, the decomposition in [GS13] works bottom-up if possible. Our approach work from top to bottom.

The work on differential assertion checking [Lah+13] modularizes relational proofs similarly to the one presented in this paper. They employ *mutual function summaries* to abstract two related function blocks, such a summary consists of relative pre- and post-condition over the input and outputs of both functions. A similar extended approach is presented in [EMH18] for faster reasoning of k-safety hyperproperties. Eilers et al. [EMH18] allow hyperproperties in procedure specifications which can later be applied in the reasoning on the k product program. The hyperproperties are tuples, written as $\hat{P} \Rightarrow \hat{Q}$ where $\hat{P}$ is a relational pre-condition and $\hat{Q}$ the relational post-condition. Each relational condition talks about the input- and output variables of the k program runs. For example, $f : x_1 = x_2 \Rightarrow y_1 = y_2$ expresses that the procedure f is deterministic—equal input x results into equal output $y = f(x)$. For the specification, the authors just use first-order logic. Such a contract can be applied during the

reasoning if all program runs reach the execution of a procedure. Essentially, both approaches ([Lah+13] and [EMH18]) are the same concept as our regression verification contracts (Section 12.1). They do not target reactive systems but individual single function invocations, and use the intermediate verification language Boogie ([Lah+13]) or the theorem prover Vyper ([EMH18]) to encode their conditions rather than a model checking verification backend.

Goudsmid et al. [GGS21] provides sound and complete compositional proof rules for MultiLTL properties (in multi-models). A multi-model is a list of Kripke structures, which are universal or existential quantified in the given MultiLTL property (see above). These proof rules allow using an abstraction rather than the original Kripke structure. Universal quantified structures are replaced with overapproximations and existentially quantified structures with underapproximative models. In contrast to our modularization, our problem and abstraction differ. First, we do not abstract a single model at once, instead, we abstract the used subroutine in two models in parallel. Both models are always overapproximations. Second, we have contracts which determine the used abstraction for the subroutine, and subroutines are verified against the used contracts separately. According to [GGS21], any approximation of a model would be fine as long as the required property could be proofed under the approximation.

The goal of Guthmann et al. [GST16] is similar to ours: Modularizing the equivalence proof. For matched procedures, two partial sets are computed. One set contains input values where the procedures behave equivalently and one set where the output differs. Both sets are approximated. The approximation is strengthened over the execution time of the algorithm. They extended their approach work with a demand-based refinement of the approximated sets in [TGK17]. This approach is not applicable to reactive systems, as the notion of internal states is missing. Also, only perfect equivalence is currently covered.

Part I

Generalized Test Tables

Chapter 4

Towards Generalized Test Tables

Complex industrial control software often drives safety-critical systems, like automated production plants or control units embedded into devices in automotive systems. Such controllers have in common that they are *reactive systems*, i. e., that they periodically read sensor stimuli, cyclically execute a defined program, and emit the computed signals to the actuators.

Usually, in practice, the correctness of implementations of reactive systems *is not* verified using formal techniques. What *is* used instead in industrial practice today is testing, where individual test cases are used to check the reactive system under test [RV17]. Test cases are commonly written in the form of (concrete) *test tables*, in which each row contains the input stimuli for one cycle and the expected response of the reactive system. Thus, the whole table captures the intended behavior of the system (the sequence of actuator signals) for one particular sequence of input signals.

In this chapter, we start the journey of the generalization of concrete test tables towards our notion of *Generalized Test Tables* (GTTs). We keep this chapter rather informal and on a phenomenological level. The formalization follows in Chapter 5.

4.1 Concrete Test Tables

Concrete test tables (CTTs) are an industry-wide used description format to describe test protocols in the automation industry. The described test protocol consists of a sequence of input values (provided by the environment) and the

#	INPUT A	INPUT B	OUTPUT X	OUTPUT Z	DURATION
1	1	1	0	5	1
2	0	3	6	5	7
3	1	4	2	5	2

Figure 4.1: Example for a CTT with two input variables, two output variables covering 10 cycles of the system.

sequence of expected output values (computed by the reactive software under test). Despite their frequent use, the CTTs are not formalized or standardized. On the one hand, this gives us more freedom in our formalization of GTTs. On the other hand, we have no anchor point to adjust our notion of GTTs against. Thus, we have to consider carefully our GTT formalization against the intuition of an application engineer (or at least an intuition that is reasonably justifiable). Because the CTTs are an instantiation of the GTTs, we automatically gain a formalization of CTTs.

The test protocol, described by a CTT, has two dimensions: data and time. The data dimension is expressed by the columns, as every column is dedicated to an input and output variable of the reactive system under test. The columns for the input variables contain the given input stimuli to the system, and the columns for the output variables contain the expected output values.

The rows in the table represent the time dimension as they form the consecutive steps of the test protocol. Each table row symbolizes possible steps in the test protocol. Instead of repeating a row several times, the number of repetitions can be annotated in the special table column DURATION (🕒). In a concrete table, the cells contain concrete values. Hence, a CTT describes only one specific test case. All variables in the example are of type integer; in general, other types, such as Boolean or enum variables, are possible.

Example 4.1. *Figure 4.1 shows an example for a CTT. The table has two input variables (A, B), two output variables (X, Z), and describes a test case of 10 cycles (as the durations of the three rows add up to 10). In this example, all variables are of type integer; whereas in general, other types, such as Boolean variables, are also possible.*

We consider periodically executed reactive systems, where each execution cycle is one step in the test protocol. A cycle consumes a fixed period of time, the *cycle time*. In each cycle, the concrete input values contained in the table row corresponding to that step are the stimuli for the system, and the system is expected to react with the output values contained in the same row. If the

observed system response differs from the expectation for at least one row of the test table, then the system violates the test protocol. The value of row duration determines how long the system has to remain in the step, i.e., how often the row is to be repeated. The row duration is given as a number of cycles (it can also be given as a real-time value, which is transformed into cycles by division with the system's specific cycle time). A table row with a duration of n is equivalent to repeating that same row n times with a duration of 1.

There is no restriction on the data types of variables and their values that can be used in the tables. The only requirement for CTTs is that they are compatible with the used operators. In the following, we use the simple data types of the IEC 61131-3, e. g., Boolean, integer, and enumerations.

4.2 Generalization of the Syntax

GTTs follow the same principles as CTTs but go beyond them by introducing three means to abstract of concrete values: (1) abstraction using constraint expressions, (2) using references to other cells in constraint expressions, and (3) using generalization in the duration column.

As a consequence, a GTT describes a (possibly infinite) set of concrete behaviors. A GTT usually does not fully specify a system and covers only its behavior for a certain scenario or situation. The characteristics of concrete test tables that we deem essential and that are preserved in GTTs are:

1. Every signal/actuator cycle corresponds to one row in the test table.
2. An implicit sequential traversing of the table rows.
3. Every row formalizes a local implication of the form: If the signal values adhere to the input constraint, then the actuator signals adhere to the output constraint.

Example 4.2. *Figure 4.2 shows an example of a GTT, incorporating the generalization concepts described above. Note that the concrete table depicted in Figure 4.1 is one of the possible instances of the generalized test table given in Figure 4.2, achieved by instantiating the global variable p with the value 3.*

In the remaining of this section, we explain in detail the generalization of the parts of GTTs: table cells, the table columns, and row repetitions.

4.2.1 Cell Expressions

The table cells are in the focus of GTTs. Whereas a CTT describes only one test protocol with concrete values, a GTT is able to describe a family of CTTs by

replacing the concrete values inside the table cells with Boolean expressions. The Boolean expressions are built up with the usual logical ($\wedge, \vee$ etc.), arithmetical ($+, *$, etc.), and comparison ($<, \geq$ etc.) operators over the input, state, and output variables. The concrete expression syntax can be aligned to the programming language of the test subjects. In our case, we allow the expression fragment of the Structured Text programming language. We also align the evaluation of the operator to the used programming language.

References to Other Cells. A reactive system may possess an internal state, i. e., its behavior may depend not only on the current but also on previously observed input values. GTTs have two expressive means to formulate such dependencies: global variables and past references.

Global variables can be used in constraints. We use lower-case identifiers to denote global variables. They have an arbitrary value that does not change throughout the application of the test protocol. For example, if v is a global variable, $A = v$ occurs in one cell, and $X = v + 1$ in another cell, then both constraints require that $X = A + 1$. While the value of a global variable v does not change in a single application, there may still vary between different applications of the same test tables. Note that using $X = v$ in a cell for X is equivalent to "don't care" if this is the only occurrence of v.

Past references are relative references to previous values of variables. A past reference "$X[-n]$" refers to the value of variable X which the variable had $n \in \mathbb{N}$ cycles before the current one. A past reference refers to the system iteration n cycles ago, not to the nth row above the current row (this may differ because rows may be repeated). Absolute references to particular cells can be expressed using global variables.

Abbreviations. In alignment with the typical spreadsheet application, GTTs support several abbreviations for better readability and ease of use (Table 4.3). These abbreviations are expanded with the designated program variable or projection of the corresponding column into Boolean expressions. First, a literal or variable n denotes the equality of the designated column variable with the

#	INPUT A	INPUT B	OUTPUT X	OUTPUT Z	⏲		
1	1	1	0	–	1	⊤ – ⊥	⊤ ≥ 1 ⊥
2	–	p	$=2*p$	$\geq Z[-1]$	$[0,7]$		
3	–	$=p+1$	$[0,p]$	$2*Z>X$	2		

Figure 4.2: Example for a generalized test table with a global variable p.

Table 4.3: Constraint abbreviations from [Bec+17]. X is the name of the variable that the cell corresponds to; n, m are arbitrary expressions of type integer; α, β are abbreviations or formulas.

Abbrev.	Constraint
n	$X = n$
$-$	$X = X$ (don't care)
$< n$	$X < n$ (same for $>, \leq, \geq, \neq$)
$[m, n]$	$X \geq m \wedge X \leq n$
P	$P(X)$
α, β	$\alpha \wedge \beta$

value of n. In the following, let X be the designated column variable. Then the literal "2" is expanded to $X = 2$, "t" becomes $X = true$, and "Y" becomes $X = Y$. Note that this syntax is ambiguous. The cell content "`true`" could be interpreted as a Boolean expression, or as the given abbreviation. We decide on the interpretation as an abbreviation for two reasons. First, if we want to specify that a variable is free from any constraints, we should use "—" ("don't care"). Second, if the cell content "`false`" would be interpreted as the Boolean expression, it would represent an unsatisfiable constraint. We expect the need for such constraints to be unlikely. By specifying a comparison operator the equality can be overwritten, thus > 5 becomes $X > 5$. We allow the default comparison operators for numbers, i. e., for $<, >, \leq, \geq, \neq$. With the interval literal $[n, m]$, we can set a lower and upper to the column variable directly, e. g., "$[n - 1, n + 2]$" constrains X to be $n - 1 \leq X \leq n + 1$. It is also possible to use user-defined predicates. These predicates are functions (returning a Boolean value) that are defined in a programming language[1], allowing to express complex constraints. These predicates have one argument which takes the value of the program variable in the column header. Later we extend this designation to a projection function that returns multiple values . Then the predicate arity needs to match the number of returned values. For example, let $P(\cdot)$ be a predicate, then it is expanded to $P(X)$. Abbreviations can be conjunctively combined using a comma ",". Thus, the cell content $[n, m], \neq Z/2$ is expanded into the constraint $n \leq X \wedge X \leq m \wedge X \neq Z/2$.

4.2.2 Table Columns

Deriving from the CTTs, the GTTs column headers carry the designated input or output variable of the system under test. Therefore, every constraint in the

[1] We use Structured Text for the implementation.

table cells under the column has an implicit subject to refer to. We exploit this subject in the table cell abbreviation in Section 4.2.1.

In the following, we lift the restriction on the column variables by allowing internal state variables, and projection functions.

4.2.2.1 State Variables

By allowing the access of (internal) state variable in GTTs, we enable the specification of the internal behavior of the system, e. g., the specification of invariants or changes of global system state.

A column with a state variable can either be categorized as an input or output, resulting in different interpretations in the semantics. If we categorize a column with a state variable as an output column, the state column behaves just as if it were an output of the system: a violation of the corresponding constraint leads to non-conformance of the system. The constraint is an assertion. If the column is categorized as an input, its constraints are a prerequisite or assumption for the application of the input to the system. The value of a state variable is determined by the system and cannot be chosen by an environment in contrast to a normal input variable. But a constraint violation on the input state column does not lead to non-conformance of the system. For this reason, we use *assume* and *assert* as synonyms for the column categories *input* and *output*.

Additionally, a state variable has an assigned value before the cycle is executed and can be modified in the execution cycle. Therefore, in a GTT, we can refer to the value of state variables at both time points. We follow the convention: The categorization of a table column affects the evaluation of the corresponding constraint. By default, if we evaluate a state variable we refer to the last value. In an input (assume) column the state variable refers to the previous cycle (because the input constraints are evaluated before the system is executed). In an output column, the state variable refers to the value after the execution of the system. To make a clear distinction and to allow column-independent access, we introduce $X_{\texttt{pre}}$ and $X_{\texttt{post}}$ to refer to the value of a state variable X before and after the execution of the system. Technically, an input column for state variable X ($X_{\texttt{pre}}$) always has an implicit constraint $= X_{\texttt{post}}[-1]$, as this variable is not modifiable by the environment.

The value of state values before the execution of the first cycle is determined by the standardized default values or given in the variable declaration (cf. IEC 61131-3).

4.2.2.2 Projection Functions

We further generalize the variables in the column headers and the predicates in the table cells to increase the readability of large specifications.

Currently, a column is dedicated to a program variable, e. g., X, noted in the column header. This dedication describes a function that projects the current program state of the system under test to the value of the variable X. We lift the dedication and allow a user-defined function. This function maps the current and observed program states to a value. For example, to specify the CTU function block in Section 7.1 we introduce the $REdge := In \wedge \neg In[-1]$ function which returns true if there is a rising edge on the input signal In.

The constraints in the column's cells restrict the return value of the function. Furthermore, we allow a projection to an n-tuple in conjunction with n-arity predicates in the cells. Then, the values of the n-tuple are the parameters of the predicate. The typical comparison predicates, like equality "=" or less-than-equals "$\leq$" are built-in. Others can be defined by user-given function in the programming language. The other built-in abbreviations, e. g., "$\geq Y + 1$", are predicates with a single argument.

The projection functions in the column headers and the n-arity predicate in the abbreviations do not extend the expressiveness of GTTs, but allow us to write more comprehensible test tables: Instead of writing complex expressions in the table, we can concentrate on the most important: the consecutive execution flow of the test. Assume we want to express the relation $X/2 = Y + 1$, then we have three options: First, we can write the constraint as is in each cell on every column, without the use of any abbreviation. Second, we use either the column for X or Y with a rewritten version of the constraint: $=(Y + 1) * 2$, or $=X/2 - 1$, respectively. Third, we use the function $f\colon (\sigma) \mapsto (X/2, Y + 1)$ in the column header and the equality predicate ("=") in the cells of this column.

4.2.3 Repetitions

In GTTs, we can specify an allowed range of repetition for a group of rows. If the row group only contains one row, we use a designated column, the duration column, to hold the repetition constraint. Otherwise, we denote the repetition with bars on the right side of the table.

4.2.3.1 Generalization of Repetitions

Like the table cells, we also generalize the repetition constraints. But we only allow intervals $[n, m]$ or $[n, -]$ (where n, m are natural numbers), and the special symbol $-_\infty$. An interval specifies the lower (n) and upper (m or no upper bound)

bounds of row applications. If the upper bound is "—", the number of row applications is arbitrary but finite. A single "—" denotes an arbitrary finite row repetition; in longer notation: $[0, -]$. Strong repetition (indicated by using "$-_\infty$") denotes that a row is repeated infinitely often (Section 4.2.3.2).

If a duration constraint includes the value 0, then that row is optional and can be skipped. For systems with a constant cycle time, duration constraint can also be specified as time intervals, e. g., in milliseconds, instead of a number of repetitions: time intervals are converted to repetitions by dividing them by the cycle time.

Note that the selection of the possible rows is nondeterministic, e. g., in Figure 4.2 Row 2 and 3 can both directly succeed from Row 1. Additionally, the user can enforce progress in the selection of rows, when using the corresponding flag on the duration constraint: "$[m, n]_\mathrm{p}$". If the progress flag is given in a table row, this table row is only repeated if all its successor rows are not applicable. Thus, the progress flag specifies mutually exclusive selection between rows in favor of the successor rows (cf. Section 10.4.1).

4.2.3.2 Strong Repetition

Strong repetition is needed to specify that the desired pattern is *never* violated. Strict safety requirements are typical use cases of strong repetition. Later in Section 5.4.3, we see the don't-care duration "—" is not suited for specifying such properties since last table rows with "—" are satisfied if the software conforms to the row for a finite number of times—this also includes zero repetitions, and therefore the row is skippable and the system does not need to adhere to the last row.

4.2.3.3 Repetition of Multiple Rows

Multiple consecutive rows can be grouped to be repeated as a block. Every group of rows has its own duration constraint. With the row groups, one can express repetitive patterns, that span over more than one row, and also express optional sub-behavior. Row groups can be nested, i. e., a group may contain other groups and rows (but they cannot partially overlap). We mark row groups graphically by a line on the right side. With row groups, we can express specifications composed of repetitive or optional sections that span over multiple rows. The sections can represent different states of the software, e. g., initialization, error recovery, or automatic and manual operation. For an example see Section 4.3.3.

The repetition of a row groups leads to a jump to previous rows in the table—a property not present in CTTs and potential obstacle for comprehensibility. On the other hand, row grouping allows more compact specification of a system,

as complex and similar behaviors can be combined in a single GTT. Grouping of rows is also present in spreadsheet applications. Grouping of rows is a compromise between comprehensibility and expressiveness: CTTs only allows a consecutive execution, whereas more complex specification, i. e., automaton, allows jumping to arbitrary state (rows). In Section 9.2, we introduce meshed GTTs, an extension that allows nondeterministic jumps to arbitrary rows of different tables.

4.2.3.4 The Progress Flag

The progress flag is a supplementary annotation to a duration interval, denoted by a subscript $\cdot_p$. If a row is annotated with the progress flag, then the test must progress to a subsequent row if possible. For example, "$_{\mathrm{p}}$" is, like "—" (don't care), a repetition of arbitrary length, but unlike "—", the flag requires that the execution continues with a successor row if possible. Only if that is not possible, the current row is repeated. Using the progress flag ensures that the test does not get stuck unnecessarily.

Moreover, the progress flag helps to specify the deterministic software requirements concisely and leads to more comprehensible test tables. A typical pattern in the specification is waiting for a trigger event to occur, and then to proceed as specified. In a GTT, this is expressed by two successive rows: the first row allows all input and output values for an arbitrary duration. Its successor row specifies the trigger event by input constraints. When the second following row is satisfied, the first row needs to be vacated in favor of the second (cf. Section 4.3.3). Without the progress flag, any system would conform to this table because the system cannot violate the first row.

The flag does not add expressive power: An equivalent specification without "$_{\mathrm{p}}$" can be obtained by including the negation of the input constraints of the following rows to the current row and using "—" as duration. But this leads to tables that are unnecessarily difficult to read, as the assumption or assertion table cells of a row are conjunctively (and not disjunctively) joined.

4.3 Examples

We want to illustrate the features of GTTs on four examples. We select the RS flip-flop and the pulse timer function block of the IEC 61131-3, a typical factory scenario with a conveyor belt, and a non-PLC example by specifying a heating system. In the presented GTTs, especially Figures 4.7 and 4.8, we replace constraints, which do not change for at least three rows consecutively, with a line to increase the readability.

```
1 FUNCTION_BLOCK RS
2
3   VAR_INPUT S, R : BOOL; END_VAR
4   VAR_OUTPUT Q : BOOL; END_VAR
5
6   Q := (Q OR S) AND NOT R;
7
8 END_FUNCTION_BLOCK
```

Listing 4.4: Function Block **RS**

4.3.1 RS flip-flop

RS flip-flop is a function block for storing one Boolean variable. Listing 4.4 shows the Structured Text source code of this flip-flop. It has two Boolean inputs S and R and an output Q. This flip-flop is reset-dominant: If R is true, the output Q becomes false, regardless of the value of S. Otherwise, Q becomes true if S is true. If neither S nor R are true, Q maintains its original state.

Figure 4.5 show a GTT for the RS function block. Starting in Row 1: We can apply (arbitrary often) an arbitrary value for reset and false for set input, then the value of output needs to stay false. This includes, that the initial state of the flip-flop is false (R and S are false). In Row 2, we switch the state Q to true by switching on S and letting R false. Done once Q is true and stays true as long as the reset is off (Row 3). In Row 4, we reset the flip-flop, thus Q becomes false regardless of the value of S. These steps are repeated infinitely often.

#	INPUT R	INPUT S	OUTPUT Q	⏲	
1	–	FALSE	FALSE	–	⊤
2	FALSE	TRUE	TRUE	1	$\overline{\ }^{\infty}$
3	FALSE	–	TRUE	–	
4	TRUE	–	FALSE	≥ 1	⊥

Figure 4.5: A GTT for Function Block RS

```
1  FUNCTION BLOCK TP
2    VAR_INPUT IN : BOOL; PT : UINT; END_VAR
3    VAR_OUTPUT Q : BOOL; ET : UINT; END_VAR
4
5    VAR oldIn : BOOL ; END_VAR
6
7    IF Q THEN
8      IF ET >= PT THEN
9        Q := FALSE;
10     ELSE
11       ET := ET + 1;
12     END_IF
13   ELSEIF NOT oldIn AND IN THEN
14     Q := TRUE; ET := 0;
15   END_IF
16
17   oldIn = IN;
18 END_FUNCTION_BLOCK
```

Listing 4.6: Function block PT

4.3.2 Pulse Timer

Function Block TP is defined in [IEC61131-3, Figure 15(a)] as one of three timer function blocks. These timer blocks provide an abstracted and simplified way to deal with real-time. They are only defined by one graphical example in the standard and their implementation is left to the vendor of the PLC system.

We implement our own version of the pulse timer in Listing 4.6. In contrast to the standard, which uses the TIME data type, we use only integer variables to have a self-contained example without a dependency on a clock. The TP provides two input variables IN and PT and two output variables Q and ET. The idea is that the function block generates a pulse (Q is true) for the given time in PT when a rising edge in IN occurs. The output ET gives elapsed time of the pulse. We use a state variable to recognize a rising edge in IN. This functionality is typically implemented by using the R_TRIG function block.

Figure 4.7 shows an example GTT specifying the pulse timer function block. As long as the timer receives the input IN=FALSE, it keeps waiting and signals constant values (Row 1). When the pulse timer is started, which happens when

#	ASSUME		ASSERT		⏲	
	IN	PT	ET	Q		
1	FALSE	–	0	FALSE	≥ 1	$\top$
2	TRUE	≥ 1	$\geq$ ET$[-1]$, $<$ PT	TRUE	1	⏐
3	⏐	$=$ PT$[-1]$	$\geq$ ET$[-1]$, $<$ PT	TRUE	≥ 0	$\}\infty$
4	⏐	⏐	$=$ PT	–	1	⏐
5	$\bot$	$\bot$	$=$ PT	FALSE	≥ 1	$\bot$

Figure 4.7: A GTT for the TP function block. The vertical lines in columns IN and PT denote a repetition of "TRUE" resp. "$=$PT$[-1]$".

IN is set to TRUE (Row 2), the software must output Q=TRUE (signaling that the timer is running) for a time period which is defined by the pulse time provided as input in PT. In this GTT, we require that the input IN remains TRUE and PT remains constant as long as the timer runs. The latter is expressed using =PT[-1] in Rows 3–5, which requires PT to have the same value as in the previous cycle. A change of pulse time is forbidden during an active pulse.

Rows 2 and 3 correspond to the state in which the timer is running and the elapsed time ET has not reached the pulse time. The entries "1" resp. "≥ 0" in the duration column specify that the timer must be in this state for at least one cycle. While the timer runs, the elapsed time output ET must monotonically increase (">= ET[-1]").

When the elapsed time reaches the pulse time, the output Q, signaling that the timer is running, must switch from TRUE to FALSE (Row 5). Interestingly, the GTT leaves the exact switching behavior unspecified, allowing the PLC software to signal TRUE or FALSE in the first cycle where the pulse time is reached (Q is "don't care" in Row 4).

The GTT contains a strong repetition "$-_\infty$", which means that repeatedly starting the timer is possible.

4.3.3 Behavior of a Conveyor Belt

Figure 4.8 shows an error-handling functionality of a conveyor belt with complicated nested row groups. Row 1 defines a standstill until a workpiece arrives (WP=TRUE). Note that the progress flag in the duration constraint. If a workpiece arrives, the system needs to wait for at least one cycle to start the conveyor belt (Row 2). With Row 3, the specification branches: Either no error occurs (ERR=FALSE), then the conveyor should move forward for 5 to 6 seconds (Row 3). The row group covering only Row 3 makes this row optional (see the duration defined as $[0, 1]$). Or (Row 4) an error is detected (ERR=TRUE), then the error recovery (Row 4 to 6) should start. The error recovery is a back (Rwd) and forth

#	ASSUME WP	ASSUME ERR	ASSERT MOVE	⏲			
1	–	–	Stop	$-_p$			
2	TRUE	FALSE	Stop	> 1			
3	FALSE	FALSE	Fwd	$[5s, 6s]$	$[0, 1]$		
4	–	TRUE	Fwd	$1s$	$[0, 3]$	$[0, 1]$	∞
5	–	⊥	Rwd	$1s$			
6	TRUE		Stop	$500ms$			
7	TRUE	FALSE	Stop	1			

Figure 4.8: Example of a nested GTT for a conveyor belt.

#	ASSUME Tc [°C]	ASSUME Tb [°C]	ASSERT P	ASSERT B	⏲		
1	$(Tc - Tb) > d$	$[10, 60 + d]$	TRUE	FALSE	30s	$[0, 1]$	∞
2	$> Tb, < Tc[-1]$	$> Tb[-1], < 60 + d$	TRUE	FALSE	–		
3	$\leq Tb$	$\leq 60 - d$	⊥	TRUE	–	$[0, 1]$	
4	$\leq Tb$	$\leq 60 + d$		TRUE	–		
5	–	$> 60 - d, \leq 60 + d$		FALSE	$[1\text{min}, -]_p$		

Figure 4.9: An example GTT for a solar thermal system.

(Fwd) movement of the conveyor belt up to three times. The error recovery is exited when the workpiece reappears at the beginning and the error is reset as specified in Row 7.

4.3.4 Heating System

Figure 4.9 shows a GTT for a simple heating system, consisting of a solar thermal collector, which uses the energy of the sunlight to heat water, and an auxiliary gas burner, which is activated when the solar energy is not sufficient. The GTT specifies how the system should control its water pump (P) and the gas burner (B) in response to the water temperature in the boiler (Tb) and in the collector (Tc). The example has a global variable d which is used to make the specification parametric in the temperature span. For example, the constraint "$[60-d, 60+d]$" restricts the boiler temperature Tb to the depicted range and is an abbreviation for $60 - d \leq Tb \wedge Tb \leq 60 + d$ for any arbitrary d. A "don't-care" (–) constraint signals that the value may be chosen arbitrarily. References to values of past cycles are done with square brackets, e.g., "$< Tc[-1]$" specifies that the collector temperature is strictly decreasing compared to the last cycle.

4.4 Semantics: Conformance

The semantics of GTTs should be coherent with the engineer's intuition of the semantics on CTTs. We use this intuition as the base for the semantics of GTTs. For the formal semantics refer to Chapter 5.

CTTs describe a series of allowed and expected steps that form a test protocol. The test protocol is applied row-wise from top to bottom row. A row is applied with two actions: Firstly, the concrete input values are given to the system, and the system is invoked. Secondly, the computed system response is checked against the expected concrete output values. The protocol is applied until the end of the table is reached, or the system emits an unexpected output value.

We consider this protocol as a two-party game between the *challenger* and the *system*. The challenger stimulates the system with allowed input values to find misbehavior of the system, whereas the reactive system behaves as determined by its internal program, which is written to (hopefully) be correct. We see that a CTT forms a game in which the challenger plays against the *hard-coded* pre-determined system. The loser of the game is the party, who first violates the protocol described by the CTTs. The other one wins. If the end of the CTT is reached, the systems wins. Winning for the system means that it fulfills to the specification. A win of the challenger expresses a flaw in the system.

The semantics of test tables describe the notion of conformance. We say that a system conforms to a CTT if and only if the system wins the game. Conformance with a given test table implies that the test protocol is negative in the notion of the test theory, in particular that no bug is discovered by the given test table.

The intuition of the semantics can be discovered in two ways. First, we follow the trail where a GTT describes a family of CTTs, and use the semantics for them. Second, we apply the two-party game described above to GTTs directly, and see where it fits or might be adapted. In the core, both definitions represent the same conformance. We assume that the system under test or verification is deterministic, and thus produces the same output values for the same (history of) input values.

Reduction to a Set of CTTs. We start with an explanation with a reduction to CTTs. A GTT describes a family of test protocols of the above kind. These test protocols can be obtained in two steps: First, we instantiate every global (universally quantified) variable of a GTT. Second, we can retrieve CTTs from an instantiated GTT by unfolding the repetitions and cell constraints. It is important to remember that there are two levels of construction. The first construction level yields a set of instantiated GTTs, whereas the seconds level yields CTTs.

The intuition of the GTT semantics is best discovered by small examples. Let us consider the GTTs given in Figure 4.10. We focus on the second construction

level, thus GTTs are already instantiated and free of global variables. Every GTT in Figure 4.10 expands into two CTTs. The first GTT (Figure 4.10(a)) is expanded into a table with a single row, and one table with two rows—all rows claim A to be 0 and Z to be 1. The two CTTs of Figure 4.10(b) consist both of a single row with a constraint of $Z = 0$, but different constraints on variable A (one table requires $A = 1$ and the other $A = 2$). Analogously, the two CTTs expanded of the third GTT in Figure 4.10(c) have either $Z = 1$ or $Z = 2$, and $A = 0$.

Given the two CTTs of Figure 4.10(c), we notice that both test tables require that the input variable A always to be 1, but the output Z could be different. A deterministic system is not able to fulfill both CTTs: Given the same input value, the response of the systems determined to be either $Z = 1$ or $Z = 2$. We see on the "output side", we only need to adhere to one of the CTT. With Figure 4.10(b), we investigate the "input side" of GTTs. We also have two CTTs, one table allows A to be 1, the other allows A to be 2. And for both input values, the output Z should 0. To maximize the coverage, check the system response for both input values. Therefore, the "output side" needs to be adhered to for every allowed input value. Thus, the "input side" is universally quantified. Our observation is not surprising. A similar pattern, that the inputs are universally quantified whereas the output is existentially quantified, can be observed in the program synthesis by considering Church's problem [BL90]. We investigate the first GTT (Figure 4.10(a)) with this observation. For CTTs, we already noticed that the system conforms if the end of the test table (or protocol) is reached. We expand this table into two CTTs with a different number of rows, and the first row of both tables are equal—given 0 as the input, we expect the output to be 1. In the second step, we are at the end of the smaller table, and the larger table still requires the same constraints. From the intuition, we would claim

#	**ASSUME** A	**ASSERT** Z	⏲
1	0	1	[1,2]

(a)

#	**ASSUME** A	**ASSERT** Z	⏲
1	[1,2]	0	1

(b)

#	**ASSUME** A	**ASSERT** Z	⏲
1	0	[1,2]	1

(c)

Figure 4.10: Three GTTs where each GTT describes exactly two CTTs and has a different interpretation in the conformance.

that the system should always adhere to the constraints of the longest CTT. But this decision results in a contradiction if we consider it more formally. Given an observation of the system as a sequence of concrete input values and the corresponding computed output values, we need to decide if this observation conforms to one of the CTTs. If the observation is longer than the CTTs, we need to cut off the overhanging suffix to compare with these test tables. Now, the following situation can occur: The prefix of the observation conforms to a CTT, but the complete observation might not conform. In this case, we decide in favor of the system. Otherwise, a contradiction can occur, i. e., a given observation conforms to a GTT, but at least one of its suffixes does not conform. In practice, this corresponds to the following situation: We notice that the software violates the n-th step of the test protocol, but after performing the next step, we consider the system as conform.

To conclude these semantics observations, a system conforms to a GTT if and only if the system is stimulated with every input sequence of the CTTs and responses with an output sequence of the corresponding CTT (with to the same input sequence) for every instantiation of the given GTT.

Reduction to a Symbolic Game. Let us restart with the game behind the test protocol of CTTs. The main difference between GTTs and the CTTs are the constraints in the cells. We need to adapt this rigid game of CTTs with concrete values for the use of symbolical inputs and outputs (or assumptions and assertions).

The challenger starts by choosing an input value of the allowed value range described by the cell constraints of the input columns. The selected input value is given to the system, and the response of the system is checked against the constraints of the output columns. In the CTT, there is only one input and output value in the allowed ranges. From the testing perspective, the best coverage is obtained when we check the output value for every allowed input value. We see the same observation as in the section before. By the introduced nondeterminism (described by the repetition constraints), multiple steps may be active at the same time. In contrast to the CTTs, where the current step in the test protocol is always clear, it is more complicated to determine the current steps in the test protocol of GTTs. In connection to our previous semantics consideration, we see that playing the game on a GTT means that the game is played on multiple CTTs at once, in which the test protocol is (a) concretized by the emitted input and output values, and (b) split by nondeterministic choices of the challenger and the row selection into different test tables. Not every test table created "on-the-fly" has to be won by the system. Whereas every allowed input of the challenger has to be answered correctly by the system, the system can decide

which active table rows it wants to adhere to.

The winning condition for CTTs needs to be refined: For GTTs, a party loses after emitting a value which violates the constraints of all current rows. From CTTs, we inherit that the system wins when the end of the GTT is reached. Considering the global variables, we summarize: a system conforms a GTT iff system wins (or never loses) the two-party game on the GTT against an arbitrary challenger.

This two-party game is the base for our formal semantics definition in Section 5.3.3.

Chapter 5

Formalization of Generalized Test Tables

In this chapter, we capture our informal considerations into a formal structure. The formalization should sharpen our understanding, and help us to separate between a frontend and a backend. The frontend is the interface for the engineers, consisting out of nicely printed and formatted tables representing GTTs. The backend is the mathematical definition, which captures the essence of GTTs. The mathematical structure allows us to build decision procedures to check the conformance of GTTs (Chapter 6). The separation of front- and backend enables us to introduce new features for the engineers. And by reduction of these features to the existing mathematical definition, we can avoid fundamental changes in our considerations, decision procedures, or tools.

Our roadmap for this chapter is as follows: We start with the definition of a reactive system (Section 5.1), then we define the elements of the frontend and backend of GTTs (Section 5.2) and finally we close this chapter with the established formal definition of the conformances (Section 5.3) and their implications (Section 5.4).

5.1 Reactive Systems

Informally, our notion of a reactive system is introduced in Section 2.1. Formally, we distinguish between a reactive program P, the reactive system S, and the behavior of the reactive system $\mathcal{B}(S)$. A reactive program is the software (or the programmable logic in PLC) of the reactive system. Such a program consists of a list of instructions, which are periodically executed by the system, and a set

of declarations of input, output, and state variables ($InVar$, $OutVar$, $StateVar$). In the following we use $\Sigma = InVar \cup OutVar \cup StateVar$ to denote the set of defined variables of a program (and the system).

The internal state of a reactive program is an assignment of values to the state variables. The input variables are determined by the environment, and the state and output are computed by the program. Especially, the initial values of state variables are determined by their declarations (using default values in case no initial value is given, e. g., see IEC 61131-3). In the following we use $\mathcal{I}, \mathcal{S}$, and $\mathcal{O}$ to denote all possible assignments to input, output, or state variables. We can establish the semantics of a reactive program, which covers the execution of a single cycle.

Definition 5.1 (Semantics of reactive programs)**.** *The semantics $\rho(P)$ of a reactive program P is a state transition function $\rho(P) : \mathcal{S} \times \mathcal{I} \to \mathcal{S} \times \mathcal{O}$.*

The semantics $\rho(P)$ depends on the instructions in the reactive program and their semantics. For an example, Weigl [Wei15] gives an operational semantics for Structured Text. During the single cycle, the program reacts to the observed input variables $InVar$ and its last state $\mathcal{S}$, and then writes to the state variables and the output variables (in $StateVar$ and $OutVar$).

To be able to consider the effects of a reactive program over time, the above definition needs to be extended to infinite sequences (ω-words) of inputs and outputs. We define $\bar{i} \in \mathcal{I}^\omega$ as an infinite sequence over assignments of input variables $InVar$. The single assignments can be accessed using subscript indices, i. e.,, $\bar{i} = i_1, i_2, \ldots$. The reactive program as a stateful system needs an initial state s_0 from which it is launched. As mentioned above, s_0 is often determined by initial values of the data types for the variables. Note that the defined semantics defined below is the semantics of the reactive system. It is the responsibility of the system to periodically execute the underlying program.

Definition 5.2 (Trace Semantics of Reactive Systems)**.** *The behavior $B(S)$ of a reactive system S executing the reactive program P with initial state $s_0 \in \mathcal{S}$ is the function $B(S) \colon \mathcal{I}^\omega \to \mathcal{O}^\omega$ defined by*

$$B(S)((i_1, i_2, \ldots)) = (o_1, o_2, \ldots)$$

where $(s_n, o_n) = \rho(P)\Big((s_{n-1}, i_n)\Big)$ for all $n \in \mathbf{N}_{\geq 1}$.
Also, $\mathcal{B}(S) \subseteq \mathcal{I}^\omega \times \mathcal{O}^\omega$ denotes all possible behaviors of a reactive system S

$$\mathcal{B}(S) := \{(i, B(S)(i)) \mid i \in \mathcal{I}^\omega\}$$

This definition says that starting from the initial state s_0, the reactive program is executed repeatedly, applying in each cycle $\rho(P)$ to its current state s_{n-1} and the input tuple $i_n \in I$ to produce the output tuple $o_n \in \mathcal{O}$ and the new state s_n.

Trace semantics use the internal state in the definition, but when taking an outside look at the semantics, it defines input/output behavior and does not make statements about the internal state space. In practice, we often do not distinguish between state and output variables as they behave nearly identically in our considerations. For example, consider the Structured Text code in Section 4.3.2, where we freely read and write to the output variables as if they were state variables. But such a separation between state and output variables can always be established and helps in our explanation.

5.2 Syntactical Representation of Tables

In this section, we describe the way from the user's perspective on GTTs to a unified mathematical notion. We define the frontend of GTTs by declaring the syntax of cells and time constraints, and the backend by using inductive definitions.

Deriving from the reactive system under test, a GTT is defined over a set of program variables, which are categorized as input, output or state variable. We denote this signature with Σ. Over this signature we can define the grammar within the table cells. We distinguish between the more restrictive cell content in the duration column, and the cell content of program variables.

Definition 5.3 (Syntax of time constraints)**.** *The content of a cell in the duration column is derived by the following production rule:*

$$\begin{aligned}\langle time\rangle \quad ::= \; & \text{`['} \; \langle pint\rangle \; \text{`,'} \; (\langle pint\rangle \mid \text{`-'}) \; \text{`]'} \; [\text{`\_p'}] \\ \mid \; & \texttt{strongrep} \mid \langle pint\rangle\end{aligned}$$

The non-terminal $\langle pint\rangle$ denotes a positive integer literal.

A short remark on our grammar notation: We use an adapted version of the Backus-Naur-Form. Terminals, the printable characters of the grammar, are marked with single quotes or monospace font, e. g., `'['` represents the square left bracket and `strongrep` the keyword "strongrep". Non-terminals are denoted by angle brackets, e. g., $\langle time\rangle$, and mark the application of this grammar. We use several meta-characters to make the grammar more readable: We use parenthesis to mark the precedence within the production rules, and square brackets mark the optional parts of a production rule, e. g., to express that the non-terminal "_p" can be added optionally to an interval we write "`['_p']`". In the next grammar, we use "$(X)^*$" to mark an arbitrary finite repetition of the parenthesized term X.

We can distinguish the two main cases for the cell content of the duration column: The content is either an interval over integers or "strongrep". The

intervals can either be restricted with a lower and upper bound $[n, m]$ (with $n, m \in \mathbb{N}$, and $n \leq m$), or have only a lower bound ($\geq n$, $[n, -]$). Singleton intervals $[n, n]$ are just specified with a single number n. Additionally, intervals with a non-singleton range (i. e., with a degree of freedom in their allowed iteration number) can be suffixed with "_p" to enable the progress-flag.

The cell content of the program variable columns is far more complex. On the top level, the cell content is a list of clauses. A clause is either an abbreviation, e. g., a literal or a single-sided expression, or a Boolean expression.

Definition 5.4 (Syntax of cell content)**.** *The syntax of a cell content is given by the following production rules with start rule* ⟨*cell*⟩*:*

⟨*cell*⟩ ::= ⟨*clause*⟩ (‘,’ ⟨*clause*⟩)*

⟨*clause*⟩ ::= ⟨*dontcare*⟩ | ⟨*id*⟩ | ⟨*constant*⟩
| ⟨*singlesided*⟩ | ⟨*interval*⟩ | ⟨*expr*⟩

⟨*dontcare*⟩ ::= ‘-’

⟨*constant*⟩ ::= ⟨*int*⟩ | TRUE | FALSE

⟨*singlesided*⟩ ::= (‘<’ | ‘>’ | ‘<=’ | ‘>=’) ⟨*expr*⟩

⟨*interval*⟩ ::= ‘[’ ⟨ *expr*⟩ ‘,’ ⟨*expr*⟩ ‘]’ ‘;’

⟨*expr*⟩ ::= ‘-’ ⟨*expr*⟩ | NOT ⟨*expr*⟩ | ‘(’ ⟨*expr*⟩ ‘)’
| ⟨*expr*⟩ ⟨*bop*⟩ ⟨*expr*⟩ | ⟨*constant* ⟩
| ⟨*id*⟩ [‘(’ ⟨*expr*⟩ (‘,’ ⟨*expr*⟩)* ‘)’]
| pre(⟨*id*⟩) | post(⟨*id*⟩)

⟨*var*⟩ ::= ⟨*id*⟩ [‘[’ ‘-’ ⟨*int*⟩ ‘]’]

where ⟨*bop*⟩ *represents a binary operator:* and, or*, "+", "-", "*", "/", and* mod*.*

The abbreviations are given in Table 4.3, and the expressions follow the rules of common programming languages like Structured Text, allowing unary and binary expressions over variables and function applications. The grammar for the predicate and variable abbreviation collide and are represented by a non-terminal ⟨*id*⟩ (in ⟨*clause*⟩). Both abbreviations are distinguished semantically by the variable signature after parsing.

In contrast to programming languages, $x[-n]$ denotes the access to values of previous cycles, and does not denote array access. The pre- and post-execution value of a state variable, e. g., X_{pre} and X_{post}, is denoted by $pre(X)$ and $post(X)$. Moreover, the access to sub-fields of structures is not covered. But this grammar is sufficient to cover the constraints in this thesis. Moreover, the grammar may be adapted to cover features of the language of the reactive program to be

verified or to meet the intuition of the engineer. For example, we often use the Unicode symbols for the mathematical operators to have more compact tables, so we write $\leq$ instead of "<=".

The grammar is ambiguous. Especially, variables and constants can either be derived via two productions from the non-terminal ⟨*clause*⟩, directly via ⟨*constant*⟩ or ⟨*variable*⟩, or indirectly via ⟨*expr*⟩. For the interpretation, we give the first production variant preference and interpret variables and constants as an abbreviation, e. g., the cell content "TRUE" or "5" is interpreted as an abbreviation and not as an expression. This takes effect on translating cell expressions down into the backend. In the backend, the cell constraints of each column category are expressed into constraint expression.

Definition 5.5 (Constraint expressions $\mathfrak{E}$). *Let $\Sigma = InVar \cup StateVar \cup OutVar$ be a signature of program variables and $GVar$ the signature of global variables, then the set $\mathfrak{E}$ of cell expressions is defined inductively:*

- $v \in \mathfrak{E}$ *and* $v[-n] \in \mathfrak{E}$ *for every program variable* $v \in \Sigma$ *and* $n \in \mathbb{N}$
- $\mathrm{next}(v) \in \mathfrak{E}$ *for every state variable* $v \in StateVar$
- $g \in \mathfrak{E}$ *for every global variable* $g \in \Gamma$.
- $e \circ f \in \mathfrak{E}$ *for every expression* $e, f \in \mathfrak{E}$ *and* $\circ \in \{+, -, \leq, \wedge, \vee, \ldots\}$.

The definition of constraint expressions is similar to the grammar above, but adds more restrictions to the expressions: All abbreviations are gone. The pre- and post-execution access has to be resolved and translated to the next-function, accordingly. Function applications have to be evaluated into a closed form.

We define a mathematical structure of time expression in the duration column:

Definition 5.6 (Time expressions $\mathfrak{T}$). *A time expression $\tau \in \mathfrak{T}$ is either ω for strong repetition, a* lower-limited *or* lower-upper-limited *interval:*

$$\mathfrak{T} = \{\omega\} \tag{5.1}$$
$$\cup \{[m, -] \mid m \in \mathbb{N}\} \tag{5.2}$$
$$\cup \{[m, n] \mid m, n \in \mathbb{N}\} \tag{5.3}$$

The mathematical structure meets the grammar in Definition 5.3 almost exactly: only the "strongrep"-Keyword and the single numbers n need a translation into ω and $[n, n]$, respectively. The progress flag vanishes during the translation and is handled differently.

Progress Flag. The progress flag prevents the challenger from selecting input values that adhere to the assumption constraints from a row and its successor rows at the same time. This restriction only applies when a row could be left, thus the time constraint needs to allow this. Thus, a progress flag has no effect on strong repetition and fixed repetitions ($[n, n]$, for any $n \in \mathbb{N}$). The mathematical language of time expression (Definition 5.6) does not contain the progress flag anymore. The progress flag is consumed during the translation (Section 5.2) of the GTT given in the graphical table-form into the mathematical notion given in Definition 5.7. This also defines its semantics in detail. But before we resolve the progress flag, we have to define the successor rows of a table row on the mathematical structure of GTTs.

GTT Structure. Due to the row groups, a GTT is a recursive structure of nested and repeated tables.

Definition 5.7 (Structure of GTTs)**.** *Let $\mathfrak{E}$ be a set of expressions over the signature $\Sigma \cup GVar$ and $\mathfrak{T}$ be the set of time expressions, then the set of all GTTs $\mathbb{T}_{\Sigma \cup GVar}$ is defined as*

- *The empty table $\epsilon \in \mathbb{T}$ is a GTT.*
- *$(\phi, \psi) \in \mathbb{T}$ is a GTTs, where $\phi, \psi \in \mathfrak{E}$*
- *Let $T, T' \in \mathbb{T}$ be two GTTs, then the sequential composition of both tables with the repetition $\tau \in \mathfrak{T}$*

  $$\langle T, T' \rangle^{\tau} \in \mathbb{T}$$

  is also a GTT,

The base of the definition is the table with a single row or the empty table ϵ. A row consists of a tuple $(\phi, \psi) \in \mathfrak{E}^2$. ϕ is the conjunction of all constraints in the assumption (input) columns of a row, and ψ is for the assertion (output) columns, respectively. The empty table ϵ is the neutral element for sequential composition, and $\tau = [1, 1]$ is the neutral element of repetition. Thus, we can repeat tables without concatenating another table, or concatenate a table without adding a repetition. Allowing us to add the time constraint to a single table row by the concatenation with ϵ. If we concatenate multiple tables together, we omit the unnecessary angle bracket and empty tables, hence $\langle T, \langle T', T'' \rangle \rangle$ can conveniently written as $\langle T, T', T'' \rangle$

Row Successor. Due to the intervals in the duration constraints and row groups, it is not automatically clear which are the successor rows of each row (as intermediate rows may have zero duration). Intuitively, the definition should be clear. A row is a successor of the row above it. Also, the beginning of a row group is a successor of the end of this row group. And if a table row is skippable, its successors are also the successors of its predecessor.

Formally, this is not easy to cover due to the inductive definition. For the definition, we define two functions $first(T)$ and $last(T)$ on tables, where $first(T)$ returns the first table row in the table T, and $last$ for the last row, respectively. We define the set $succ(r) \subseteq R$ as the set of the possible successor for the row r in GTT where R is the set of all table rows.

Definition 5.8 (Successor row $succ$)**.** *Given a GTT $T \in \mathbb{T}$. Let R be the set of all tables rows in T, $r = last(T_1)$ (the last row of T_1), and $s = first(T_2)$ (first row of table T_2). We define the successor relation by case distinction over the structure of T.*

The row $s \in R$ is a successor $r \in succ(s)$ of a row $r \in R$ if and only if

- *s is a direct successor of r, occurring as $\langle \ldots, T_1, T_2, \ldots \rangle^\tau$ in T, or*
- *s and r are only separated by skippable test tables $T_m, \ldots, T_n$, occurring as $\langle T_1, T_m, \ldots, T_n, T_2 \rangle$ in T where $0 \in \tau_i$ for all $i \in [m, n]$*
- *r is the end $\langle T_2, \ldots, T_1 \rangle^\tau$ and s is the start of a row group and the block can be repeated more than one time ($\exists n.n \geq 2 \wedge n \in \tau$),*
- *a row t is successor of $t \in succ(r)$ and $s \in succ(t)$, also t is skippable (the time constraint τ_t on t allows 0), when $r \in succ(s)$.*

The first item captures pairs of rows that are directly successive in the table. This also spans across row groups, where the upper row is in a different group as the row below. If we count the rows from top to bottom (regardless of the row groups), then $k+1$-th row is the successor of k-th row. We use the function $first$ and $last$ to avoid dealing with arbitrary nested definitions. The second item deals with the situation of skippable row groups between a table row and its successor. The third item describes the situation of the jump from the end of a row to the start of the group. This is only possible if the row group can be executed at least twice. The last item is the recursion in the definition describing that the predecessors of a row inherit its successor if the row is skippable. Note that the last rule is not included in the second rule—the order of the tables (or

row) is determined (row s is above row r). In the last rule, we do not care about the table orders, e. g., row r could be the second row of a group, row t is the first row of the same group and skippable, and s is the last row. Then, using the last rule, we can establish that we are able to jump from s to r directly.

With $succ(0)$ we denote the rows which are initially reachable. The 0 stands for an imaginary unskippable zeroth row which is added in front of the user-declared rows.

Translation. Given a GTT in its table form, we can translate it to its mathematical notion (Definition 5.7) with the following steps: First, every cell content is expanded to its longer form as a constraint expression. Second, the constraint expressions of the assumption columns are joined into one conjunction, analogously for assertion columns. In the third step, we handle the access of state variables. Let $X \in StateVar$ be a state variable, then $pre(X)$ is translated into the access of X, and $post(X)$ is translated into $\mathrm{next}(X)$. Every access to a state variable Y where an explicit pre- or post-operator is missing is treated like $pre(Y)$ if this access is in an assumption column; or as $post(Y)$ in an assertion column, respectively.

Graphically spoken, we have rewritten a GTT T into a new GTT T' with only two columns containing constraint expressions. This can directly be rewritten into the mathematical structure if we just drop the progress flag. We assume, that $succ$ is the successor relation $succ$ for T', and as T and T' are equal in the number of rows and their nesting, $succ$ is also valid for T. We get rid of the progress flag with the following rule: Let $[n, m]_p$ be a time expression of the row r. W. l. o. g. let $n = 0$, meaning the progress flag is instantly considered on entering the table row r, then we replace the old assumption constraint ϕ_r of row r in T' with the new assumption constraint ϕ' defined as

$$\phi'_r := \phi_r \wedge \bigwedge_{s \in succ(r)} \neg \phi_s \tag{5.4}$$

If $n > 1$, we need to split the row r into two rows, one containing the fixed time constraint $[n, n]$ and the other row with time constraint $[0, m - n]$. Both rows contain the same assumption ϕ and assertion constraint ψ. If $n = m$ or the time constraint is ω, the progress flag has no effect, due to the lack of the nondeterministic choice of rows.

Example 5.9. *The following term shows the structure for the GTT in 4.8:*

$$
\begin{aligned}
\langle\ & \langle \neg WP \vee ERR, true\rangle^{[0,\infty]}, \\
& \langle WP \wedge \neg ERR, MOVE = \texttt{Stop}\rangle^{[1,\infty]} \\
& \langle\langle \neg WP \wedge \neg ERR, MOVE = \texttt{Fwd}\rangle^{[50,60]}\rangle^{[0,1]} \\
& \langle\langle\ \langle ERR, MOVE = \texttt{Fwd}\rangle^{[10,10]} \\
& \quad \langle ERR, MOVE = \texttt{Rwd}\rangle^{[10,10]} \\
& \quad \langle WP \wedge ERR, MOVE = \texttt{Stop}\rangle^{[5,5]}\rangle^{[0,3]}, \\
& \langle WP \wedge \neg ERR, MOVE = \texttt{Stop}\rangle^{[1,1]}\rangle^{[0,1]}\rangle^{\omega}
\end{aligned}
$$

Note that the progress flag is translated into the assumptions.

5.3 Semantics

The semantics of GTTs define whether a reactive system, given as a set of infinite traces defined by the trace semantic (Definition 5.2), conforms to a GTT. In Chapter 4 we give an informal description of our conformance notion. To express a formal definition of the conformance, we define the evaluation of our constraint expressions (Definition 5.5).

5.3.1 Evaluation of Constraints

Constraint expressions are evaluated against a finite trace of the input, state, and output variables, often given as a prefix of an infinite trace from a behavior of the system under test. Such a trace is a finite sequence over input values $\mathcal{I}$ and state values $\mathcal{S} \cup \mathcal{O}$. For the definition below, we consider the values for the output variables as part of the returned state. Note that the constraint expressions do not contain any global variable. The global variables are instantiated beforehand, thus, they do also not appear in evaluation.

Definition 5.10. *Let $v \in (\mathcal{I} \times \mathcal{S})^*$ be a partial trace of length $n \geq 2$. Let $v{\downarrow_n}$ be the prefix of v with length n, whereas $v_n = (i, s)$ is the last element of the trace. Then, the valuation function $val_v(e)$, which assigns a value to every expression*

$e \in \mathfrak{E}$, *is inductively defined by:*

$$
\begin{aligned}
val_v(e \circ f) &= val_v(e) \circ val_v(f) \quad \textit{for} \circ \in \{+, -, \leq, \wedge, \vee, \ldots\} \\
val_v(X) &= i(X) \quad \textit{if } X \in \mathit{InVar} \\
val_v(X) &= val_{v\downarrow_{n-1}}(X) \quad \textit{if } X \in \mathit{StateVar} \\
val_v(\mathrm{next}(X)) &= s(X) \quad \textit{if } X \in \mathit{StateVar} \\
val_v(X) &= s(X) \quad \textit{if } X \in \mathit{OutVar} \\
val_v(X[-k]) &= val_{v\downarrow_{(n-k)}}(X) \quad \textit{if } k < n \\
val_v(X[-k]) &= \texttt{undef} \quad \textit{if } k \geq n
\end{aligned}
$$

The evaluation of constraints is forwarded to the typical interpretation of the binary or unary operations. The detailed operation semantics is selected in alignment with the operator semantics from Structured Text. More special are the past references. Given a reference $X[-k]$ we cut off the last k symbols of the given trace, and evaluate X on this prefix. Values of the input variables are obtained from the assignment of the input variables i, and for state variables s, respectively. State variables can be referenced in the pre- or post-execution (X_{pre} and X_{post}) of the system. To capture this, let $val_v(X)$ refer to the previous value of the state variable X, and $val_v(\mathrm{next}(X))$ to the current value of X. An output variable always refers to its current value. Access to past reference $X[-k]$ which lies outside the observed trace v ($|v| <= k$) is undefined, and can be interpreted depending on the intended purpose, e. g., verification or synthesis.

5.3.2 Unrolled Instances of Generalized Test Tables

The rows of GTTs have a duration and may be repeated more than once. In a first step towards defining the semantics of GTTs, we eliminate the indeterminism w.r.t. the repetition of rows and define the set of *unrolled instances* of a GTT by making the repetitions explicit.

Definition 5.11 (Unrolled Instances $\mathrm{SP}(T)$)**.** *Let $T \in \mathfrak{T}$ be a GTT without global variables. The set*

$$\mathrm{SP}(T) \subseteq (\mathfrak{E} \times \mathfrak{E})^{\omega} \cup (\mathfrak{E} \times \mathfrak{E})^{*}$$

denoting unrolled instances *of T consists of finite and infinite words which are obtained by unfolding the time constraints:*

$$
\mathrm{SP}(T) := \begin{cases} \{\varepsilon\} & \textit{if } T = \epsilon \\ (\phi, \psi) & \textit{if } T = (\phi, \psi) \\ (\mathrm{SP}(T') \cdot \mathrm{SP}(T''))^{\tau} & \textit{if } T = \langle T', T'' \rangle^{\tau} \end{cases} \tag{5.5}
$$

The instances in $\mathrm{SP}(T)$ are obtained from T by treating a given table structure as a regular expression over infinite words. In particular $\mathrm{SP}(T)$ can be constructed recursively by applying the concatenation and Kleene-closure of formal languages. Note that ε denotes the empty word, whereas ϵ represents the empty GTT.

The set $\mathrm{SP}(T)$ defines an infinite language with finite and infinite words. Each word represents possible plays (traces of concrete input and state values) between the challenger and the system in a symbolic abstract fashion. A word in $SP(T)$ is a sequence of assumption and assertion pairs (ϕ, ψ), describing valid moves in the play before and after the execution of the system. Later, we try to match an observed trace over input and state values to such a symbolic play. One symbolic play is similar to a single (possible infinite) GTT without any time constraints.

In general, the set of unrolled instances for a GTT is infinite. This does not pose a problem as the notion of unrolled instances is only used as a theoretical concept for defining the semantics of GTTs.

Global variables are not considered in unrolled instances, as their semantics is defined via universal quantification: A system has to conform to the test table for all its instances (see Definition 5.12).

5.3.3 Two-Party Game for Defining Test Conformance

The intuition behind the following definitions is the following: A reactive system S conforms to a GTT T if every trace $t \in \mathcal{B}(S)$ conforms to T, where a trace conforms to T if one of the following conditions holds: (a) the input/state pairs of t satisfy *all* rows of *at least one* instance in $\mathrm{SP}(T)$, or (b) t fails to satisfy the input constraints of *all* instance in $\mathrm{SP}(T)$. In the former case, the trace is covered by the specification described by T, in the latter case, the input sequence triggers an application scenario that is not covered by the specification.

Formally, we define the semantics of a GTT T utilizing a game played between a challenger (that chooses the inputs) and the reactive system S under test (that chooses the values of the state and output variables). The challenger can be identified with the environment of the system. The game is played operating on a set $\widehat{\mathrm{SP}}$ of unrolled instances of T from which in every round inconsistent and conflicting symbolic plays are removed.

The player removing the last consistent instance from $\widehat{\mathrm{SP}}$ loses the game. In addition, the system can win by successfully reaching the end of one of the non-eliminated table instances.

Figure 5.1 shows the game's rules in algorithmic form. During a game, $\widehat{\mathrm{SP}}$ holds the set of unrolled instances of T which have not been eliminated, v holds the observed partial trace up to and including the current move, and

Input: A GTT T
$\widehat{\mathrm{SP}} \leftarrow \mathrm{SP}(T)$;
$v \leftarrow \epsilon$;
$k \leftarrow 1$;
while *true* **do**
  Challenger chooses $i \in I$;
  System computes $o \in O$;
  $v \leftarrow v \cdot (i, o)$;
  $\widehat{\mathrm{SP}} \leftarrow \{D \in \widehat{\mathrm{SP}} \mid v \models \phi_k$ for the k-th row $t_k = (\phi_k, \psi_k)$ in $D\}$;
  if $\widehat{\mathrm{SP}} = \emptyset$ **then**
    /* Chosen input not covered by T */
    return *System wins*;
  end
  $\widehat{\mathrm{SP}} \leftarrow \{D \in \widehat{\mathrm{SP}} \mid v \models \psi_k$ for the k-th row $t_k = (\phi_k, \psi_k)$ in $D\}$;
  if $\widehat{\mathrm{SP}} = \emptyset$ **then**
    /* Chosen output violates T */
    return *Challenger wins*;
  end
  if $\exists D \in \widehat{\mathrm{SP}}.\ |D| = k$ **then**
    /* Unrolled instance D has finished */
    return *System wins*;
  end
  $k \leftarrow k + 1$;
end

Figure 5.1: Game between challenger and system w.r.t. a GTT T

k counts the iterations. Initially the set $\widehat{\mathrm{SP}} = \mathrm{SP}(G)$ contains all unrolled instances of the GTT T. In each round, the challenger chooses input values, and the system under test computes its output from its internal state and the input values. The functions which choose the input/output values depending on the observed partial trace are called *strategies*. Since reactive programs are deterministic, there is only one strategy for the program, which is encoded in its implementation. The challenger is not confined in its choices; there are many possible strategies for the challenger. Whenever $\widehat{\mathrm{SP}}$ becomes empty, i. e., no unrolled instance of T satisfies the partial trace, the respective player loses the game: If this is caused by the input constraint ϕ_k being violated, the challenger loses and the system wins. If $\widehat{\mathrm{SP}}$ becomes empty because the output

#	**I**	**O**	🕒
1	—	1	—
2	—	2	[1,1]

Figure 5.2: GTT illustrating the difference between strict and weak conformance

constraint ψ_k is violated, the system loses and the challenger wins. If $\widehat{\mathrm{SP}}$ contains a consistent unrolled instance that has been fully traversed (its length is the current iteration counter), then the partial trace v is a witness for the system conforming to the GTT. The system wins.

A single game has three possible outcomes: Either (a) the challenger wins, or (b) the system wins, or (c) neither party wins (draw). In the case of a draw, the game is infinite, while a game where one player wins ends after a finite number of iterations. Note games on a GTT T without strong repetition are always finite, because all unrolled instances $\mathrm{SP}(T)$ are finite.

A strategy for one party is called a winning strategy if it wins every possible play regardless of the other party's strategy. The definition of conformance to a GTT can now be defined based on who wins the games:

Definition 5.12 (Weak and Strict Conformance)**.** *The reactive system S* strictly conforms *to the GTT T iff its strategy is a winning strategy for the game shown in Figure 5.1 for all instantiations σ of global variables of the GTT T. The reactive system S* weakly conforms *to T iff its strategy never loses.*

The difference between weak and strict conformance is whether the analysis of a system (w.r.t. a test table) successfully finishes after finitely many steps or the system is under consideration for infinitely many steps. For example, consider the GTT shown in Figure 5.2. Intuitively, it requires that, independently of the input, the output must eventually be 2 after an arbitrary number of cycles with output 1. The reactive system that always returns 1 (and never 2) does not have this property. Correspondingly, it does not strictly conform to the table (it does not have a winning strategy). But it weakly conforms (it never loses either). This corresponds to the fact that by inspecting finite partial traces, one cannot decide whether this system violates the test table.

Any analysis that only considers partial traces (like run time monitoring or testing) can, in general, only test weak conformance. A static analysis, however, is able to analyze a reactive system w.r.t. strict conformance. The definition of conformance (Definition 5.12) can be lifted to the case of nondeterministic reactive systems by requiring that *all* possible strategies of S must be winning strategies.

This semantics definition seems unnecessarily complicated, but an attempt to define it on the program traces is bound to fail as the implication of a violated

constraint is different depending on whether it occurs on the input or on the output values: A GTT with constraint $false$ on the input side is trivially satisfied, while $false$ on the output side makes it unsatisfiable. Yet, both describe the same set of concrete traces: the empty set.

There is a third conformance definition which also requires a certain liveness and progress of the system:

Definition 5.13 (Cooperative Conformance)**.** *The reactive system S* cooperatively conforms *to GTT T iff its strategy always reaches the end of every instantiation of T.*

This conformance definition states the system wins only by reaching the end of the table and forbids winning by breaking the constraints of the challenger. Consider that the assumptions can refer to previous values of output and state variables. These back-references give the system the chance to make the constraints of the challenger falsifiable. In cooperative conformance, the strategy of the system must be cooperative and any limitation for the challenger is prohibited. Besides a cooperative system, verification of this conformance requires also a cooperative challenger (cf. Equation (6.8)), which tries to move the plays forward to the next table rows. No system can cooperatively conform to a GTT containing strong repetition. In detail, if every symbolic play in SP is infinite, cooperative conformance is not applicable. The cooperative conformance is mainly a tool for the validation of the GTT-specification on the verification subject. As this conformance fails when no suitable option for the challenger exists, we can check whether a system strictly conforms because of invalid turns of the challenger, or because the end of the table is reached.

5.4 Properties

The game semantics yields interesting and surprising properties for the conformance of systems and their practical application.

5.4.1 Strict Conformance and Strong Repetition

There are two remarkable properties for strict conformance in connection with strong repetition.

No Strict Conformance. Strong repetition in a row prevents a system from reaching the end of the table. Therefore, a system can only strictly conform to such GTT if there is an inevitable violation of the challenger. Generalized to an arbitrary play of the game, we can state:

Proposition 5.14. *Let $\widehat{\mathrm{SP}}_k$ be a set of remaining symbolic plays in round k. If all remaining plays are infinite, i. e., $\forall w \in \widehat{\mathrm{SP}}_k.\ |w| = \omega$, then the system cannot win by reaching the end of the table.*

Proof. If the end of the table is reached, then it is reached in a finite number of rounds $k \geq k'$, but all remaining plays have length ω (which is larger than every $k' \in \mathbb{N}$). □

Moreover, to prevent the system from winning, we need to require the existence of a challenger that can survive the game infinitely long, i. e., there exists a never-losing strategy. Therefore, all input constraints always need to be satisfiable.

Proposition 5.15. *If there exists an infinite valid challenger stimulus $\bar{i}$, i. e., $\exists(\bar{\phi}, \bar{\psi}) \in \widehat{\mathrm{SP}}_k.\ \forall k \in \mathbb{N}.\ \bar{i}_k \models \bar{\phi}_k$, and the condition of Proposition 5.14 holds, then there is no* strict *conform system.*

Proof. Consider the proof for Proposition 5.14. The only possible winning-exit for the system is blocked by the assumption of an infinite valid stimulus. □

Strong Repetition does not Extend the Expressiveness. Interestingly, for every GTT T_∞, that contains strong repetitions, we can construct a GTT T_*, that does not contain strong repetitions, with equal (*strict* and *weak*) conformance for every system. T_* is obtained by replacing every strong repetition with a "don't-care", and adding a sentinel row that prevents the row from ever being left. Formally, the transformation of T_∞ to T_* is defined as:

Definition 5.16 (Construction of T_*)**.** *T_* for a given GTT T_∞ is constructed by the following recursive function $trans(T_\infty)$:*

$$trans(T) := \tag{5.6}$$

$$\begin{cases} \epsilon & \text{if } T = \epsilon \\ (\phi, \psi) & \text{if } T = \langle (\phi, \psi) \rangle \\ \langle trans(T_1), \ldots, trans(T_n) \rangle^{\tau'} & \text{if } T = \langle T_1, \ldots, T_n \rangle^{\tau} \wedge \tau \neq -_\infty \\ \langle \langle trans(T_1), \ldots, trans(T_n) \rangle^{[0,-]}, & (\Phi(T), \texttt{false}) \rangle^{[1,1]} \\ & \text{if } T = \langle T_1, \ldots, T_n \rangle^{\omega} \end{cases} \tag{5.7}$$

$\Phi(T)$ is defined by the disjunction $\bigvee_{(\phi,\psi,\tau) \in succ(0)} \phi$ of all immediately reachable rows $succ(0)$ in T from the start.

The trick of the construction lies in $\Phi(T)$ and `false`. The latter constraint prevents the system from winning the test table, and $\Phi(T)$ prevents a *cheating* challenger.

Our goal is to prove the equality in the *weak* and *strict* conformance.

Proposition 5.17 (Equal Conformance)**.** *A reactive system* weakly, strictly *or* cooperatively *conforms to* T_∞ *iff it is* weakly *conform to* T_*.

The conformance of a test table is based on the outcome of all possible plays, i. e., a system *strictly* conforms if and only if it is a winning strategy and *weakly* conforms if and only if its strategy never loses.

The next lemma reduces the equal conformance to the equal game outcome on all plays.

Lemma 5.18. *Two* GTTs T, T' *have equal conformance if the game outcome for all possible plays is equal.*

In the remaining of the proof for Proposition 5.17, we need to show that for an arbitrary play, without assumptions on the table or the system, the outcome (winner and loser) is equal on T_* and T_∞. The outcome is determined by the Figure 5.1. We show that by defining a coupling between both runs of the algorithm for T_* and T_∞. More formally, the algorithm is based on a set $\widehat{\mathrm{SP}}$ that holds the remaining possible unwound plays of the GTT. $\widehat{\mathrm{SP}}$ is determined by $\mathrm{SP}(T)$ in the beginning. The outcome is decided round-wise. We established a coupling relation between the set $\widehat{\mathrm{SP}}$ from the run over a play in T_∞ and T_*: $\widehat{\mathrm{SP}}^k_\infty$ and $\widehat{\mathrm{SP}}^k_*$ for round $0 < k$.

To define the coupling relation, we need the following property on the structure of $\widehat{\mathrm{SP}}^k_\infty$ and $\widehat{\mathrm{SP}}^k_*$.

Lemma 5.19 (Separation of $\widehat{\mathrm{SP}}^k$)**.** *In every round* $k \geq 0$, S^k_∞ *and* S^k_* *can be separated:*

$$\widehat{\mathrm{SP}}^k_\infty = L^{fin} \cup \bigcup_l \alpha_l \cdot \beta_l^\omega \tag{5.8}$$

$$\widehat{\mathrm{SP}}^k_* = L^{fin} \cup \bigcup_l \alpha_l \cdot \beta_l^* \cdot (\Phi_l, \texttt{false}) \tag{5.9}$$

for the languages α_l *and* β_l.

L^{fin} contains the finite plays defined by a test table, e. g., if a strong repetition is avoidable. It is also possible to select a separation, s. t. α_l is the path into a strong repetition, and β_l is the strong repetition. There is only a finite amount of strong repetitions in a GTT, thus the set union is finite. Moreover, in the

separation $\widehat{\mathrm{SP}}^k$, β represents the block (or row) that is strongly repeated, which is implicitly described in Lemma 5.21. This connects the β_l with the construction T_*. Lemma 5.19 is similar to the separation theorem of ω-regular languages.

Further, we define the languages $V_C, V_S \subseteq \widehat{\mathrm{SP}}^k$ that represent the words in $\widehat{\mathrm{SP}}^k$ that are violated by the turn of the challenger or the system, respectively.

Definition 5.20. *Let* SP *be a language over* (ϕ, ψ)*, and* $p \in (\mathcal{I}, \mathcal{S})^*$ *a play, and* $k > 0$*:*

$$V_C(\mathrm{SP}, p, k) = \{w \in \widehat{\mathrm{SP}}^k \mid (\phi, \psi) = w[k-1] \wedge p \not\models \phi\} \tag{5.10}$$

$$V_S(\mathrm{SP}, p, k) = \{w \in \widehat{\mathrm{SP}}^k \mid (\phi, \psi) = w[k-1] \wedge p \not\models \phi \wedge \psi\} \tag{5.11}$$

We can prove our Lemma 5.19.

(Proof by induction over k*).* For $k = 0$, the separation (Lemma 5.19) follows immediately from the definition of SP. In particular, $\widehat{\mathrm{SP}}^0$ is always regular language.

For $k > 0$, in every round $\widehat{\mathrm{SP}}^k$ is filtered by the current play p with $|p| = k$, formally $\widehat{\mathrm{SP}}^k = (L^{fin} \cup \widehat{\mathrm{SP}}^{k-1}) \setminus V_S(\mathrm{SP}, p, k)$. From the induction hypotheses, we know that $\widehat{\mathrm{SP}}^{k-1}$ can be separated, and by definition of the game, the words in $\widehat{\mathrm{SP}}^{k-1}$ have at least the length k, otherwise the game has terminated. Assume w. l. o. g. that every α_l in the separation of $\widehat{\mathrm{SP}}^{k-1}$ contains only words with length k, otherwise we would unwind β many times as needed. Therefore if we remove the violated words $V_S(S, p, k)$ of the system (or the challenger), we adhere the separation:

$$\widehat{\mathrm{SP}}^k_\infty = (L^{fin} \setminus V_S(S, p, k)) \cup \bigcup_l (\alpha_l \setminus V_S(S, p, k)) \cdot \beta_l^\omega \tag{5.12}$$

$$\widehat{\mathrm{SP}}^k_* = (L^{fin} \setminus V_S(S, p, k)) \cup \bigcup_l (\alpha_l \setminus V_S(S, p, k)) \cdot \beta_l^*(\Phi_l, \texttt{false}) \tag{5.13}$$

□

We established a relation between both sets, which follows from the Definition 5.16.

Lemma 5.21 (Coupling of $\widehat{\mathrm{SP}}^k_\infty$ and $\widehat{\mathrm{SP}}^k_*$)**.** *There exists a separation of* $\widehat{\mathrm{SP}}^k_\infty$ *and* $\widehat{\mathrm{SP}}^k_*$*, s. t. both* L^{fin} *are equal and*

$$\forall l \,.\, \alpha_l \cdot \beta_l^\omega \in \widehat{\mathrm{SP}}^k_\infty \Leftrightarrow \alpha_l \cdot \beta_l^*(\Phi, \texttt{false}) \in \widehat{\mathrm{SP}}^k_*,$$

We show the coupling between both sets is maintained after each round, additionally, we prove a bit more: The coupling is maintained after the turn of the challenger and the system.

(Proof by induction over k). The lemma is immediately valid for $k = 0$.

Let $\widehat{\mathrm{SP}}_{\infty}^{k-1}$ and $\widehat{\mathrm{SP}}_{*}^{k-1}$ be coupled, we need to show that the coupling is ensured after the round k for $\widehat{\mathrm{SP}}_{\infty}^{k}$ and $\widehat{\mathrm{SP}}_{*}^{k}$. We also assume an arbitrary play $p \in (\mathcal{I}, \mathcal{S})$ s. t. $p = p' \cdot (a, b) \wedge |p| = k$ and there is no winner of the play in all previous rounds $k' < |p|$, otherwise the game would have terminated (both sets are empty and the coupling relation would immediately hold).

The coupling is *stable* under set difference: Stable means that the coupled $\alpha_l\beta_l^{\omega}$ and $\alpha_l\beta_l^{*}(\Phi_l, \texttt{false})$ will stay coupled in all rounds. In detail: By induction hypotheses the languages of the finite words in $\mathrm{SP}(T_{\infty})$ and $\mathrm{SP}(T_{*})$ are coupled, i. e., $L_{\infty}^{fin,k-1} = L_{*}^{fin,k-1}$. Applying the set difference keeps equivalence:

$$L_{\infty}^{fin,k-1} \setminus V_{k,\xi} \;=\; L_{\infty}^{fin,k} = L_{*}^{fin,k} \;=\; L_{*}^{fin,k-1} \setminus V_{k,\xi}.$$

The same holds for any coupled $\alpha_l\beta_l^{\omega}$ and $\alpha_l\beta^{*}(\Phi_l, \texttt{false})$ of the separation. W. l. o. g. we can pump up the prefix α_l with expanding iterations from β_l s.t. all words $w \in (\alpha_l \cdot \beta_l)$ are at least equal to k.

Leaving one open case, $\alpha_l(\Phi, \texttt{false})$ (we choose $\varepsilon \in \beta^{*}$). Obviously, this word is removed by subtracting with $V_S(\mathrm{SP}, p, k)$ (ψ at position k is `false`), but this is not valid for the challenger filter $V_C(\mathrm{SP}, p, k)$. Here the following situation might be possible, that $\alpha_l\beta_l^{\omega} \subseteq V_C(\widehat{\mathrm{SP}}_{\infty}^{k-1}, p, k)$, but $\alpha_l(\Phi_l, \texttt{false}) \not\subseteq V_c(S_{*}^{k-1}, p, k)$. Note that β_l represents the corresponding block for which Φ_l is built for. By definition, Φ_l is the disjunction of all ϕ of the first symbols in β_l. Therefore, the play p violates Φ_l iff it violates all *first* ϕ in β. Therefore, the situation can not appear. □

Proposition 5.17. From coupling in Lemma 5.21 it follows that $\widehat{\mathrm{SP}}_{\infty}^{k} = \emptyset \Leftrightarrow \widehat{\mathrm{SP}}_{*}^{k} = \emptyset$. □

Example 5.22. *For better comprehensibility, we show the equivalence of the conformance on the special case with*

$$T_{\infty} = \langle \phi, \psi \rangle^{\omega} \qquad T_{*} = \langle \langle (\phi, \psi) \rangle^{-}, \tag{5.14}$$

$$(symIn, \texttt{false}) \rangle. \tag{5.15}$$

The row $(\phi, \texttt{false})$ *prevents the strict conformance of the system—the system cannot fulfill* `false`*, the same holds for the strong repetition in* T_{∞}*. Therefore, we only need to consider* weak *conformance. Let* $p \in (\mathcal{I}, \mathcal{O})^{*}$ *an arbitrary play. The*

#	ASSUME In	ASSERT Out	⏲
1	—	=p	1

Figure 5.3: A GTT with a global variable p, without any compliant system.

#	ASSUME In	ASSERT Out	⏲
1	=p	=p	1

Figure 5.4: A GTT with a global variable p specifying the identity function for the first cycle.

system wins in T_∞ if $p \not\models \phi$. At the same time it would win in T_*, as p is not allowed in both rows. Otherwise, the system loses in T_∞ ($p \models \phi$ and $p \not\models \psi$) if and only if it also violates T_* as none of both rows are adhered, i. e., $p \not\models \psi \vee p \not\models$ `false`.

5.4.2 The Issue with Universally Quantified Output

Let's investigate the GTT given in Figure 5.3. This GTT contains one row where the only constraint is an assertion to the system: the output is equal to a global variable. This simple GTT has a surprising property:

Proposition 5.23. *There exists no system weakly or strictly conforms to the GTT in Figure 5.3.*

Proof. For the proof, we assume the opposite. Let S be system conforming the GTT T in Figure 5.3. Note that S is a deterministic system by our assumption at the beginning of Chapter 5. We unfold the semantics for this special case.

Then, S conforms to every instantiated GTT $T[p/x]$ where p is replaced by the value $x \in \mathbb{N}$. W. l. g. we pick the cases $p = 1$ and $p = 2$: As S conforms $T[p/1]$, the output of S in the first cycle is 1 for any given input `In`. Analogously, for $p = 2$, where the output is 2.

This is a contradiction to the assumption of a deterministic system, which describes the output is determined by observed input values. In our cases, the behavior $\mathrm{B}(S)(i) = 1$ and also $\mathrm{B}(S)(i) = 2$ for all $i \in \mathbb{N}$. □

This property is a common pitfall when using GTTs, and it is valid for all conformance definitions. The expected specification is that the global variable p is bound to the output of the system. Instead, the universal quantification is instantiated into different separated games, and each of them has to be won by the system.

Interestingly, the similar looking GTT in Figure 5.4 does not have this issue. The difference is that the global variable is restricted by the challenger beforehand. The challenger differs from the system in two relevant points. The strategy of challenger allows picking arbitrary values, where the strategy of the system is completely determined by the input. Also, the challenger loses by playing an invalid turn in almost all plays, thus, the game is highly asymmetric in favor of the challenger. To disprove the conformance, the challenger only needs to find one violation of the system to win, whereas the system needs to win (or not to lose) in *all* plays. As a simple rule, a global variable should always be mentioned in an assumption before it is used in an assertion.

Existential quantification. The introduction of existential quantification in GTTs seems to be a solution candidate for these issues. Reconsider the GTT T in Figure 5.3, but now with existentially quantify p. Therefore, a system S conforms to T if and only if there exists an instantiation $x \in \mathbb{N}$ for p, such that S conforms to $T[p/x]$. But our problem is still not solved. Naively, we set the instantiation of p to be $\mathrm{B}(S)(i)$, but the variable i is free and represents the input of the challenger. As the input is nondeterministic selected by the challenger, we need to find an instantiation that is valid for all inputs i. Unfolding the constraints in the considered GTT, we can express the situation as follows:

$$\exists x.\forall i.x = \mathrm{B}(S)(i)$$

The formula clarifies the problem: We need to select an instantiation of the output independently of the input of the environment. Note that this issue can be solved for a limited number of cases by using universal and existential quantification together. For our considered case, we introduce a new global variable q for constraining the input `In` of the challenger. Then expressed as a formula, the situation would be

$$\forall q.\exists x.\forall i.q = i \wedge x = \mathrm{B}(S)(i) \quad .$$

By introducing an additional universal quantification, we can switch the alternation of the quantifiers. This approach requires limitations on the GTTs, e. g., no global variables given in repeated rows because then the challenger input of the second row iteration can depend on the output of the system from the previous iteration. For a bounded time constraint, this can be fixed by unrolling.

Assignment. The next possible solution requires a new feature in GTTs: assignments. Assignments introduce an explicit state to the GTTs by introducing variables. These variables are assigned locally after the turn of the challenger

#	ASSUME	ASSERT		⏲
	In	Out	p	
1	—	—	$:= Out$	1
2	p	—		1

(a) A valid GTT with an assignment

#	ASSUME	ASSERT		⏲
	In	Out	p	
1	—	—	$:= Out$	[0,1]
2	—	—	$:= Out + 1$	[0,1]
3	p	—		1

(b) Demonstrating nondeterministic assignments

Figure 5.5: Two GTTs with a GTT-local variable p, which can be assigned.

or the system. Therefore, assignments are always assigned under the right quantification: after the universal quantification of the challenger input, and the existential quantification of the system. Thus, they do not suffer under the order of instantiation.

Also, the presentation in the table would be rather simple. We allow columns for each local GTT variable under both column categories. Figure 5.5(a) shows a small example, where p becomes a local variable, which binds the value of the first output in Row 1. The value is later used in Row 2 for constraining the input.

But assignments bring also drawbacks to the semantics and understandability. Both arise from the nondeterminism in the selection of the current row. Consider the example in Figure 5.5(a). Let the duration constraint on Row 1 be "don't-care" ("—"), then the first row is skippable. Thus, Row 2 may become active with an unassigned variable p. Such a GTT is not a well-defined specification. This issue is fixable adapting the semantics. The understandability suffers under the same circumstance. It may become hard to figure out how the variables are bound in certain rows, especially, if repeatable (and skippable) rows and row groups occur together.

Another issue shows that assignments cannot be handled by the current game representation. Consider the table in Figure 5.5(b). This GTT has initial rows which assign a different value to p. In each round of the game, the challenger chooses the input value. The input constraints of both rows are satisfied with any value for `In` and `Out`. Thus, both rows survive the first and second filter pass. Now, we have to assign the specification-local variable p accordingly to two different assignments. The game needs to nondeterministically split on both possible assignments, leading to separated plays in the next round: one with $p = Out$ and one with $p = Out + 1$. Moreover, we need to decide whether the conformance requires that all or at least one separated play needs to be won by the system. Besides this definitional question, this also requires large changes in our decision procedure (Chapter 6).

5.4.3 Arbitrary Repetition in Last Row.

Our choice of semantics leads to an unintuitive case. Let us consider the meaning of the time constraint $[1, 2]$ on the last row of a GTT (cf. Figure 4.10(a)). In our informal semantics (Section 4.4), we notice that the repetition has no effect: If the system adheres to the row once, it is conforming. Only the lower bound on repetition matters in the last rows. As a consequence, if the repetition in the last row allows zero repetition (row is skippable) the last row is completely ignored. For this reason, strong repetition is introduced to forbid early termination of the play by the choice of the system. For engineers, this circumstance seems strange, and a possible solution is either to replace all arbitrary repetition by their lower iteration limit, or by strong repetition.

Chapter 6

Decision Procedures

This chapter is dedicated to two decision procedures for the verification of our conformance definition for GTTs (Definition 5.12). The first decision procedure exploits model-checkers for invariants and LTL-properties to prove the weak, strict, or cooperative conformance. The second decision procedure translates our verification subject and the given GTT into a C-program and uses modern Horn-based solvers for the verification.

Outline. This chapter is split into two sections. First, we present the decision procedure with the model-checker (Section 6.1). For this procedure, we introduce the notion of *normalized tables* and the automata construction, which corresponds to the defined game. These definitions are also exploited for the verification via the translation into a C-program (Section 6.2). Finally, we visit some open questions, especially transformation of the verification subject, to build the verification pipelines (Section 6.3)

6.1 Model-Checking for Conformance

Based on the state-of-the-art model checker NUXMV [Cav+14], we present an automatic decision procedure for conformance checking (Figure 6.1). The decision procedure translates the given reactive program, given as STRUCTURED TEXT, and the GTT into a combination of two automata, such that the conformance can be checked either as an invariant or LTL property on them. The program automaton describes the transition relation between two cycles and is obtained from the PLC source code using symbolic execution and a transformation into the single static assignment form [RWZ88]. The GTT is translated into an automaton in which automaton states keep track of the current table rows—these

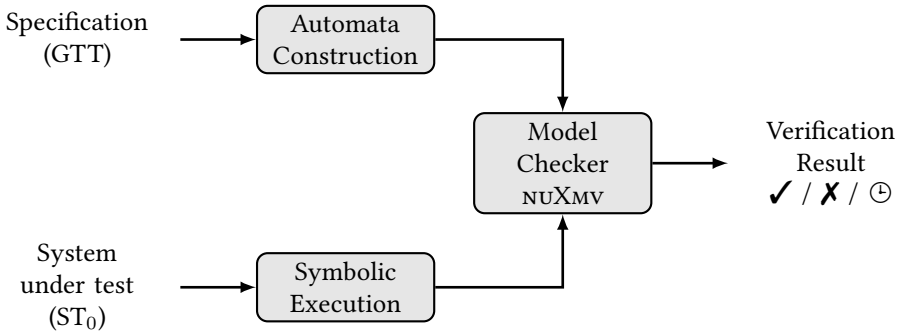

Figure 6.1: A schematic view of the procedure deciding whether system under test conforms to the given GTT. PLC software is preprocessed into a simplified version of Structured Text: Structured Text 0. The result of the conformance check is either (a) that the software indeed conforms to the GTT, (b) a counterexample, or (c) a time out due to limitations on resources.

also correspond to the set of remaining symbolic plays $\widehat{\mathrm{SP}}$. The program conforms to the GTT if this automaton either reaches the end state or if no automaton state is active anymore ($\widehat{\mathrm{SP}}$ is empty by the choice of the challenger). It violates the specification if the output values violate the output variable constraints in all currently active automaton states.

The focus of this section is on the construction of the automaton, and its acceptance condition which corresponds to the conformance of GTTs.

6.1.1 Transforming Generalized Test Tables into Automata

Normalized table. In order to ease the presentation of this construction, we assume a normalized form of test tables which allow only a restricted form of time constraints:

Definition 6.1 (Normalized GTT)**.** *A GTT $T \in \mathbb{T}$* normalized *if every time constraint in T is any of $[1,1]$, $[0,1]$, $[0,\infty]$ or ω.*

The construction of such a normalized table T_0 for T is canonical: Every row with a finite duration interval $\tau = [a,b]$ is unrolled into b rows with the first a repetitions having duration $[1,1]$ and the remainder having duration $[0,1]$. If $\tau = [a,\infty]$, then the row is repeated a times with duration $[1,1]$ and one row with duration $[0,\infty]$. Strong repeated rows are simply transferred. This scheme is also applicable to row groups, where the complete row group is duplicated. Note that a single row in a GTT (in row group nesting) can be expanded in

	I	O	DURATION
1	...	...	$[0, \infty]$
2	...	...	$[1, 1]$
3	...	...	$[0, 1]$
4	...	...	$[1, 1]$

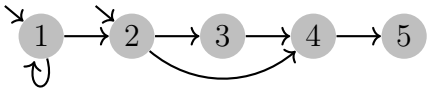

Figure 6.2: A normalized table and the successor relation on its rows.

exponential many duplicates, i. e., a row with $\tau_0 = [a, b]$ nested in n row groups with time constraint $\tau_i = [a_i, b_i]$ $(1 \leq i \leq n)$ is exploded into $b\Pi_{1 \leq i \leq n} b_i$ rows.

Figure 6.2 illustrates the row successor relation (right) for a normalized GTT (left). The syntactical restriction of normalized tables does not pose a limitation on the expressiveness of GTTs. We make the following summarizing observation:

Proposition 6.2. *For every GTT T there is a semantically equivalent normalized GTT T_0.*

Alphabet. We construct a Büchi automaton, which accepts ω-words in $(\mathcal{I} \times \mathcal{S})^\omega$ produced by a reactive system (Section 2.1). The alphabet of the automaton is defined over the values of input $\mathcal{I}$ and state variables $\mathcal{S}$ of the reactive system. In the following, we use Boolean formulas to describe subsets of the alphabet.

States. The idea behind the normalization is that every table row of the normalized table is also a state in the automata. A normalized GTT T_0 with m rows results in an automaton with 2^{m+2} states. Thus, the constructed automaton simulates the power automaton from the automaton shown in Figures 6.2 and 6.3. The states are characterized by vectors $(s_1, \ldots, s_{m+1}, \mathit{fail})$ of Boolean variables, one for each row in T_0 (s_1 to s_m), one indicating termination s_{m+1}, and one indicating failure (fail). Intuitively, s_k is true in a state if and only if the table is in a situation where the test table may have been executed by the trace up to the k-th row. The initial state $(s_1^0, \ldots, s_{m+1}^0, \mathit{fail}^0)$ is defined by

$$s_k^0 = \text{true iff } k \in \mathit{succ}(0) \text{ and } \mathit{fail}^0 = \text{false} \ . \tag{6.1}$$

The relation $\succ$ is defined in Definition 5.8.

State Transition. Given a state $(s_1, \ldots, s_{m+1}, \mathit{fail})$, then its successor state $(s'_1, \ldots, s'_{m+1}, \mathit{fail}')$ is deterministically computed according to these equiva-

lences:

$$\bigwedge_{k=1}^{m} \Big(s'_k \leftrightarrow \bigvee_{i=1}^{m} (s_i \wedge k \in succ(i) \wedge \phi_i \wedge \psi_i) \Big) \tag{6.2}$$

$$s'_{m+1} \leftrightarrow \Big(s_{m+1} \vee \bigvee_{i=1}^{n} (s_i \wedge m{+}1 \in succ(i) \wedge \phi_i \wedge \psi_i) \Big) \tag{6.3}$$

$$\mathit{fail}' \leftrightarrow \Big(\mathit{fail} \vee \big(\bigvee_{i=1}^{m} (s_i \wedge \phi_i \wedge \neg\psi_i) \wedge \bigwedge_{i=1}^{m+1} \neg s'_i \big) \Big) \tag{6.4}$$

The equivalences in (6.2) encode that the k-th row is active in the next step (variable s'_k) if there is an active row i preceding k such that its input constraint ϕ_i and output constraint ψ_i are satisfied. The same applies to the virtual row $m+1$ at the end of the table in (6.3). Here, additionally, once true, the variable s_{m+1} never falls back to false again. The fail flag indicating a specification violation is defined in (6.4). It is triggered whenever there is one active row i such that its input constraint ϕ_i is satisfied while the output constraint ψ_i is violated and there is no other active row in the next state. Note that the equivalences above ensure the state transition system is always deterministic.

Acceptance Condition. The acceptance condition remains to be described. By definition, a Büchi automaton accepts an infinite word if at least one state from the set of final states is traversed infinitely often by the given word. We construct three different sets of accepting states: condition A_{WC} for weak conformance, A_C for strict conformance, and A_{Coop} for cooperative conformance. The following formulas are identified with the set of states that satisfy them:

$$A_{WC} := \neg \mathit{fail} \tag{6.5}$$

$$A_C := \Big(\bigwedge_{i=1}^{m} \neg s_i \wedge \neg \mathit{fail} \Big) \vee s_{m+1} \tag{6.6}$$

$$A_{Coop} := s_{m+1} \tag{6.7}$$

For weak conformance, the automaton accepts any trace that never set the flag fail to true. For strict conformance, the automaton accepts a trace if it reaches a state in which s_{m+1} (the flag for finishing a table) is true or there is no active table row anymore without a previous violation of the system. The cooperative conformance simply requires that the end of a table is reached, indicated s_{m+1}. Due to the construction of the automaton, the acceptance condition of A_{WC} can be ensured with an invariant, whereas the acceptance condition A_C and A_{Coop} are reachability conditions. Strict and cooperative conformances require a *fair* challenger which infinitely often tries to make progress. For example, if a row can be repeated infinitely (like Row 1 in Figure 6.2), the challenger

could choose inputs only adhering to the input constraint from the first row and preventing the system from making the required progress and finally reaching the end of the table (and so a state where s_{m+1} is true). Therefore, we require that the challenger enforces progress in the test protocol from time to time. Such a *fair* challenger is described by following the LTL formula:

$$fair_C := \square \left(s_i \rightarrow \bigvee_{r \in succ(i)} \square\Diamond \phi_r \right) \quad , \tag{6.8}$$

Thus, $fair_C$ requires, when the ith table row is active (s_i is true), that the challenger needs to pick an input adhering to one of its successor assumptions from time to time. For strict conformance, a fair challenger seems unnecessary, but the play can also become stuck: the end of the table is not reachable (like cooperative conformance), and the challenger only emits valid inputs. In this situation, the game is not winnable for the system, thus it cannot strictly conform.

Example 6.3. *Figure 6.3 sketches the construction principles for the* GTT *in Figure 4.9 by visualizing the state vector as an automaton. Note that final encoded automaton is the power automaton of the depicted automaton, where the state vector consists of eight independent Boolean variables representing the current table rows in the normalized* GTTs*.*

In particular, an automaton state $s_i^{(k)}$ in Figure 6.3 represents the kth iteration of the ith row of the table and expresses that the ith row is currently a possible step of the test protocol. The state $fail$ is activated when a violation of a row assertion (of the system under test) occurs, state s_{m+1} represents the end of the table. If this state is reached, the system conforms to the GTT*.*

There are two kinds of transitions: An α edge from a state $s_i^{(k)}$ to the state $fail$ is triggered if the assumption of the ith row is satisfied, but the assertion of the same row is violated (6.4)*. A β edge is taken when both the assumption and the assertion hold, leading to the next possible steps in the test protocol, defined by the successor relation $succ$;* (6.2) *and* (6.3)*. Note that due to the strong-repeated row group in Figure 4.9, the end-of-the-table and thus the final state s_{m+1} is not reachable. We model this situation by labeling the edge to s_{m+1} with the contradictory guard* `false`*.*

The automata. Based on the above constructions, we define three Büchi automata. They share the states, initial states (6.1), and the transition function (6.2)–(6.4), but have different acceptance conditions. The automaton $\mathcal{A}_{WC}$ for weak conformance uses the accepted condition A_{WC} as a set of accepting states, the one for strict conformance uses A_C, and for cooperative conformance A_{Coop}.

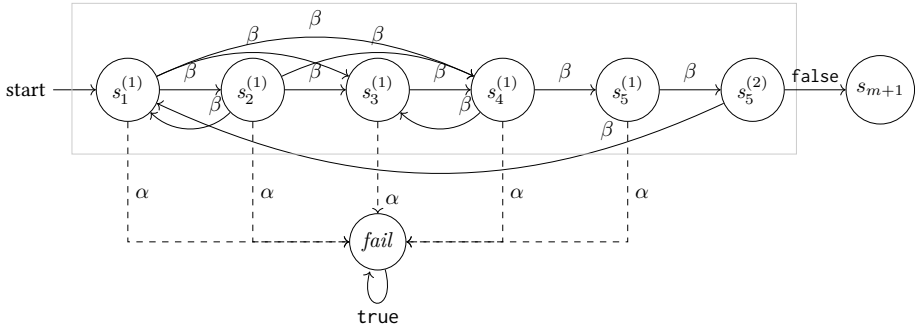

Figure 6.3: Sketch of the automaton generated for the GTT of Figure 4.9

Proposition 6.4. *Let T be a normalized GTT (Definition 6.1). A reactive system S weakly conforms to T iff all traces of $\mathcal{B}(S)$ are accepted by $\mathcal{A}_{WC}(T)$, i. e., $\mathcal{B}(S) \subseteq \mathcal{L}(\mathcal{A}_{WC}(T))$. A reactive system S strictly conforms to T iff all traces of $\mathcal{B}(S)$ are accepted by $\mathcal{A}_C(T)$, i. e., $\mathcal{B}(S) \subseteq \mathcal{L}(\mathcal{A}_C(T))$. A reactive system S cooperatively conforms to T iff all traces of $\mathcal{B}(s)$ are accepted by $\mathcal{A}_{Coop}(T)$, i. e., $\mathcal{B}(S) \subseteq \mathcal{L}(\mathcal{A}_{Coop}(T))$.*

The following proofs bases on the algorithmic presentation of the game as presented in Figure 5.1. We argument, that the specific early exits of the game are coupled with the state of a parallel running automaton. An early exit denotes an either invalid move by the challenger or the system (Lines 14 and 10), or the end of the table is reached (Line 10) For convenience reason, we assume, that we are in an arbitrary round $k \leq 0$, thus, all played turns in rounds $k' < k$ were not settled. In the core, this is a short form of an induction proof over the rounds k.

Weak Conformance. Consider the direction from left to right: Let the reactive system S weakly conform the GTT T, then for any round the game with $k' \in \mathbb{N}$ and current play $v \in (I \times S)^{k'}$, we know that game has not terminated in favor of the challenger. Also, as the system is weakly conform, the Line 14 in Figure 5.1 is not reachable, i. e., at least assertion constraint ψ is always adhered. Therefore, there exists a word $(\psi_{r_0}, \ldots, \psi_{r_k})$ over the assertion of the rows r_i in T such that each assertion constraint ψ_{r_i} hold in round $i \leq k$ with respect to the successor relation $r_{i+1} \in succ(r_i)$. Thus, the forbidden state $fail$ could not have been reached in the automaton.

Consider the case, where the current play v is longer than the number of rounds k $|v| > k$, then the play was decided in an earlier round. By assumption of weak conformance, either no row state s_i ($1 \leq i \leq m$) is active anymore, or s_{m+1} is active (end of specification reached). In both cases, the current play is accepted (and $fail$ is not reachable anymore).

The same argumentation is valid in opposite direction. If we reached the forbidden state, we can construct a set of word pairs over the assumptions $(\phi_{r_0}, \ldots, \phi_{r_k})$ and the assertion $(\psi_{r_0}, \ldots, \psi_{r_k})$, s.t. the assumption hold in every turn, and the last assertion ψ_{r_k} is violated by the system. Each entry represents a counter-example for the remaining symbolic plays in $\widehat{\mathrm{SP}}$ in round k. □

Strict Conformance. The argumentation follows the scheme of the weak conformance, but instead of unreachability of lines, we know, that either the Line 17 or Line 10 in Figure 5.1 have been reached. Each of the reached lines is a reason why the system wins, and therefore, also why A_C holds.

Case Line 10: There are no active row states ($\neg s_r$, for all table rows r) which denotes an invalid turn of the challenger. Case Line 17: The end of the table has been reached, thus s_{m+1} is active. This is mutual exclusive with fail by definition (6.4) if the system has violated a constraint.

The opposite direction can be reasoned with similar arguments in contraposition, then the system loses due to the violation of the last possible assertion constraints ψ_i. Thus, fail is active and A_C could not have been reached. □

Cooperative Conformance. The proof follows immediately from the connection of s_{m+1} to the Line 17 in Figure 5.1. □

Extension for Past References and Global Variables. The automata construction described above does not cover GTTs with back-references of the form $v[-k]$. To handle back reference, the state space needs to be enriched by additional variables. For any input or output variable v for which a back-reference $v[-k]$ occurs in a table, the state variables $v_1, \ldots, v_k$ are added. Moreover, the following equivalencies are added to the state transition:

$$v_1' = v \ \wedge \ \bigwedge_{i=2}^{k} (v_i' = v_{i-1}) \tag{6.9}$$

The expression $v[-c]$ then refers to the variable v_c for any constant $c \in \{1, ..., k\}$. Note that the initial value for each v_i is undefined (cf. Definition 5.10). The same construction is applied for each global variable g. As global variables have the same value in all states, we add the simple equality $g' = g$ to the state transition. Also, their initial state is undefined. Therefore, the automaton must accept the observed trace for any instantiation of the global variables and also for any unset back reference.

We know that model-checking of LTL formulas by using Büchi automaton is decidable ([Büc90]). Therefore, we state the following:

Theorem 6.5. *The weak, strict, or cooperative conformance of a reactive Boolean program to a GTT is decidable.*

The Boolean programs are programs that are expressible as an arbitrarily large circuit. This restriction is necessary as they can be expressed as automata. For the PLC domain, this is a rather minor limitation, which enforces an upper bound on the state space and executed statements in all execution cycles. Both constraints are already satisfied by real-world PLC programs. Their language specification IEC 61131-3 forbids dynamically allocated memory and loops should always have an upper bound due to the real-time requirements.

To create a decision procedure, we translate the GTT (as described above), and the program to an automaton (as described below). Both automata are combined into a product automaton, in which the input, state, and output variables from the program automaton are the input for the GTT-automaton. And the input of the product automaton is nondeterministically chosen values for the input variables of the verification subject.

For strict and cooperative conformance, we use the LTL property $fair_C \rightarrow \Diamond A_C$ (A_{Coop}, respectively), which enforces that the required condition is finally met under a fair challenger. As the acceptance condition for weak conformance A_{WC} is expressible as an invariant. We use faster verification techniques IC3 [Bra11].

6.2 Horn-based Verification via C-program Verifier

The next decision procedure is based on a translation to a C-program and the use of state-of-the-art Horn-based decision procedure for C-programs, like [HR18; Gur+15]. But of course, any other C-verification which can handle assumptions and assertions, like CBMC [KT14] or the CPACHECKER [BK11], should also be usable. We especially investigate Horn-based solvers because they provide fully automatic reasoning without the necessity of auxiliary specifications. An additional advantage is their support for unbounded integers which bring better performance for complex arithmetic. Multiplication and division explode when encoded into a propositional formula, however, this is required for SAT-based methods as our model-checker pipeline. Besides the Horn solver backend, the C-program verifier also provides additional program analyses which further increase the pipeline performance. Also, they are capable of dealing with unbounded loops. Therefore, we spare the symbolic execution and directly translate the PLC program into a C-program.

Of course, there is a drawback of this pipeline: it is only capable of proving the weak conformance. The other conformances are not expressible as an invariant.

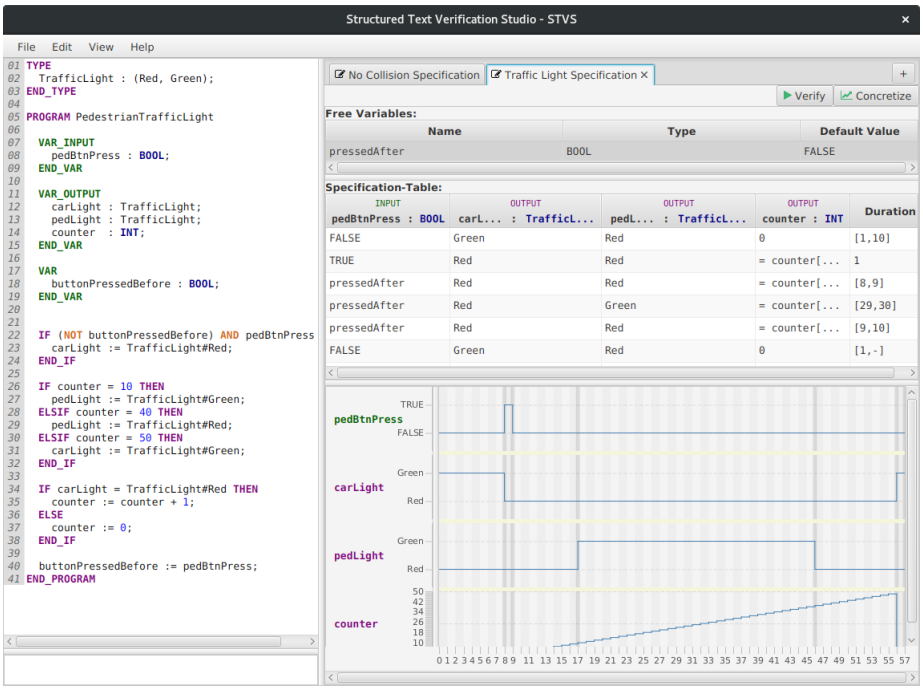

Figure 6.4: Screenshot of STVS, a graphical frontend for the verificaton with GTTS, showing the ST source code on the left, and the counter example as line diagram and a concrete test table on the right side.

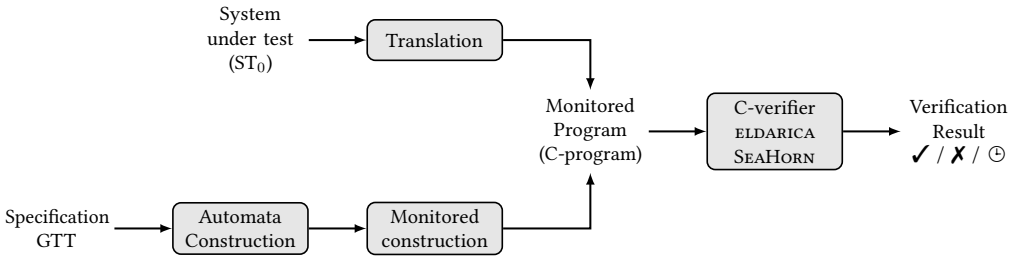

Figure 6.5: Horn-based verification pipeline for GTTS

Figure 6.5 shows the pipeline for this decision procedure. The decision procedure takes the system under test and a GTT, and combines them into a monitored program. The monitored program simulates a parallel synchronous execution of the given system and the Büchi-automata similarly to the product automaton from the previous decision procedure. We call the program, which encodes the Büchi automata and decides the correctness, tester. In the last step, we use ELDARICA (a horn-solver which supports C programs) or SEAHORN (C-program verifier) to verify that the monitored program never signals *fail*. The verifier returns *safe* (✓) if the program adheres to the given specification via assumption and assertion statements in the C-program. Otherwise, *unsafe* (✗) denotes that there exists a program execution leading to a fail state. Again, timeouts (🕒) are possible. *Safe* denotes that the system under test weakly conforms to the given GTT. Internally ELDARICA and SEAHORN translates the given C program into a set of constraint horn clauses (CHC), and solves the set of clauses by finding a satisfying interpretation of the declared variables and functions. Additionally, SEAHORN as dedicated C-program verifier provides a rich feature, e. g., the inference of loop invariants via abstraction interpretation, for the program analysis.

Language of C-programs. The constructed monitored C-program requires three additional statements, which are well-known to the verification community, but does not exist in the C-language.

- The statement `assume expr;` expresses an assumption during the program execution. If the assumption is violated in the current execution path, the program execution is discarded and not further investigated.
- The opposite is `assert expr;`. If a program execution reaches this statement, we require that the given expression holds, otherwise, we consider the program as incorrect, and the verification of this program fails.
- The statement `havoc var;` overrides the given variable with a nondeterministically chosen value. A *havoc'd* variable has any possible value allowed by its data-type, identical to a declared but not initialized local variable in C.

For this section, we introduce the notion of a *reactive fragment* which is a lightweight structure for a reactive program:

Definition 6.6. *A reactive fragment* $(InVar, StateVar, Init, Trans)$ *is a four tuple, where* $InVar$ *is the set of input variables,* $StateVar$ *the set of state variables,* $Init$*, and* $Trans$ *are C-procedure bodies.*

First, a reactive program is defined for a set of program variables, the input and state variables. These variables also define the program state. This program state needs to be initialized at the start of the reactive system. The initialization is described by the provided $Init$ procedure, and procedure $Trans$ is the reactive program (Section 5.1) describing the state transition of a single cycle of the reactive system.

We can construct such a reactive fragment for a given PLC program. For this, we need to establish a version of the system under test, where all subroutines are embedded (Section 6.3). We need to care about semantical differences between C-language and IEC 61131-3 on expression and statements evaluation. For example, whereas the C semantics describe which behavior is expected for integer arithmetic, the IEC 61131-3 left these semantics implicitly undefined. For floating-point arithmetic, both standards follow IEEE 754. The IEC 61131-3 does not specify the behavior if an error occurs, e. g., a division by zero, whereas the C standard explicitly leaves such cases unspecified and open to the compiler implementation. Additionally, the verification tool can have a different semantical interpretation of the C-program. In our case, this occurs with an SMT-based solver which uses the built-in mathematical theories. For example, the C-program verifier often uses unbounded integer arithmetic in contrast to the bit arithmetic used by the language standards.

The remainder of this section is split up into three parts: Encoding of the automaton (Section 6.2.1) into a reactive fragment, construction of a monitored program given two reactive fragments (Section 6.2.2), and correctness of this pipeline (Section 6.2.3)

6.2.1 Construction of the Tester

The tester denotes a reactive fragment, which simulates the Büchi automaton $\mathcal{A}_{WC}$ and thus simulates the game (Figure 5.1). The tester takes the input values, chosen by the challenger, the state (including the output values), computed by the system, and asserts that the condition for the weak conformance A_{WC} holds.

The tester is a reactive fragment $T_{WC} = (InVar_t, StateVar_t, Init_t, Trans_t)$. Given a normalized GTT T_0, its Büchi automaton $\mathcal{A}_{WC}(T)$, and the signature Σ of the program under test, then we construct the reactive fragment as follows: The input variables $InVar_t$ of the tester are the program variables Σ (input, state and output variables) of the verification subject. The state variables $StateVar_t$ corresponds to the state vector of the Büchi automaton. In particular $StateVar_t$ contains Boolean variables for each table row $\mathtt{s}_1, \ldots, \mathtt{s}_{\mathtt{m}+1}$, and the variable $fail$ marking the failing of the system. For each program variable v, which has a past reference with a maximum lookbehind of $k_v \geq 1$, $StateVar$ contains k_v variables $\mathtt{v}_i$ ($1 \leq i \leq k_v$) with the same datatype as v. And each global variable

g, becomes a variable in $StateVar_t$.

$$
\begin{aligned}
InVar_t :=& \Sigma \\
StateVar_t :=& \{\mathtt{s_i} : \mathtt{boolean}, \ldots, \mathtt{s_{m+1}} : \mathtt{boolean}, \mathtt{fail} : \mathtt{boolean}, \} \\
& \cup \{\mathtt{v}_i^j : t_v \mid 1 \leq i \leq k_v \text{ and every past-referenced } v \text{ with type } t_v\} \\
& \cup \{\mathtt{g_1}, \ldots, \mathtt{g}_k\}
\end{aligned}
\tag{6.10}
$$

The initializer $Init_t$ takes care of the correct initialization of the tester. It corresponds to the initial state of the Büchi automaton:

$$
\begin{aligned}
&Init_t := \\
&\quad \mathtt{s}_i = \mathtt{false}; && \textit{// for } 1 \leq i \leq m+1 \\
&\quad \mathtt{s}_i = \mathtt{true}; && \textit{// for } i \in succ(i) \\
&\quad \mathtt{fail} = \mathtt{false}; \\
&\quad \mathtt{havoc}\ \mathtt{v}_i; && \textit{// for every past-ref. variable } v \\
&\quad \mathtt{havoc}\ \mathtt{g_i}; && \textit{// for } 1 \leq i \leq k \\
&\quad \mathtt{assume}\ P_1(\mathtt{g_1})\ \mathtt{and}\ \ldots\ \mathtt{and}\ P_l(\mathtt{g_1}); &&
\end{aligned}
\tag{6.11}
$$

$Init_t$ contains the initialization of the initial and non-initial table row variables s_i, and the error variable $fail$. Also, we anonymize (havoc) the global variables and the variables v_i which store the history information of a program variable. Additionally, the value ranges for the global variable can be specified by assumptions, given in the predicates P_i.

The transition statements $Trans_T$ are simulating the transition of the automaton given in Equations (6.2) to (6.4). In contrast to the logical definition, the assignments are evaluated sequentially and not in parallel as assumed in the equations. Thus, we need to introduce temporary variables to store the active automaton states to avoid clashes. After the automaton states have been evaluated in $Trans_t$, we overwrite $\mathtt{s}_i$ with the value of the temporary variable $\mathtt{s}'_i$. Also, the invariant for A_{WC} is evaluated and is asserted, and finally, we maintain the history of the program variables where needed by simply shifting the values, like in (6.9). The complete transition $Trans_t$ looks as follows:

$Trans_t :=$

$$\texttt{boolean } \mathtt{s}'_{\mathtt{k}} := \bigvee_{i=1}^{m} (\mathtt{s}_{\mathtt{i}} \wedge k \in succ(i) \wedge \phi_i \wedge \psi_i);$$

// for each $1 \leq k \leq m$

$$\texttt{boolean } \mathtt{s}'_{\mathtt{m+1}} := \mathtt{s}_{\mathtt{m+1}} \vee \bigvee_{i=1}^{n} (\mathtt{s}_{\mathtt{i}} \wedge m{+}1 \in succ(i) \wedge \phi_i \wedge \psi_i);$$

$$\texttt{boolean } \mathit{fail} := \mathit{fail} \vee \left(\bigvee_{i=1}^{m} (s_i \wedge \phi_i \wedge \neg\psi_i) \wedge \bigwedge_{i=1}^{m+1} \neg s'_i \right);$$

`assert !fail;`

$\mathtt{v}^{\mathtt{j}}_{\mathtt{i}} := \mathtt{v}^{\mathtt{j}}_{\mathtt{i-1}}$; *// for each past-ref variable* v *and* $1 \leq i \leq k_v$

$\mathtt{v}_{\mathtt{1}} := \mathtt{v}$; *// for each past-ref variable* v

$\mathtt{s}_{\mathtt{i}} = \mathtt{s}'_{\mathtt{i}}$; *// for each* $1 \leq i \leq m+1$

6.2.2 Monitored Program

In this section, we describe how the verification subject given as a reactive fragment is combined with the tester (also a reactive program). The result is a reactive fragment, which is rendered as a sequential C-like program suitable as an input for a common C-program verifier.

Definition 6.7 (Construction of the Monitored Program)**.** *Given the verification subject* $S = (InVar_s, StateVar_s), Init_s, Trans_s)$*, and the generated reactive fragment for the tester* $T_{WC} = (InVar_t, StateVar_t), Init_t, Trans_t)$*, then the monitored program is defined as*

$$P_{S \times T_{WC}} = (InVar_s, StateVar_s \uplus StateVar_t,\ (Init_s\,;Init_t),\ (Trans_s\,;\mathtt{Trans}_t))$$

Note that the variable domains of both fragments (from the verification subject and the tester) are compatible by the construction of the tester because the variable domain of the subject is already reserved. The program variables of the verification subject S are the input variables of the tester T_{WC}. The procedures for the initialization and transition are simply sequentially executed after one another.

The monitored program $P_{S \times T_{WC}}$ behaves like the verification subject S, as long as the verification subject weakly conforms to the GTT from which the tester T_{WC} was generated. The tester does not interfere with the state variables of the subject. Thus, the only difference in behavior occurs when the assertion fails, and this is only the case for non-conformant programs.

We prefer the notion of *monitored* program over the notion *product* program. The concepts look similar, both concepts are a parallel execution of several programs, but product programs guarantees, that these parallel executions do not interfere which each other. Thus, a run of the product program can be projected to a run of every single original program. In contrast, a monitored program requires the interference of the tester: The tester needs to abort the execution if the verification subject is not correct. If the verification subject is run without the tester, the run would have continued.

Encoding into a C program Finally, we generate from the (monitored) reactive fragment $P_{S \times T_{WC}}$ a C program that simulates this fragment. The C-program is straight-forwarded and valid for every reactive fragment. In this section, we show the construction specialized on the monitored reactive fragment $P_{S \times T_{WC}}$.

In contrast to a C-program, a reactive fragment is periodically executed. For this reason, we have separated the initialization procedure $Init$ and the periodically executed procedure $Trans$ in the reactive fragment. The C-program template is given in Listing 6.6. First, the reactive fragment is wrapped inside the `main`-procedure (the entry point of a C-program), starting with the declaration of every variable v_i in $InVar$ and $StateVar$ with the appropriate type t_i. Afterward, the initialization procedure is executed—in our case $Init_s$ and $Init_t$. The periodic code $Trans_s$ and $Trans_t$ are infinitely often executed in a loop, simulating the periodical execution cycle. But before the execution of the transition code, every information of the input variables in $InVar_s$ needs to be removed with a havoc-statement to simulate fresh sensor information of the environment.

The generated C-program can be analyzed using verification tools that support C-programs. We use ELDARICA [HR18] and SEAHORN [Gur+15] for the analysis. Both tools translate the given C-program into a set of Horn clauses.

6.2.3 Correctness

We dedicate this section to the correctness of our Horn-based or C-verifier based verification pipeline. Correctness is narrowed by the following statements:

Proposition 6.8. *Given a reactive program S and a GTT T, the Horn-based pipeline (Figure 6.5) successfully terminates (proving that the assertion always holds) if and only if the reactive program weakly conforms the GTT T.*

We prove the correctness by showing that the monitored program, especially the tester, is coupled to the state of the Büchi automaton $\mathcal{A}_{WC}(T)$ running in parallel to the monitored program. As a consequence, we state that the tester simulates the Büchi automaton, and if a word is not accepted by the

```
1  void main() {
2    t_i v_i; // declare every input and state variable
3
4    Init_s; Init_t;
5
6    while(true) {
7      havoc v_i; // for every variable v_i ∈ InVar_s
8      Trans_s;
9      Trans_t;
10   }
11 }
```

Listing 6.6: Sketch of the monitored program.

automaton, the assertion in our monitored program is always adhered to. We make our considerations under the assumption that the arithmetic semantics of the operations are equal in the IEC 61131-3 program, the C-program, and the C-verifier.

Proof. The proof is done by induction over the number of rounds k. By construction, the state of the tester is coupled with the Büchi automaton, e. g., the every variable $\mathtt{s_i}$ of the tester corresponds to the configuration variable s_i (analogue for global variables g_i, history of variables v_i^j and $fail$). However, this coupling is established only at certain positions (lines) in the tester program.

For $k = 0$, we established the coupling between the initial states of the Büchi automaton, and the set of possible states of the tester program after both initialization procedures $Init_t$ and $Init_s$ were executed. The Büchi automaton has multiple initial states due to the degree of freedom in global variables and the history of program variables. Also, the tester program has multiple reachable states after $Init_t$ due to the nondeterminism caused by the havoc-statement. In detail, the coupling means that for each start state there exists a corresponding state in the tester program after $Init_t$, and vice versa. By the construction, the mapping should be obvious due to the same variable names. As the havoc-statements create exactly the combination of global variables (and history of program variables) as the Büchi automaton start states, the coupling is established.

The induction step $k = k' + 1$: We established the coupling behind the execution of $Trans_t$. As the computation of $\mathtt{s_i}$ ($1 \leq i \leq m+1$), $fail$, and history

of program variables are identical to the equations Equations (6.2) to (6.4) of the Büchi automaton these variable are equal. And the assertion of the tester is violated if and only if the acceptance condition A_{WC} is also violated. □

6.3 Implementation of the Verification Pipeline

The presented decision procedures are implemented into a verification tool for the IEC 61131-3 languages, called GETETA. GETETA provides a backend for the verification of GTTS, covering the transformation of the reactive program, the automaton or C-program construction, and the preparation of counter-examples. However, it does not provide a graphical interface. A user frontend for the verification is provided by STVS Figure 6.4 with support for the visualization of counter-examples and the generation of concrete test tables. Note that STVS only covers a subset of the language features.[1]. GETETA is part of VERIFAPS software project which provides a library and several tools for the verification of IEC 61131-3 software.

In this section, we want to dive into the details of the backend, and especially cover the remaining missing points: required transformations of the verification subject, and the input language of the GTTS.

6.3.1 Preparation of the Verification Subject

Our target, in this section, is to translate the given verification subject into a logical model: the SMV (Symbolic Model Verifier) format. SMV is the input format (textual representation) of the NUXMV model-checker. The software of the reactive system is given as a text file containing the Structured Text, Sequential Function Chart, or any other IEC 61131-3 language. Often an aPS project is exported as a PLCOpenXML file. The VERIFAPS-project provides a toolfor the extraction from XML into a suitable text file. In short, we translate everything into Structured Text, and then simplify the source code into a reduced form (called ST_0). The ST_0 version is translated into a single static assignment (SSA) form by a symbolic execution step, which can easily be encoded into SMV representation. In the SSA form, the PLC program is represented as a set of equations, which describes the new value of each state after a single cycle, depending on the input and old state values. The SSA form correspond to $\rho(P)$ given Definition 5.1.

The simplification into ST_0 has two limitations: both the execution and the memory have to be bounded. Bounded execution means, that there exists a

[1]Both tools are available online at https://verifaps.github.io/geteta and https://verifaps.github.io/stvs

maximal number of iterations for each loop and all possible execution cycles. Analog for the memory, there needs to exist an upper bound for its size. Both limitations are usually met by reactive software. In practice, bounded execution is already required to meet the real-time behavior of the systems. The memory limitation is met by the IEC 61131-3 standard, as it forbids the dynamical allocation of memory. Therefore, every memory is statically allocated at the start of a PLC program (before the first program invocation).

Symbolic Model Verifier Format. The software is represented by an SMV-module, where the behavior of the software is described by initial and next predicate over the defined variables. A variable definition consists of the name and the data-type.[2] The initial predicate describes the set of initial system states by determining the possible variable values. The next predicate defines a relation between the current and the successor states. Both predicates can model nondeterministic behavior either by allowing multiple initial or multiple successor states (for one concrete state). For example, the encoding of a GTT automaton (Figure 6.3) into SMV allows multiple initial states to match the possibilities of the values of global variable d, whereas the initial state of a IEC 61131-3 software is fully determined either by user-defined initialization or the default value of the standard.

Intermediate Representation ST_0. We build an intermediate representation (IR), called ST_0, which contains a rudimentary subset of Structured Text (assignment and if-statement) and the goto-statement. ST_0 originated from our previous work in [Wei15; Bec+15].

Definition 6.9 (ST_0)**.** *A ST_0 program is generated by the following grammar rules:*

$\langle ST_0 \rangle$ ::= $\langle type \rangle$ $\langle funcs \rangle^*$ $\langle prg \rangle$

$\langle type \rangle$ ::= TYPE ($\langle name \rangle$: '(' $\langle name \rangle$ (',' $\langle name \rangle$)* ')')* END_TYPE

$\langle funcs \rangle$ ::= FUNCTION $\langle name \rangle$: $\langle type \rangle$ $\langle scope \rangle$ $\langle stmt \rangle$ END_FUNCTION

$\langle prg \rangle$ ::= PROGRAM $\langle name \rangle$ $\langle scope \rangle$ $\langle stmt \rangle$ END_PROGRAM

$\langle scope \rangle$ ::= (VAR | VAR_INPUT | VAR_OUTPUT)
$\langle name \rangle$: $\langle type \rangle$ END_VAR [$\langle scope \rangle$]

[2] In more detail: Besides the state variables, NUXMV supports the definition of input and frozen variables. The first variables are nondeterministically re-assigned in each system transition. A frozen variable has a constant value after the initial state.

$$
\begin{aligned}
\langle stmt \rangle \quad ::= \; & \langle stmt \rangle \; \langle stmt \rangle \\
| \; & v := \langle expr \rangle \; \texttt{;} \\
| \; & \langle name \rangle \; \texttt{:} \; \langle stmt \rangle \\
| \; & \texttt{GOTO} \; \langle name \rangle \; \texttt{;} \\
| \; & \texttt{IF} \; \langle expr \rangle \; \texttt{THEN} \; \langle stmt \rangle \\
& \qquad (\texttt{ELSEIF} \; \langle expr \rangle \; \texttt{THEN} \; \langle stmt \rangle)^* \\
& \qquad [\texttt{ELSE} \; \langle stmt \rangle] \; \texttt{END\_IF} \\
| \; & \epsilon
\end{aligned}
$$

Variables in a ST_0 program can have any built-in type of IEC 61131-3 and user-defined enumeration types, defined in $\langle type \rangle$. But due to the restriction to the nuXmv model-checker, we only support bit-based data types when using this model-checker. In particular, these are boolean, bit words (BYTE, WORD, DWORD), the signed and unsigned integer types (SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT) and enumerations. Other user-defined types, like records or arrays, are handled by the simplification pipeline. Note that the Horn-based pipeline may handle floats directly depending on the chosen C-program verifier.

A ST_0 program consists of exactly one program definition, multiple function declarations, and enumeration type declarations. Thus, every call of a function block instance needs to be embedded into the (main) program body. The statements inside the program and functions are either an assignment, an if-statement, or a goto-statement. Statements can also be annotated with a label marking it as a target possible jump target of a goto-statement. Expressions are not restricted. The goto-statements are an extension in comparison to the IEC 61131-3 standard, but required to handle vendor-specific implementations of Structured Text, e. g., CodeSys allows goto-statement in Structured Text, and helps to efficiently translate Instruction List and Function Block Diagrams into Structured Text. (Both, Instruction List and Function Block Diagrams can contain goto-statements according to IEC 61131-3.)

Translation Steps. ST_0 is established by step-wise simplification steps on the given Structured Text program. The transformation pipeline is given in Figure 6.7. It starts with the translation of Sequential Function Charts, Function Block Diagrams, and Instruction List of the verification subject into a behavioral equivalent Structured Text source code. Additionally, a preamble is added to the source code containing *stubs* for the functions and data-types of the standard library. These stubs are either a complete implementation, e. g., `MIN(a,b)` or `SEL(cond, then, else)`, or an adaptation of the original implementation for the verification (cf. pulse timer in Listing 4.6). For example, the timer function blocks are accurate for static verification by counting cycles rather than on real-time clocks.

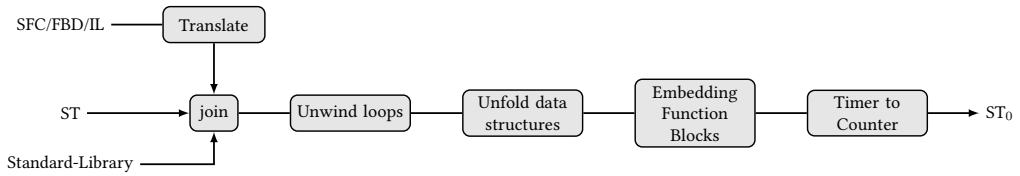

Figure 6.7: Preprocessing for ST_0

The translation for Instruction List and Function Block Diagram code into Structured Text is straight forwarded as the semantics of expression or function block calls are the same—only the representation differs. For example, the Instruction List commands operates on an implicit register (the accumulator) on which designated operations and operator are applied, like `LD 1; ADD 2` which loads 1 into the register and adds 2. For Sequential Function Chart code, the generation of equivalent Structured Text code is more complicated. A translation for a subset can be found in [Wei15]. Also, the following program transformations are defined by [Wei15], but we want to waste some paragraphs to revisit them briefly.

Unwinding of Loops. We distinguish between for-loops, which have an apriori known number $k \geq 0$ of loop iterations, and while-loops, where no such k exists. The unwinding of for-loops is trivial, the body of the for-loop is copied k-times, where the loop variable in the body is replaced by the current iteration. For while-loops, we have to guess the number of iteration k, but then we apply the same pattern with two extensions. Every copy of the loop body is wrapped in an if-statement asserting that the loop condition holds. Therefore, if the loop terminated after $k' < k$ iterations, then the remaining $k - k'$ bodies are not executed. Also, after the k-th if-statement, we add an additional if-statement with the loop-condition and body which set a (freshly introduced) error flag to true. Via this error flag, we can later assert that the chosen k was large enough. For these transformations, we silently assumed, that the loop condition is free of side effects.

Unfold Composite Data-Types. Structures and arrays are composite data types. In structures, the sub-elements are addressed by name and in arrays by one or multiple indices. In both cases, the number of sub-elements is known at compile-time and cannot be de- or increased at runtime.

The idea for removing the composite data type is the same as for structure and arrays: Unfolding the data type into multiple primitive variables. For this,

we need to consider how to rewrite the access and writing of sub-elements in expression, and the handling in assignments.

Let `v` be a variable of a structure type in a specific variable scope. First, we unfold the structure by introducing a new variable `v$e` for each sub-element `e` in the structure. Second, access to sub-elements `v.e` are rewritten as access to the introduced variable `v$e`. The same is valid for write access. Third, every assignment of `v:=w` needs to unfolded into multiple assignments for each sub-element `e`, i. e., `v$e := w$e`. We use the dollar sign "$" as the separator, because after IEC 61131-3 this sign is not allowed in identifiers and thus we automatically avoid clashes with other variables in the same scope.

For arrays, the index is computed at runtime. First, we unfold an array variable `a` into multiple variables `a$i`, where `i` is the index value. Note that we only consider the case where the array has only one dimension. Access and assignments of sub-elements need to be unfolded into a case distinction on the provided index. Optimizations may be possible if the array indices are computable at compile-time, e. g., if indices are literals populated by the loop-unwinding.

These transformations need to be applied recursively to cover the case of nested structure types or arrays. According to IEC 61131-3, data-type definitions need to be free of cycles.

Timer to Counter. IEC 61131-3 defines timer function blocks, e. g., `TOF`, `TON` and `TP`, that depend on the real-time clock of a PLC system. These timer function blocks help the developer to trigger actions depending on clock-time. These timer blocks are the back-bone of the time-dependent features of Sequential Function Chart. For example, function block `TON` provides a Boolean output `Q` which becomes true, after the input `IN` was true for a given duration `T`.

We simulate real-time by counting the execution cycles. Behind this transformation lies the assumption that every cycle consumes the same amount of time, and the execution (of a cycle) is done instantly (consuming no time, cf. [Hal98]). In practice, this assumption is not always valid due to the inaccuracy of the used clocks or the load of the system. On the other hand, the inaccuracy would affect the time constraint given in the specification (GTTs) equally. Thus, we prove that the software conforms to a GTT when both are evaluated under the same perfect clock.

Our approach requires a special implementation for each function block, depending directly on the real-time clock. Also, this transformation affects the variables with the built-in type `TIME`. These variables are converted into integer values representing the number of cycles under an overall considered cycle time.

For example, Fernández Adiego et al. [Fer+14] shows an abstraction of

timer function blocks with a completely nondeterministic behavior. This over-approximation is only valid if the required properties do not depend on the exact adherence to the timings, e. g., invariants. In [Fer+14], the authors require an additional monitor, which observes whether the timer triggers, to be able to use CTL. Also in [Fer+14], they give a fully correct implementation, which requires a global time variable. In contrast, our approach is a simplified version correct for our use cases. It is an open research question if there is a verification method for GTTs which allows such an over-approximation of the timers for the verification of GTTs.

Embedding Function Blocks. The peculiarity of function blocks is that they can be instantiated, and each instance maintains its own state. Under the embedding of function blocks, we understand that the state and the implementation body of each instance are embedded into the parent function block (or program). Therefore, there are two transformation steps for each instance: First, we introduce the variables of the instance in the caller scope, where the variable names are prefixed with the instance name. Second, each call site of the particular instance is replaced by the body of the function block, also, with prefixed variables. This process is applied recursively, and terminates finally due to the memory limit enforced by the ban of recursive instantiation. We use the dollar sign "$" (for the separation between the instance and the variable of the function blocks) again to avoid variable name collisions. This transformation is given in more detail in [Wei15].

Applying these transformations exhaustively, we finally reach the ST_0 form of a program.

6.3.2 A Textual Input Language for GTTs

Defining a table-shaped input mechanism is quite cumbersome and error-prone. We decided on a concise textual representation for GTTs, that can be understood and written by humans as well as easily generated by other tools. We only introduce the textual notions required for GTTs in this section. The notions for the relational test tables (Part II) are presented in the second part of this thesis.

Figure 6.8 shows the grammar for a single GTT. A file can contain multiple GTTs. A table declaration is initiated with the keyword `table`, followed by a table name (for identification) and the table body. The table body can contain different elements.

The ⟨*signature*⟩ defines the variables within a table. Program variables are declared with the `var` keyword (Rule ⟨*signature*⟩ in Figure 6.8) followed by the name and the data-type of the variable. This variable should also be defined in the verification subject. A program variable is an input, state, or output variable.

⟨*table*⟩ ::= `table` ⟨*name*⟩ '`{`' ⟨*body*⟩ '`}`'

⟨*body*⟩ ::= ⟨*signature*⟩* [⟨*option*⟩] ⟨*group*⟩ ⟨*functions*⟩*

⟨*signature*⟩ ::= `var` ⟨*modifier*⟩* ⟨*name*⟩ [`as` ⟨*name*⟩] '`:`' ⟨*datatype*⟩
| `gvar` ⟨*name*⟩ : ⟨*datatype*⟩ [`with` ⟨*clause*⟩]
| `column` ⟨*modifier*⟩* ⟨*name*⟩ `as` ⟨*expr*⟩
| `inherit_from` ⟨*name*⟩

⟨*modifier*⟩ ::= `next`|`assume`|`assert`|`output`|`input`|`state`

⟨*option*⟩ ::= `options` '`{`' (⟨*name*⟩ '`:`' ⟨*literal*⟩)* '`}`'

⟨*group*⟩ ::= `group` [⟨*name*⟩] [⟨*time*⟩] '`{`' (⟨*group*⟩ | ⟨*cell*⟩)* '`}`'

⟨*row*⟩ ::= `row` [⟨*name*⟩] [⟨*time*⟩] '`{`' (⟨*name*⟩ '`:`' ⟨*cell*⟩ ['`;`'])* '`}`'

⟨*function*⟩ ::= *a function definition after IEC 61131-3*

Figure 6.8: Grammar of the input language for GTTS. ⟨*cell*⟩ and ⟨*expr*⟩ are defined in Definition 5.4 and ⟨*time*⟩ in Definition 5.3

A state variable can also be evaluated in the current or *next* state (the state after computation). Every program variable can be categorized as an assumption (`assume`) or an assertion (`assert`). For example, the output modifier is a shortcut for the modifiers `state assert next`. The data-type can be any name of a built-in IEC 61131 or just "`ENUM`" (to refer to any enumeration). For some test table tools, e. g., `ttmonitor` (Chapter 8), the name of the enumeration type is required. We can provide it by appending it to the datatype, i. e., `ENUM_<name>`.

The variable names have to be valid in the simplified version of the verification subject, where all function blocks are embedded, and array and structures are unfolded. Therefore, to access a field `f` in a structure variable `s`, we need to write `s$f`. Note that we can assign a fresh name to a declared variable by using the `as` keyword. This is used to have shorter names for program variables, to get an abstraction from the concrete names in the body cells of the table, or to access the same state variable in the current and next state.

Additionally to the program variables, we can declare a column as a table expression. A column behaves like a program variable that is assigned to the defined expression. Global variables are introduced with `gvar` beside the name and data-type, their value range can be limited with a given constraint. Finally, for signatures, we can import the defined variables of another table by using the inheritance clause (`inherit_from`) along with the table name.

With ⟨*option*⟩, we can provide set the configuration parameters of the verifi-

```
1  table T {
2    var input IN : BOOL
3    var input PT : INT
4    var output IN : BOOL
5    var output ET : INT
6
7    group G1 omega {
8      row r1 [1,-] { IN: FALSE, PT: -; ET: 0; Q: FALSE}
9      row r2 [1,-] { IN: TRUE, PT: >=1; ET: >=ET[-1],<PT ; Q: TRUE}
10     row r3 [0,-] { PT: =PT[-1]; ET: >=ET[-1],<PT ; Q: TRUE}
11     row r4 [1,1] { PT: -; ET: PT; Q: -}
12     row r5 [1,-] { PT: -; ET: PT; Q: FALSE}
13   }
14 }
```

Listing 6.9: Textual representation of Figure 4.7

cation engine, e. g., with `mode` we can switch between bounded model-checking or IC3 for checking the weak conformance.

The main part of a test table is given in the root-`group`. There are two clauses: ⟨*row*⟩ represent a row and ⟨*group*⟩ a row group. Both have similar headers which allow specifying an identifier (⟨*name*⟩) for the entity and a time constraint (⟨*time*⟩). The row and group identifiers have to be unique. If the identifier is omitted, the row (group) receives a generated value. Omitted time constraints are treated as singleton ‘[1,1]‘. The body of a group clause consists of either groups or rows clauses, and the body of a row clause is a list of name and expression pairs, where the name refers to a defined column (a program variable, an alias, or defined column). If we omit an entry for a column in a row, we use the previously seen expression for this column from the rows above. Thus, we only specify the changes in the cell expressions between successive rows.

At the end of a table, user-defined functions used in the specification can be declared using the Structured Text according to the IEC 61131-3 grammar. Note that the function scope of GTT and the verification do not overlap. Thus, we are not able to use any function of the verification in GTT directly. Instead, every function needs to be defined explicitly in the table.

Listing 6.9 shows the GTT in Figure 4.7 in its textual representation.

Chapter 7

Evaluation

In this chapter, we evaluate GTTs. The evaluation includes two parts: On the one hand, we investigate and elaborate the specification on typical and practical examples. Our goal is to find their strengths and weaknesses regarding expressiveness and comprehensibility. The other hand is the feasibility of the verification. To show this, we use the implementation of our decision procedures.

Outline. The sections of this chapter are dedicated to three different evaluation categories: First, we explore function blocks, like counters and timers, which are defined by IEC 61131-3 (Section 7.1). Second, we dive into two similar reusable function blocks derived from industrial use cases (Section 7.2). Third, we look into function blocks for driving specific in a demonstrator plant (Section 7.3). The verification is discussed in a common section at the end of this chapter (Section 7.4).

Partially, the GTTs and the source code were already presented previously. Namely, Section 7.2 is derived from [Wei+17], and [Bec+17]. Moreover, Section 7.3 is a short version of [Cha+18b]. Section 7.1 is previously unpublished.

7.1 Built-Ins of IEC 61131-3

In this section, we present GTTs for common function blocks defined in the IEC 61131-3 standard. These are rather small function blocks but used throughout PLC programs, which rely on their correctness. Previously in Section 4.3.2, we show the pulse timer. Now, we present two new function blocks: `CTU` and `DEBOUNCE`.

7.1.1 Counting-Up

The first function block is CTU as shown in Listing 7.1. CTU provides a counter that is incremented by one if there is a rising edge on the input CU. A rising edge is a change from FALSE to TRUE between two invocations of a function block instance. The Function Block R_TRIG helps to recognize rising edges. Note that we diverge from the version presented in IEC 61131-3 by making the instance of R_TRIG and the call of the instance edge explicit. The standard allows the R_EDGE (and F_EDGE) modifier on Boolean input variables to add such a rising (or falling) edge detection. The input R allows to reset the internal counter to zero, and the input PV is the upper counting limit. If this limit is reached, the output Q becomes TRUE. The aggregated sum is provided in the output CV.

A valid GTT for the CTU is given in Figure 7.2. In addition to the variable signature of CTU, there are two new variables in the GTT. Firstly, the upper counting limit is specified by a global variable max. As a result, CTU must be valid for all possible limits. Secondly, we introduce a column $REdge$ which holds a projection function defined as `REdge := NOT CU[-1] AND CU`. $REdge$ is true if and only if a rising edge has occurred. Note that we could also use the state variable `edge$Q` (which is created from the embedding of the instance edge).

Moreover, in this case, it is possible to encode the entire behavior of CV into a single expression `=CV+SEL(NOT CU[-1] AND CU, 1,0)`, covering both the cases (no increment and increment). The function SEL provides a case distinction, where the function returns the second argument if the first argument is TRUE, or otherwise the third argument. Instead, we unfolded the case distinction by using multiple rows.

The first row expresses the initial state of the counter, the aggregated values are zero, and the overflow signal already depends on the provided upper limit. The second row models the increment on a rising edge, where CU is either PV (when the upper bound is hit) or incremented (previous value CU[-1] added by one). The third row is the case of a non-rising edge resulting in the same aggregated value. The fourth row allows resetting the counter. Note that the constraints on Q are identical on all rows, as we avoid the repetition and omit the duplicates. Moreover, rows 2 to 4 can be repeated arbitrarily often (or skipped). This table describes a family of scenarios which all have in common, that PV is changed during execution.

For ω-regular languages the infinite repetition ϵ^ω of the empty word ϵ is undefined. By our automaton construction, this case is well-defined for GTTs. In this particular case, we obtain an automaton with four states, where the states s_2, s_3, and s_4 (for the corresponding table rows) are fully connected. Thus, leaving s_2 (w.l.g.) requires that we enter one of the three states. We summarize: A strong repetition on a row group over skippable table rows results in the

```
1  FUNCTION_BLOCK R_TRIG
2    VAR_INPUT CLK: BOOL; END_VAR
3    VAR_OUTPUT Q: BOOL; END_VAR
4    VAR M: BOOL; END_VAR
5    Q:= CLK AND NOT M;
6    M:= CLK;
7  END_FUNCTION_BLOCK
8
9  FUNCTION_BLOCK CTU
10   VAR_INPUT
11     CU, R : BOOL; PV : INT;
12   END_VAR
13
14   VAR_OUTPUT
15     Q : BOOL; CV : INT;
16   END_VAR
17
18   VAR edge : R_TRIG; END_VAR
19
20   edge(IN := CU, Q => CU);
21
22   IF R THEN
23     CV:= 0;
24   ELSIF CU AND (CV < PVmax) THEN
25     CV:= CV+1;
26   END_IF;
27   Q:= (CV >= PV);
28 END_FUNCTION_BLOCK
```

Listing 7.1: Function Block counting-up on rising edges.

#	ASSUME				ASSERT		⏲	
	CU	*REdge*	**R**	**PV**	**CV**	**Q**		
1	FALSE	–	FALSE	*max*	0	=CV>=max	1	
2	–	TRUE	FALSE	⏐	=SEL(Q, PV, CV[-1]+1)	⏐	$[0,1]$	⊤
3	⏐	FALSE	FALSE	⏐	=CV[-1]	⏐	$[0,1]$	$-\infty$
4	⊥	–	TRUE	⊥	0	⊥	$[0,1]$	⊥

Figure 7.2: GTTs for CTU

```
1  FUNCTION_BLOCK DEBOUNCE
2          VAR_INPUT
3                  IN: BOOL;
4                  DB_TIME: TIME:= t#10ms;
5          END_VAR
6
7          VAR_OUTPUT
8                  OUT: BOOL;
9                  ET_OFF: TIME;
10         END_VAR
11
12         VAR
13                 DB_ON: TON;
14                 DB_OFF: TON;
15                 DB_FF: RS;
16         END_VAR
17
18         DB_ON(IN:= IN, PT:= DB_TIME);
19         DB_OFF(IN:= NOT IN, PT:= DB_TIME);
20         DB_FF(S:= DB_ON.Q, R:= DB_OFF.Q);
21
22         OUT := DB_FF.Q1;
23         ET_OFF := DB_OFF.ET;
24 END_FUNCTION_BLOCK
```

Listing 7.3: Function Block DEBOUNCE

situation where at least one of the table rows needs to be selected—written as a formal language, we would write $r_1 \cdot (r_2 \cup r_3 \cup r_4)^\omega$ for the table rows second, third and fourth table row.

7.1.2 Debouncing of Signals

Listing 7.3 shows the Function Block DEBOUNCE for debouncing a signal. Often a mechanical component does not provide a clean signal. For example, in a perfect world pressing and holding a button results in one rising edge of the input signal. In practice, multiple (falling and rising) edges can occur for a short time period after the button is pressed due to physical limitations. A solution to

#	ASSUME IN	ASSUME DB_TIME	ASSERT OUT	ASSERT ET_OFF	⏲
1	–	T	FALSE	–	$T-1$
2	FALSE	–	⏐	⏐	1
3	TRUE	T	⊥	⏐	$T-1$
4	TRUE	⏐	TRUE	⏐	≥ 1
5	FALSE	⏐	TRUE	⏐	$T-1$
6	FALSE	⊥	FALSE	⊥	1

Figure 7.4: GTT for Function Block DEBOUNCE, where meta-variable T represents an arbitrary waiting time, and is instantiated to retrieve a valid GTT.

this bouncing is to ignore the signal changes that only persist for a short time span. In the implementation of DEBOUNCE this time is given in the input variable DB_TIME. The input variable IN takes the input signal which should be *debounced*. And the debounced signal is written to OUT. To manage the timings, we use two TON timers. The output Q of these timers becomes true, after the input variable IN was true for at least PT milliseconds. Additionally, we use an instance of RS to hold the current state of the signal. The Function Block RS implements a simple reset-dominant flip-flop (4.3.1). If R is true, the state of the RS flip-flop becomes false. It becomes true if S is true and otherwise the state is unchanged. Note for the verification, the variables of datatype TIME are degraded to integers denoting the number of cycles.

The GTT is given in Figure 7.4 and describes a non-repeating test sequence. We notice, the variable T as an input variable and as a time constraint. Note that T is neither a global variable nor a program variable. An issue of GTTs is their missing support of non-rigid time constraints. Every time constraint must specify a range with concrete numbers. Specifying a function block whose timing behavior is parametric is only possible using a variable on the meta-level, i. e., T. But the verification is only possible for an instantiation of T.

In the first row, we flood arbitrary input signals for up to $T-1$ cycles, and require a negative output. The time constraint $\leq T-1$ is required, otherwise the first row would include the input IN=true for T cycles and output variable OUT needs to be true. Row 2 is a reset of the function block because we need to ensure a certain state (IN was not true in the last cycle) for the Row 3. In Row 3 and 4 we test for a correctly timed switch of the output from false to true after supplying true for IN for T cycles. Analogously, Row 5 and 6, and the switch of

the output from true to false.

7.2 Industrial Examples

We increase the code complexity by diving into two pieces of industrial-inspired code. The first function block `LinRe` is for linear re-scaling of incoming sensor values. The second function block `MinMax` provides a calibratable *clamp* function, which enforces the incoming value to be between the learned minimum and maximum value. If this range is violated for too long, a warning rises.

7.2.1 Linear Re-Scaling

Function of the Software. The core of this function block is a linear interpolation that maps actual sensor values to a defined range of physical values in software. This is commonly needed functionality in software for automated systems. A typical scenario for this function block is the translation of the measured discretized voltage, e. g., in a range of $[0, 4095]$, into a normalized scale, e. g., the brightness of a workpiece. The normalization is established by measuring defined references, in our example a light and dark workpiece, during the calibration phase. The Function Block `LinRe` (given in Appendix B.2) can operate in two different modes: Before the actual operation, the calibration function (mode "Teach") expects two independent reference points to learn the linear relationship between sensor (input value) and physical values (output value). After calibration, the mapping function (mode "Op") performs linear interpolation where it translates a sensor reading into the physical value according to the learned data.

A schematic view of the function block is shown in Figure 7.5. In addition to the mode selector (Mode) and the sensor value input (`X`), the block has two additional inputs needed during calibration: (`TPy`) is used as the physical reference value during teaching and (`TPSet`) indicates that teaching is in progress if set to true. The block's single output is the physical value (`Y`). In normal operation, after two reference points (x_1, y_1) and (x_2, y_2) have been learned, the input value `X` results in an output value $Y = L(x_1, y_1, x_2, y_2, X)$ which the linear regression between the two points defined as

$$L(x_1, y_1, x_2, y_2, X) := \; y_1 + \frac{y_2 - y_1}{x_2 - x_1}(X - x_1) \;\; . \tag{7.1}$$

If the two reference points make interpolation impossible (e.g., if $x_1 = x_2$), the function block enters an error state. If no reference point is presented for more than a certain amount of cycles (while in teaching mode), the block also switches into the error state. Before finishing calibration or if the function block

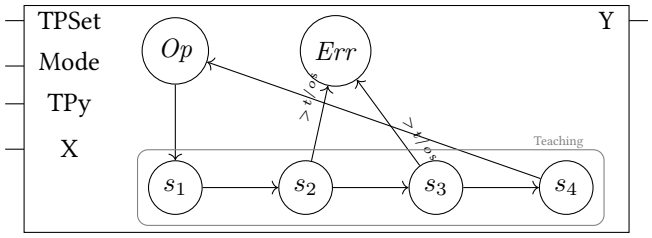

Figure 7.5: Schematic view of the investigated function block with a state machine describing its operation.

	ASSUME				ASSERT	
#	TPy	TPSet	Mode	X	Y	
1	60	0	Op	0	0	10
2	60	0	Teach	1.2	0	10
3	60	1	Teach	1.2	0	1
4	2108	0	Teach	4.0	0	10
5	2108	1	Teach	4.0	0	1
6	0	0	Teach	0	0	1
7	0	0	Op	28	.438	1
8	0	0	Op	316	3.65	1
9	0	0	Op	3132	4.20	1

Figure 7.6: Concrete test table of analog sensor function block

is in the error state, the output Y is always 0. Figure 7.5 includes a state chart for the block which contains the normal operation state, the error state, and the teaching state which is subdivided into four substates s_1 to s_4.

A Concrete Test Table. One concrete test case for this block is shown in Figure 7.6. It covers the calibration and normal operation. The block is set into teaching mode and two reference points $(60, 1.2)$ and $(2108, 4.0)$ are used for block calibration from Row 2 to 5. Afterward, the normal operation of the block is tested in steps 7–9 which send inputs (X) of 28, 316, and 3132. The expected physical values (Y) are 0.438, 3.650, 4.200 according to linear interpolation. The software fails this test case if it does not produce the expected output in one or more steps.

Test table. This test case can be generalized to a GTT with increased test coverage. The resulting generalized table does not only cover the few concrete test values mentioned in the concrete table but includes a wide range of possible input sequences. The generalization (Figure 7.7) of the concrete test begins

#	ASSUME TPy	TPSet	Mode	X	ASSERT Y	⏲
1	–	–	Op	–	0	*
2	–	0	Teach	–		$[1, t/o]$
3	y_1	1		x_1		1
4	–	0		–		$[1, t/o]$
5	y_2	1		$x_2, \neq x_1$		1
6	–	–	⌴	–	⌴	1
7	–	–	Op	–	L	$-_\infty$

Figure 7.7: Generalized test table of function block for linear re-scaling, where L is the linear regression, see (7.1).

in Row 1 where before calibration, `Y` is always 0. In Row 2, the mode is set to "Teach" (entering s_1 in Figure 7.5), and in Row 3, the first teaching point (x_1, y_1) is provided to the block (s_2). In Row 5, the second point (x_2, y_2) is sent to the system (s_3). Row 2, 4 and 6 are waiting for phases between the teaching points. The waiting time is not fixed but limited by the maximum waiting time given as timeout (t/o). After calibration, the block is set back to normal mode in Row 7 where an arbitrary sensor value X is sent to the function block. The expected output is the linear interpolation value according to (7.1). This last step is repeated indefinitely often.

This generalized test table represents infinitely many individual finite test cases for all possible reference points and queries. However, it is still not a complete behavioral specification for the block. The sequence of steps is fixed and does not cover all cases. Figure 7.7 only represents the normal operation. All situations where the block switches to an error state are not covered (e.g., if three reference points are given or if more than t/o waiting cycles have occurred).

In contrast to the concrete test table, the GTT allows us to directly capture the dependencies between the variables, e. g., we see that `Y` is defined by a formula. Given only the concrete test table, it is hard to produce new valid test tables.

7.2.2 Clamping Function Block

Function of the Software. We consider a system whose purpose is to watch over the input values and to raise a warning if they repeatedly exceed the previously learned range of allowed values. Such diagnosis functionality is common in safety-critical applications. More precisely, the system under test is the Function Block `MinMaxWarning` (Appendix B.1) with input variables `mode`, `learn`, `I` and the output variables `Q`, and `W`. `MinMaxWarning` learns the typical range of input values and warns the caller when encountering subsequent

#	ASSUME mode	learn	I	ASSERT Q	W	⏲
1	Active	–	–	0	TRUE	–
2	Learn	TRUE	q	0	FALSE	1
3	Learn	TRUE	p	0	FALSE	1
4	Active	–	$[p, q]$	$[p, q]$	FALSE	*
5	Active	–	$> q$	q	FALSE	5
6	Active	–	$< p$	p	FALSE	5

(a)

#	ASSUME mode	learn	I	ASSERT Q	W	⏲
1	Learn	TRUE	q	0	TRUE	1
2	Learn	TRUE	p	0	TRUE	1
3	Active	–	$> q$	q	FALSE	10
4	Active	–	$> q$	q	TRUE	≥ 1
5	Active	–	$[p, q]$	$[p, q]$	TRUE	5
6	Active	–	$[p, q]$	$[p, q]$	FALSE	≥ 1

(b)

#	ASSUME mode	learn	I	ASSERT Q	W	⏲
1	Learn	TRUE	q	0	FALSE	1
2	Learn	TRUE	p	0	FALSE	1
3	Active	–	$= p - 1$	p	FALSE	10
4	Learn	TRUE	$= p - 1$	0	FALSE	1
5	Active	–	$= p - 1$	$= p - 1$	FALSE	≥ 1

(c)

Figure 7.8: Three GTTs for the specification of the `MinMaxWarning`'s behavior

outliers.

`MinMaxWarning` operates in two modes, `Active` and `Learn`, as selected by the caller via `mode`. During the learning phase, the function block learns the minimum and maximum values of the input values (`I`). When switched into the active phase, the function block checks that the input value (`I`) stays within the previously learned interval. The output value `Q` is equal to `I` if `I` is within the learned interval; otherwise, the nearest value from the interval is returned. If the input value keeps being out of range for a specified number of cycles, then the function block raises an alarm via the variable `W`. The alarm is reset after a certain cooldown time if the input value falls back into the learned interval. An unlearned function block always signals a warning.

Test Tables. The required functionality is partially described by the three GTTs shown in Figure 7.8. These tables have two global integer variables p, q. The global variable p represents the minimum input value, and q the maximum supplied value respectively, we specify the constraint $p \leq q$ in the model checker. The waiting time before an alarm is raised is fixed to ten cycles, and the cool-down time to five cycles.

The first GTT (Figure 7.8(a)) specifies a behavior without warning. In the beginning, it is checked that the unlearned system returns the default constants (`Q=0` and `W=TRUE`; Row 1). This phase can be interrupted by switching into the learning mode (Rows 2 and 3). During learning, the system learns the minimum p and the maximum q input values. Subsequently, the system response is only allowed to be within this range. In Row 4, we test the non-warning case, in which only inputs between p and q are supplied. Rows 5 and 6 test for input values outside the range, and ensure that no warning rises too early.

The second GTT (Figure 7.8(b)) targets the case where warnings need to be risen. We use the same initialization, but require a warning due to a too high input (Rows 3 and 4). Rows 5 and 6 specify a cool-down phase with a duration of five cycles.

The third GTT (Figure 7.8(c)) specifies the behavior in the corner case that the system is switched into learning mode just one cycle before a warning would occur. After re-teaching, no alarm should be triggered, as the value should now be included in the learned interval.

7.3 Plant-Specific Function Blocks

We evaluate our GTTs on the specification and verification of plant-specific function blocks. The verification subject is the Pick-and-Place Unit (PPU) [Vog+14]. In contrast to the previous examples which are universally usable in different contexts, the function blocks of the PPU control a specific hardware module of a concrete factory.

These function blocks are programmed specifically for an implicit environment model tailored to the PPU and a specific set of situations. The environment model is implicit, as in our case, the software reads sensors and maintains its state locally. This program state estimates the state of the environment partially without making the expected and assumed environmental model explicitly. Therefore, the program state is the minimally needed abstraction of the environment required for the verification and created by the software engineer.

For printing, we omit and rename variables in the presented GTTs. A detailed version of each GTT can be found on the companion material [Wei21] or the

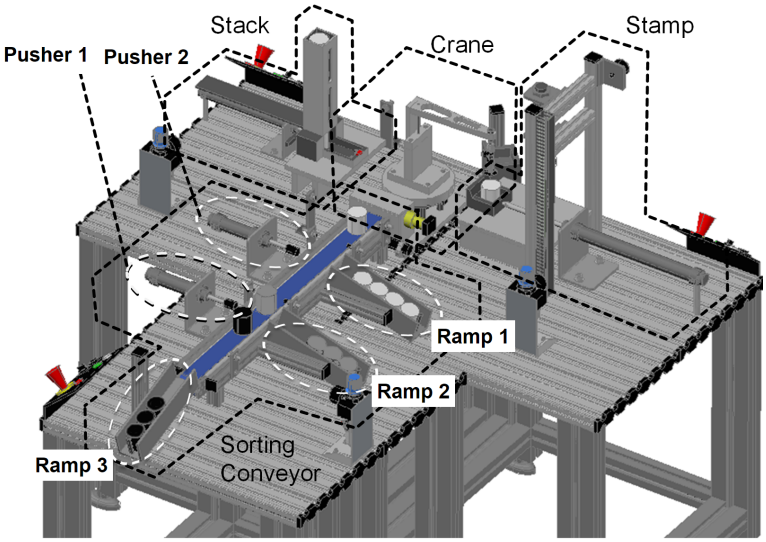

Figure 7.9: The Pick-and-Place Unit in a medium-sized configuration consisting a stack, crane, stamp, and the conveyor belt with different ramps and pushers. Figure provided by Institute of Automation and Information Systems from Technical University of Munich (TUM)

companion web page[1] of the original publication [Cha+18b].

Verification Subject: (x)PPU. The Pick-and-Place Unit (PPU) is a lab-size manufacturing plant demonstrator [Vog+14] to benchmark evolution scenarios for aPS. The extended PPU (xPPU) [VBS18] brings new extensions of the hardware, software, and evolution scenarios. In the following, we do not distinguish between xPPU and PPU. The PPU provides various configuration levels with different hardware and software versions. Figure 7.9 shows the plant in a medium-sized configuration level.

The PPU realizes the basic functionalities representative in intralogistics systems as identified by [Spi+17]. The hardware of the original PPU consists of four hardware modules: a stack (providing workpieces), a stamp (manipulation of workpieces), a conveyor belt (sorting of workpieces), and a crane (transportation of workpieces between the hardware modules).

In the PPU, workpieces are processed differently depending on their material (black plastic, white plastic, and metal). The operator places the workpieces into the Stack, which provides single workpieces for the processing to the Crane. The Crane delivers the workpieces either (a) directly to the Conveyor belt for sorting or (b) first to the Stamp and then, after processing, to the Conveyor Belt.

[1]Companion Website: https://formal.iti.kit.edu/at2018

#	INPUT		OUTPUT						⏲
	EmergencyStop	*Sucked*	*StackSlider*	*CraneLower*	*VacuumOn*	*VacuumOff*	...	*StampPusher*	
1	TRUE	–	–	–	–	–	...	–	≥ 0
2	FALSE	⊥	–	–	Sucked	=¬Sucked	...	–	$\leq 250\,\text{ms}$
3	FALSE	⊥	FALSE	FALSE	Sucked	=¬Sucked	...	FALSE	$-_\infty$

Figure 7.10: A GTT for specifying the emergency-stop behavior (Case 1). For readability, we use a single input variable *EmergencyStop* combining the three active-low signals for the emergency buttons (in the original version, there are three input variables, one for each button). Also, 11 output variables are not shown that control further emergency actions.

Transportation by the Crane requires a sequence of actions: turning to the target, stopping, lowering the arm, gripping the workpiece with pneumatic pressure, rising the arm, turning, stopping, lowering the arm, releasing the pneumatic pressure, and rising the arm again. Turning and stopping, and lowering and rising are implemented by two actuators: a motor and a cylinder.

The Stamp starts processing once a workpiece has been placed into the tray at the end of a slider. The slider cylinder is retracted to move a workpiece to the stamp, and after the workpiece has been stamped, the slider is extended to move the workpiece back to its arrival position. The stamping process is just the movement of a slider cylinder that is pressed the stamp against the workpiece.

The Conveyor Belt provides sorting capabilities. It starts when a workpiece is placed by the Crane at the start of the Conveyor Belt. The Conveyor Belt moves the workpieces towards its end, and during the workpiece transportation it is either pushed off on a side ramp by a pusher, or delivered to ramp at the end of the belt.

The three PPU scenarios presented in the following sections are variations of this basic configuration. They differ in their mechanic, electric, electronic, and/or software configuration.

7.3.1 Case 1: Emergency Stop

In this case, we dive into the handling of emergency stops. Emergency stops are either triggered by the operator or detected by safety measurements, e. g., light barriers or temperature sensors, or software diagnosis modules.

In contrast to the expected (normal) behavior of the software, which follows a well-known and well-defined processing sequence, these unexpected or abnormal situations can emerge at any point in time. For this reason, a complete validation or tests of these emergency situations are hard to establish.

In our case, the emergency stop of the PPU brings the system to a safe halt to avoid damage to the plant and the workpieces. Figure 7.10 specifies the emergency stop behavior. Note that this particular GTT only describes the emergency routine; to fully specify the PLC behavior, other GTTs for non-emergency situations would be needed.

The PPU has three buttons to trigger the emergency stop, which we summarize under a projection function. Note that the emergency stop is expressed negatively: `TRUE` signals that no emergency stop is triggered. The column's projection function is defined as:

$$\begin{aligned}\texttt{EmergencyStop} := \texttt{SorterEmergencyStop} \wedge \\ \texttt{StampEmergencyStop} \wedge \texttt{MagazinEmergencyStop}.\end{aligned}$$

As soon as one of the emergency stop (Row 1) is triggered, the emergency-stop process is initiated and the following end position should be reached (Row 3): The pneumatic cylinders in the stack are retracted, the rotation of the Crane is stopped, the Conveyor Belt of the sorting module is stopped, and the pushers on the Conveyor are all retracted. The end position of the pneumatic pressure on the Crane's gripper is more delicate, as turning off the vacuum without considering the current state of the gripper may allow a workpiece to fall to the ground. Therefore, if a workpiece is currently being gripped, then the gripper must continue to hold onto that piece until the emergency procedure has terminated (or the workpiece is removed manually). To reach the end position, the system is allowed to consume (up to) 250 ms. For the verification, we translate the time value into the number of cycles by dividing it by the cycle time.

The end position is maintained infinitely long. The release and restart of the system are not covered by this GTT.

7.3.2 Case 2: Partial Functional Behavior of the Crane

This application case is based on the fifth scenario in the PPU ([Vog+15]), where the behavior of the crane is optimized to achieve a higher overall throughput of workpieces. In the previous version (Scenario 3), the plant processes only a single workpiece at a time, i. e., a new piece is taken from the stack only after completely processing the previous piece and storing it in the ramp. In the new version (Scenario 5), the processing at the Stamp is used for the transportation of workpieces from the Stack to the Conveyor Belt. This is only possible if the

#	ASSUME							ASSERT			⏲
	CraneUp	*Metallic*	*WPReady*	*Go_Up.X*	*OnStack*	*OnConveyor*	*OnStamp*	*TurnCCW*	*Lower*	*VacuumOn*	
1	–	–	–	–	–	–	–	–	–	–	$-_p$
2	TRUE	FALSE	TRUE	TRUE	–			–			1
3	–	–	–	–	FALSE			FALSE			≥ 0
4					TRUE						1
5					–				TRUE	TRUE	≥ 0
6									TRUE		1
7	FALSE								FALSE		≥ 0
8	TRUE										1
9	–					FALSE		TRUE			≥ 0
10						–		TRUE			≥ 0
11						TRUE		FALSE	TRUE		1
12											1
13										FALSE	1
14	FALSE								FALSE		≥ 0
15	TRUE										1
16	–						FALSE	TRUE			≥ 0
17							TRUE	TRUE			1
18							TRUE	FALSE			1
19							–	–	–	–	$-_p$

$-_\infty$

Figure 7.11: A GTT for the crane-as-buffer maneuver to bypass the Stamp and improve overall workpiece throughput.

next workpiece offered by the Stack does not require an imprint. After this extra transportation has been completed (from the Stack to the Conveyor Belt), the Crane returns to the Stamp and the plant in Scenario 5 behaves exactly as in Scenario 3.

Figure 7.11 shows the GTT specifying the functional behavior during the optimization phase. Row 1 is a *waiting* row, which waits for the constellation of input values defined in Row 2. Note that the progress flag on Row 1 disallows waiting when Row 2 becomes possible. This is a typical pattern for partially modeling behavior in time. The variable `Go_Up.X` is a state variable indicating whether we are in a certain step in the SFC of the Crane. Variables `S.X` are automatically generated for each step `S` in an SFC which are true when step `S` is currently active. In the GTT, we are using this state variable to spare modeling the environment. Otherwise, we need to model the state of the Crane, the Stamp, and the Stack. In particular, we have to encode the following situation in the GTTs: "The Crane has delivered a workpiece to the Stamp. The workpiece is dropped. Also, a new workpiece, which does require a stamping, is available at the Stack." This situation requires the history to be modeled, we have to remember that a workpiece has been delivered to the Stamp. Moreover, our estimation has to be in sync with the decision of the software. By using the state variable `Go_Up.X` we avoid all of this by using the abstract environment model built into the Crane function block.

#	ASSUME SFCReset	Lightness	StartVar	ASSERT Push1	Push2	⏲
1	FALSE	–	FALSE	–	–	≥ 1
2	↓	↓	TRUE	–	–	1
3		⊥	–	FALSE	FALSE	1
4	⊥	FALSE	–	FALSE	FALSE	$-_\infty$

Figure 7.12: A GTT specifying the PLC software's behavior for sorting black work pieces.

#	ASSUME Metallic	Lightness	PusherIn1	PusherOut1	StartVar	ASSERT Push1	Push2	⏲
1	–	–	–	–	FALSE	–	–	≥ 1
2	↓	↓	↓	↓	TRUE	–	–	1
3	⊥	⊥			–	FALSE	FALSE	1
4	FALSE	TRUE	↓	↓	↓	↓	↓	≥ 1
5	–	–	⊥	⊥		⊥		340 ms
6	↓	↓	TRUE	FALSE	↓	TRUE	↓	1
7	↓	↓	–	–	↓	TRUE	↓	200 ms
8	⊥	⊥	TRUE	FALSE	⊥	FALSE	⊥	≥ 1

Figure 7.13: A GTT specifying the PLC software's behavior for sorting white non-metal work pieces.

The Rows 2-18 specify the optimization maneuver which consists of the following steps: (1) the crane turns to the Stack (from the Stamp), (2) picks up the new workpiece (lower, vacuum-on, raise), (3) turns to the Conveyor Belt, (4) release the piece (lower, vacuum-off, raise), and (5) turns back to the Stamp. After finishing the maneuver, the GTT goes back to waiting for the conditions of Row 1 to become true (Row 19). Note that Rows 1–19 are grouped together and can be repeated so that the behavior specified by the GTT can contain the bypass-maneuver infinitely often. Alternatively, the GTT can remain in Row 19 (forever) waiting for a repetition of the maneuver.

7.3.3 Case 3: Sorting the Workpieces

This application case is based on Scenario 11 of the PPU [Vog+15]. The scenario is concerned with sorting workpieces on three different ramps according to their type: white non-metal (plastic) pieces are put into Ramp 1, metal pieces into Ramp 2, and black pieces into Ramp 3. There are two sensors at the beginning of the Conveyor Belt: a light sensor for distinguishing black and non-black workpieces, and an inductive sensor that detects metallic workpieces. The sensors are placed at the start of the Conveyor Belt. If a workpiece is detected,

it is moved to the end of the belt (Ramp 3) past two pushers. Sorting is done at the Conveyor Belt by pushing off the moving workpiece of the belt at the right time.

We present two GTTs for the sorting on the Conveyor Belt. Figures 7.12 and 7.13 specifies the correct behavior for handling black and white non-metal pieces, respectively.

Black workpieces are sorted into Ramp 3 making the specification easy. If a black piece is recognized (`Lightness`), the pushers should not be extended. In Figure 7.12, the GTT waits in Row 1 that the Conveyor Belt starts. `StartVar` is an input variable indicating when a workpiece arrived, and `SFCReset` is an (implicit) input variable for SFCs which triggers a reset when set to `TRUE`. With Row 2 the handling of workpieces starts. After that, the two pushers must be retracted (Row 3). If the input variable `Lightness` is `FALSE`, the pushers stay retracted. Thus, the black workpiece moves into Ramp 3.

The GTT in Figure 7.13, specifying the case of sorting a white non-metal piece is more complex as it must consider the sensor input variable `Metallic` (in addition to `Lightness`). Moreover, it must specify the activation of the first pusher, which needs to be extended at the right moment. Rows 1-3 are similar to the GTT for sorting black pieces: After starting the process (Rows 1 and 2), the pushers are retracted (Row 2). When a white non-metal piece is detected (`Metallic=FALSE`) and `Lightness=TRUE` in Row 4), the behavior starts to diverge from the one for black pieces. After waiting for 340ms (Row 5) and checking that the first pusher is indeed retracted (`PusherIn1=TRUE` and `PusherOut1=FALSE`, Row 6), the PLC software must set the output variable `Push1` to `TRUE` for 200ms to activate the pusher and move the workpiece into Ramp 1. Finally, the software must retract the pusher again (Row 8).

7.4 Verification

In Table 7.14 we present the results including the runtime statistics for the verification of the weak conformance. Our verification GETETA tool handles the construction of the input files for model-checking (SMV) or C-program verifier (C source code) (Chapter 6). The given run-times are the used time on the CPU (CPU time) of the verification backend of both pipelines. Successful verifications were repeated five times. We show the median of the CPU times. In some cases the verification was unsuccessful. We distinguish the following reasons for failure:

- The verifier detected an error (“unsafe”).
- The time-out of 10 minutes was reached (“t/o”).

- The memory was exhausted ("oom").
- The verifier complained about the given input file ("err").

We used the Version 2.0.0 of the model-checker nuXmv [Cav+14], eldarica in Version 2.0.5 and SeaHorn in Version 10.0.0-rc0-d9ef838f. The experiments were run on a (max.) 4.6 GHz system with Intel Core i7-8565U and 16 GB RAM. We used the IC3 verification technique in nuXmv. No special options were given to eldarica, especially, no extra memory limitations were set. But eldarica requires an additional pre-compiler to run on the input file, as eldarica is not able to handle pre-compiler directives. SeaHorn was called with enabled analysis of crab ("`sea pf -crab`"). crab is an abstraction interpretation machine for finding helpful auxiliary specifications. In Table 7.15 we also present results for the verification of strict conformance. For strict conformance, we use nuXmv with its IC3 for LTL verification ("`check_ltlspec_ic3`").[2]

We were forced to make limitations on the global variables to make verification feasible in certain cases: In particular, for the verification of `CTU` (Figure 7.2) we restricted max to the range $[0, 100]$ for the verification with nuXmv (eldarica and SeaHorn were not restricted). The global variables p, q in the gtts for Function Block `MinMaxWarning` (Figures 7.8(a) to 7.8(c)) were restricted to the same range for all three verifiers. `LinRe` (Figure 7.7) is only practical verifiable with small ranges of the given reference points (x_1, y_2) and (x_2, y_2). We restricted each of the four variables to $[0, 3]$. No restriction on global variables for the PPU experiments and `TP` function block.

We instantiate the meta-variable T in the gtt in Figure 4.7 with 10.

Discussion. As can be seen in the table, all required proofs were successful in reasonable time with the model-checking pipeline and nuXmv. This is the oldest verification pipeline and therefore most stable one. SeaHorn has promising run-times in some cases, especially for unbounded global variables in `CTU`. In contrast to SeaHorn, which is mainly a verifier for C-programs, eldarica focuses on solving Horn clauses. We recognize this by a slower performance, that may arise from the missing pre-analyses of the given program.

The proof time and the proof complexity is mainly influenced by three factors: First, the number of required row iterations. Second, the value range of the global variables. Third, the arithmetic complexity inside the program and the table. These factors can be ascribed to a costly forward search for a counter-example in the IC3 algorithm (cf. IC3 in Section 2.3). The number of row iteration increases the required search depth as IC3. Often a row assertion can only be spuriously

[2]Side note: The default LTL backend in nuXmv abnormally terminates (segfault); it is reported and will be hopefully fixed soon.

Table 7.14: Statistics for the verification of weak conformance. CPU times are the median of five samples if the verification was successful. Otherwise, "unsafe" denotes that the verifier detected an error, "t/o" the time-out of 10 minutes was reached, "oom" out-of-memory or "err" the verifier was not able to parse the given input file.

Specification	Implementation	Model-checking nuXmv		C-program verifier eldarica	C-program verifier SeaHorn
		cpu time [secs]	model size [bit]	cpu time [secs]	cpu time [secs]
Figure 7.2	Listing 7.1	34.29	77	80.25	0.33
Figure 4.7	Listing 4.6	0.08	74	11.78	0.90
Figure 7.4	Listing 7.3	1.39	137	t/o	75.92
Figure 7.7	Appendix B.2	171.28	554	t/o	unsafe
Figure 7.8(a)	Appendix B.1	3.96	178	t/o	unsafe
Figure 7.8(b)	Appendix B.1	32.10	182	t/o	unsafe
Figure 7.8(c)	Appendix B.1	1.39	178	297.40	unsafe
Figure 7.10	PPU Sc 13	1.05	784	err	16.37
Figure 7.11	PPU Sc. 5, Crane	0.50	114	t/o	53.92
Figure 7.12	PPU Sc. 11, Pusher	0.64	127	t/o	7.81
Figure 7.13	PPU Sc. 11, Pusher	30.74	186	oom	unsafe

violated, then IC3 needs to search deeper in the state space (exploring further row iterations) until the next table row is reached. Only the violation of the invariant helps to narrow the approximated set of reachable states and strengthened the required inductive invariant. Similar things, happen if the ranges of the global variables are wide. Consider the example with Function Block `CTU`: To find a violation we need to count until the maximal value could be reached and every increment requires a new application of the state transition. The third factor makes these applications costly as arithmetical operation (especially, multiplication and division) explodes in their bit-wise symbolical representation. Hence the actual semantics of the arithmetics is not present anymore. Therefore, IC3 has to guess and build clauses which captures the origin arithmetics.

Unsatisfied with the results of the GTTs for the Function Block `MinMax-Warning`, we tested these cases with IC3IA [Dan+16], an IC3 implementation with support for the theory of linear integer arithmetic. We did not use this theory and stick with bit-operations. IC3IA performs better without any bounds for the global variables p, q. In detail, we achieved the runtimes in Table 7.15.

Although, we could only prove the conformance with a strong restriction on the global variables, the proofs still cover a wide variety of possible input sequences.

Table 7.15: Alternative verifications for Function Block `MinMaxWarning`. Table gives runtimes for the verification of strict conformance with nuXmv and weak conformance with ic3ia. Runtimes are cpu time given as median of five samples. Note that weak conformance verification with ic3ia have no bound on the global variables.

Specification	Implementation	Strict Conformance nuXmv cpu time [secs]	Weak Conformance ic3ia cpu time [secs]
Figure 7.8(a)	Appendix B.1	1.12	4.00
Figure 7.8(b)	Appendix B.1	2.64	5.88
Figure 7.8(c)	Appendix B.1	2.46	0.37

A short disclaimer: Statical verification does not fully replace software testing. Software tests have some advantages, in contrast to verification. The tests consider the complete build-process. Thus, the correct translation of the program to the target system by the compiler is also part of the test. Moreover, software tests can be applied to real hardware under the actual environment and conditions. Whereas in the verification, we assume idealized semantics of reactive system (e. g., timer reduction Section 6.3.1), or create assumption in gtts which are violated during operation. In the next chapter, we present the monitor generation for gtt. Monitors permit a *dynamical* verification of the system at runtime. This allows checking that the made assumption is indeed correct, and provides a holistic toolbox for the verification with gtts.

Chapter 8

Runtime Verification with Generalized Test Tables

8.1 Introduction

Motivation. Safety-critical systems are usually validated using testing or static verification to ensure it conforms to its specification. Testing can usually only cover few possible scenarios, and for many systems static verification is infeasible. One reason the lack of relevant information available for static verification, e. g., the missing of suitable models of the environment and context of the system. Another potential problem is that the actual static verification may require too many resources (in terms of time, memory, or effort needed to come up with suitably strong environment models) to be feasible in practice (see the time-outs and out-of-memory in Table 7.14). Moreover, the static verification relies on several assumptions, like an ideal language semantics, perfect accurate timers, instantaneous cycle computation, which might not adhered by the deployment context of the software. The last point, static verification primarily deals with functional properties, because non-functional often tangible without concrete context information.

Runtime verification (or, synonymously, monitoring) [Bar+18], on the other hand, does not suffer from these problems, as the verification subject is inspecting during operation in its expected context. Monitors are software systems, produced from (formal) specifications, that run in parallel to the production code and raise an alarm if the system runs (or potentially runs) into a bad state thus provide a sensible alternative to ensure the dependability and reliability of software systems during operation.

Of course, they have drawbacks: The additional execution of the monitoring

requires CPU and memory resources either on the supervised or a separate system, The former solution may have an impact on the supervised system and can endanger its real-time property. Later one requires a fast communication channel between both systems. Runtime monitors are not capable to cover completely cover the range of temporal logics, especially they are not able to investigate liveness property unboundedly. This limitation is also valid for us, thus, we build runtime monitors which are checking the weak conformance. Like any software, the monitoring software can be buggy itself. To avoid bugs in the monitor components, the generation of runtime monitors from temporal specifications is a well-studied problem (e.g., for Metric Temporal Logic [HOW14], Bounded Linear Temporal Logic [FK09], HyperLTL [Fin+19]).

In this chapter, we present the approach behind the monitor generator tool TTMonitor which generates efficient runtime monitors code in C++ from GTT specifications. It implements new features (which we describe in this paper) that make it particularly suitable for monitoring analysis for workflows where each process step is specified as an individual test table. Trigger-based mechanisms spawn monitors dynamically to allow this specification technique to work. The tool is also part of our collection formal analysis toolbox for PLC verification code.

GTTs are stateful contract specifications that have assumptions (preconditions) and assertions (postconditions) in every I/O cycle. This distinction in the conditions allows us to distinguish a monitor terminating because of a failed assumption (uncovered case) from a monitor halting because of a failed assertion (specification violation). The contract design of GTTs allows us to distinguish four different modes of a monitor:

- *running*—system and monitor in operation, no violation;
- *bailed out*—the specification does not cover this concrete run;
- *failure*—the monitored run violates the specification;
- *finished*—the monitor has finished, the system continues, but cannot fail this specification any longer.

Contributions and Outline. In this chapter, we present an approach by which GTT specifications can be verified dynamically using runtime monitors w.r.t. to their weak conformance. It extends an earlier approach [Cha+17] that was limited to fewer language constructs. In particular, the presented extensions include row groups, strong repetitions, global variables, and nondeterminism (Section 8.2.1). We introduce the concept of *Dynamic Monitors*, by which monitors can be restarted, and can have multiple instances running at the same time

(Section 8.2.3). We present an approach for the hierarchical combination of monitors. This approach allows adding and removing runtime monitors during operation. The hierarchical combination allows a flexible aggregation of monitor results using a variety of functions (Section 8.2.2). We provide TTMonitor, a monitor-generation tool that creates monitors from GTT specifications. The C++ code of the monitor produced by TTMonitor is highly portable as it does not depend on libraries. The tool sources are publicly available as part of VERIFAPS http://verifaps.github.io/. Section 8.3 we give application examples for the use of the newly introduce specification features for the runtime monitoring. We conclude with a discussion of further potential optimizations in Section 8.4.

Distinction to Earlier Work. This work extends and generalizes ideas of generating runtime monitors from GTTs presented by Cha et al. [Cha+17], where the approach was tailored to the specific needs of the domain of automated production systems and did not support row groups, omega repetition, global variables, and nondeterminism. Additionally, in this thesis, we already presented the notion and the construction of *program* in Section 6.2 for static verification. But runtime monitor requires different constructions. The main difference: For the static verification we used three special statements (assert, assume, havoc) to model the tester program. In runtime monitors, we can not use these statements, in particular, we cannot fall back to the nondeterminism provided by the havoc-statement to manage arbitrary states of global variables. For the generation of runtime monitors, we need to find a different trick.

8.2 Monitor Generation

We now explain how a runtime verification monitor is created from a GTT.

Monitor. A monitor is a software module that runs alongside the monitored reactive system and is executed at the end of each I/O cycle—after the output of the reactive system has been computed. It checks whether the trace (consisting of input, output, and internal state values) observed thus far (i.e., the current system state together with previously observed system states) satisfies the given specification. In the case of a monitor derived from a GTT T, the monitor can report one of four cases:

(1) The trace adheres to the specification, i.e., there is at least one sequence of rows in T such that all assumptions and assertions are satisfied (*running*).

(2) There is no sequence of rows in T such that all trace assumptions are satisfied (*bailed out*), i.e., the specification does not cover the observed trace.

(3) There is a sequence of rows in T such that all assumptions of the trace are satisfied, but no sequence satisfies all assertions (*failure*).

(4) The trace adheres to the specification for a sequence of rows in T such that the end of T has been reached (*finished*).

The state *finished* is a special case of *running*, but it is particularly interesting since the monitor can idle as it can no longer change its state (in particular it can no longer fail the specification).

Definition 8.1 (Monitor)**.** *Let $P\colon (\mathcal{I} \times \mathcal{S}) \to (\mathcal{S} \times \mathcal{O})$ be a reactive program with input space $\mathcal{I}$, output space $\mathcal{O}$, and state space $\mathcal{S}$ (Definition 5.1). A monitor $\mathcal{M}$ with internal state space $\mathcal{S}_M$ is a reactive program $\mathcal{M}\colon (\mathcal{I} \times \mathcal{O} \times \mathcal{S}) \times \mathcal{S}_M \to \mathcal{S}_M \times \{\text{OK}, \text{IE}, \text{OE}, \text{FIN}\}$ that takes as input the current input, output, and state values of S and returns as output a verdict. The verdict may be* OK *(for* running*),* IE *(for* extraneous*),* OE *(for* failure*), or* FIN *(for* finished*).*

From an Automaton to a Monitor. We use the automaton definition from Section 6.1.1 to build a monitor $\mathcal{M}(T)$ from a GTT T that realizes such an automaton. Since the automaton can be nondeterministic, $\mathcal{M}(T)$ needs to consider all possible runs, and, hence, has to maintain in its state space $\mathcal{S}_M$ a *set* of current automaton states $\mathbf{S} \subseteq \{s_{error}, s_{assum}, s_{final}, \dots s_k^{(i)} \dots\}$. Note that we slightly diverge with our notation as a new state is introduced: s_{error} marks the failed states (known as $fail$) and s_{final} is the sentinel state (s_{m+1}). Additionally, the state s_{assum} captures the violation of assumptions, which is not necessary for the static verification, thus the state is omitted in Figure 6.3. Its transition function is defined as

$$s'_{assum} \leftrightarrow \left(s_{assum} \vee \left(\bigvee_{i=1}^{m} (s_i \wedge \neg \phi_i) \right) \right)$$

w.r.t. to the transition definitions in Equations (6.2) to (6.4).

The automaton construction is suitable for GTTs which do not use global variables. We derive the verdict $m_T(\mathbf{S})$ of the monitor $\mathcal{M}(T)$ from the current automaton states $\mathbf{S}$ as follows:

$$m_T(\mathbf{S}) := \begin{cases} \text{FIN} & : s_{final} \in \mathbf{S} \\ \text{IE} & : \mathbf{S} = \{s_{assum}\} \vee \mathbf{S} = \emptyset \\ \text{OE} & : s_{error} \in \mathbf{S} \wedge \mathbf{S}_{row} \cap \mathbf{S} \neq \emptyset \\ \text{OK} & : \mathbf{S} \cap \mathbf{S}_{row} \neq \emptyset \end{cases}, \tag{8.1}$$

where $\mathbf{S}_{row} = \{s_{final}, \dots s_k^{(i)} \dots\}$ is the set of automaton states representing a table row or the end of the table. If the condition A_{WC} for the weak-conformance

is violated by a system trace, then the verdict function (8.1) returns OE for that trace. In case that the invariant is satisfied for a (finite) trace, the verdict function can make a more fine-grained statement and return one of the three other verdicts, distinguishing between situations in which the specification does not cover the trace (IE), the end of a specification has been reached (FIN), or the trace runs according to the specification (OK). The monitor construction is designed to maintain conformance (Section 5.3).

Proposition 8.2 (Relation to Conformance)**.** *Let S be a reactive system, T a GTT, and $\mathcal{M}(T)$ the generated monitor. S weakly conforms to T if and only if $\mathcal{M}(T)$ does never produce the verdict OE in any I/O cycle step for any possible behavior of S.*

This property is achieved by reusing the automaton construction and the weak-conformance condition A_{WC}.

Challenges. One challenge of the monitor is to determine the instantiation of the global variables from the observable system state. In contrast to static conformance verification, where a system needs to adhere to all global variables' instantiations, the monitor supervises and assesses only the current trace, where the instantiations (along with the input and output values) of the global variables are determined by the environment and by the system.

In the remainder of this section, we explain how we tackle the following challenges: handling global variables, especially in combination with a nondeterministic row choice (Section 8.2.1); combining multiple GTTs into a single monitor (Section 8.2.2); restarting after bailing out (Section 8.2.3); and monitoring concurrent events and their effects (Section 8.2.3). These topics have solutions for static verification which cannot be transferred to the case of runtime verification.

8.2.1 Global Variables and Nondeterminism

Global variables within a GTT are universally quantified, which works fine for static verification with model checkers. But for runtime monitoring, the monitor needs to determine the instantiation of global variables by observing the current input-output trace. Hence, we need to decide *when* and *to which value* a global variable should be bound.

In GTTs, a global variable can occur at any position within the constraints, e.g., the first occurrence of a global variable g could be $g^2 - 4In^2 = 0$ (where "In" is a program variable of the reactive system). Such constraints could have zero, one, or multiple solutions, hence the value of g may be ambiguous. We tackle

this problem by a syntactical restriction: the first appearance of a global variable g needs to be in a binding equation, where g stands alone on one side of the equation. In our example with the global variable g, the user needs to rewrite the equation, and make the solution bound to g explicit, e.g., "$g = +2In$". In the discussion (Section 8.4), we present two approaches to eliminate this syntactical restriction.

Since time constraints allow rows (and blocks) to be skipped, it cannot be guaranteed that the syntactically first occurrence of a global variable is evaluated. However, it can be statically ensured that the first evaluation of a global variable during a run is within a binding equation. Alternatively, this check can also be performed at runtime by the monitor.

Another challenge for global variables is potential ambiguities induced by nondeterministic tables as multiple rows (automaton states) with different assignments for the same global variable could be active at the same time and thus force a binding to different values. To resolve this challenge, we use a token-based evaluation of the automaton, where each token represents a possible run of the automaton. Each token carries an assignment of the global variables together with its current automaton state. A token is always in a single state, and therefore the value bound to a global variable is unambiguous. If there are multiple possible successor automaton states for a token, the token is duplicated and each copy obtains a different successor state. Because the automaton can be in multiple states, there might be multiple tokens. Furthermore, it is also possible that there are two tokens at the same automaton state with different assignments of the global variables. Two tokens at the same state with identical assignments can be reduced to a single token as both behave identically.

8.2.2 Combined Monitors

Since GTTs are designed to describe a set of similar system behaviors, it is oftentimes not possible to describe the complete system behavior in one table. Hence, the specified behavior of a GTT is only a partial view of the complete system and a more comprehensive specification can be gained by using several GTTs to specify a system. To support such multi-table specifications, we need to support monitoring of several GTTs at the same time. In the following, we show how the generated monitors of GTTs can be stitched together into one combined monitor.

A combined monitor $\mathcal{M}_{T_1,\ldots,T_n}$ is a reactive system which monitors a set $\{T_1, \ldots, T_n\}$ of GTTs by using the monitors $\mathcal{M}(T_i)$ for $1 \leq i \leq n$. The combination essentially runs the monitors in parallel, and the combined monitor state is the tuple of the states of the individual monitors: $\mathbf{S}_{1,\ldots,n} = (\mathbf{S}_1, \ldots, \mathbf{S}_n)$. The most relevant part of the combined monitor is the aggregation function

$\overline{m}_{T_1,\ldots,T_n}(\mathbf{S}_{1,\ldots,n})$ which combines the verdicts $m_{T_i}(\mathbf{S}_i)$ of the sub-monitors $\mathcal{M}(T_i)$ (for $1 \leq i \leq n$) into a single verdict.

$$\overline{m}_{T_1,\ldots,T_n}(\mathbf{S}_{1,\ldots,n}) = \mathrm{agg}(m_{T_1}(\mathbf{S}_1), \ldots, m_{T_n}(\mathbf{S}_n))$$

There are two canonical aggregation functions: $\mathrm{agg}_\wedge$ and $\mathrm{agg}_\vee$.

For the aggregation, we filter out the *bailed out* results from the sub-monitor verdicts ($\mathrm{filter}_{\text{IE}}(\cdot)$), and then the functions $\mathrm{agg}_\wedge$ and $\mathrm{agg}_\vee$ can be defined as the minimum and the maximum functions with respect to the order OE $<$ OK $<$ FIN on the results. Formally,

$$\mathrm{agg}_\wedge(a_1, \ldots, a_n) = \min\{(\}\, \mathrm{filter}_{\text{IE}}(a_1, ..., a_n))$$
$$\mathrm{agg}_\vee(a_1, \ldots, a_n) = \max\{(\}\, \mathrm{filter}_{\text{IE}}(a_1, ..., a_n))$$

with the special case that $\max\{(\}\,\emptyset) = \min\{(\}\,\emptyset) =$ IE. The aggregation functions $\mathrm{agg}_\wedge$ and $\mathrm{agg}_\vee$ correspond to the conjunction and disjunction in a three-valued logic with the given order.

The $\mathrm{agg}_\wedge$ function corresponds to the conjunction of the monitors returning OK if no sub-monitor returns OE and at least one monitor is OK. Similarly, $\mathrm{agg}_\vee$ represents the disjunction returning OK if at least one sub-monitor signals OK. The value IE expresses that a monitor has diverged, and this value is ignored in both aggregations.

In general, aggregation functions can be user-defined functions which are fine-tuned for the given tables and the automation system based on gained experience. For example, we allow complex aggregation functions which compute histograms of the given monitor results and aggregate their results based on a given threshold for each category (e.g., a combined monitor indicating OK implies that at least a given percentage of the sub-monitors are fine (OK) and the number of errors (OE) is below a threshold).

Note that combined monitors themselves can be subject to a combination, which allows the construction of sophisticated combinations. For example, imagine one GTT *emerg* which describes the emergency behavior of a system, and two mutually exclusive GTTs *man* and *auto* covering the manual and automatic operation modes. We can compose a comprehensive specification by logically combining the corresponding monitors for the GTTs, expressing that "*emerg* and *man* or *auto*" should always be satisfied. The corresponding combined monitor is

$$\mathcal{M}^\wedge(\mathcal{M}_{emerg}, \mathcal{M}^\vee(\mathcal{M}_{man}, \mathcal{M}_{auto})) \ .$$

Performance Considerations. The monitor combination could have been implemented as a single product automaton construction combing all constraints

of a set of GTTs. We decided against this product automaton construction, as the implementation effort would be higher and there are no clear performance benefits. States and tokens of and in the product automaton can be saved if the GTTs share initial rows, but this effect is negligible for long-running systems.

On the other hand, if global variables occur in the GTTs, the approach with several individual monitors (and, hence, a separate token for each GTT) is more flexible as each monitor can consider a separate global variable binding. There is an additional drawback: the tokens of the product automaton would need to carry the values of all global variables of the combined GTTs. These large tokens are duplicated on nondeterministic choices. Moreover, the approach of combining individual monitors allows the user to include handwritten monitors and supports dynamic monitors (Section 8.2.3).

8.2.3 Triggered Restarts and Dynamic Monitors

Triggered Restarts. If a monitor $\mathcal{M}$ runs into a situation where its monitored table does not cover the current run, i.e., the assumptions of all currently possible rows are violated, $\mathcal{M}$ does not need to be continued since it cannot recover from that state. Let us call such a monitor *diverging*. Consider a situation where a GTT describes the normal behavior of a system. If an (abnormal) emergency situation has been triggered for the system, the monitor diverges when the abnormal situation occurs since this behavior is not covered by the table. After recovery, it can no longer be used to monitor the system.

This problem was already identified in [Cha+17], and a solution which allows a simple and precise monitoring of event-triggered processes has been proposed there. An additional specification can be provided which triggers a restart of a monitor for a GTT. A restart trigger is a condition ϕ on the current state in the constraint language of the table cells. A monitor $\mathcal{M}$ restarts if it has diverged, i.e., once it results in a verdict of IE, and the observed system trace meets ϕ. The restart resets the monitor to its initial state.

Dynamic Monitors. We generalize the idea of restarting further by allowing—beside a restarting condition—a starting condition ψ for a GTT T. Whenever ψ is met by the current system trace, a new instance of the monitor $\mathcal{M}_T$ is created and started. Note that, unlike the restart condition ϕ, the trigger ψ is not bound to another diverged monitor being stuck in the IE state.

Dynamic monitors can be used to compose event-triggered specifications, where the expected system reaction to the event is described. For example, they can be used to specify the flow of workpieces and tracking the correct processing of each workpiece in the software of production systems. Whenever a workpiece appears at the beginning of the conveyor belt, this event triggers

the spawning of a new monitor which monitors that particular workpiece. With dynamic monitors, it is not necessary to globally formalize the entire work process chain, but rather one can focus locally on each process step for a single workpiece.

A starting condition ψ is evaluated before the execution of the sub-monitors. Therefore, the newly created monitor instances start in the same I/O cycle in which ψ has been satisfied. At the end of a cycle, dynamic monitors which have diverged are discarded to avoid growing memory consumption. As a best practice to keep the memory consumption low, every dynamic monitor should eventually terminate, e.g., the end of the specification is reachable.

The concept of dynamic monitors seems to subsume the concept of restarting monitors. But there is a subtle difference: with restarting, there always exists only one monitor instance which can be restarted after it has diverged, whereas a dynamic monitor can have multiple active instances at the same time.

8.2.4 Implementation

We have implemented our approach in TTMonitor on top of the existing VERIFAPS framework. The generated monitors are in the C++ programming language, mainly due to the availability of a sophisticated standard library and its affinity for embedded systems.

Listing 8.1 shows the interface. The class is parameterized in *io_t*, which is instantiated with a structure containing the (at the least the used) input, output, and state variables of the supervised system. A monitor instance the basic functionality: providing the current state "`state()`" and calculation of the next state ("`next(io_t)`"). The reset method sets the monitor into its initial state. For non-resettable monitors, reset is only called once during the construction.

Listing 8.2 is the implementation skeleton of a generated monitor. We define structures for holding bindings for each global variable, along with a Boolean flag signaling that the global variable is bound. Also, we define enumeration containing the values for the different automaton states. The test table monitor is itself a derived class from our interface. Each instance holds a resizable vector of the current tokens. A token is composite of an automaton state (`state`), and a global variable binding (`globalVars`).

Each monitor implements a `reset` method, that (re-)initializes the monitor to its starting state. The `next` method is called every cycle and cares about restarting (triggered by the reset predicate in `resetCondition`), moving the automaton token forward as well as setting the new monitor state. The output of the monitor is kept *_state*.

The handling of dynamic monitors falls into the responsibility of the combined monitor implementation. Combined monitors (see Section 8.2.2) also,

```
1  template <typename io_t>
2  class IMonitor {
3  private:
4    MonitorState _state;
5
6  public:
7    virtual ~IMonitor() {}
8    virtual void reset() = 0;
9    virtual void next(
10     const io_t &input) = 0;
11
12   MonitorState state()
13   { return _state; }
14   void state(MonitorState s)
15   { _state = s; }
16 };
```

Listing 8.1: The C++ interface of GTT-monitors

follow this class interface, but work differently for the `reset` and `next` methods. On `reset`, the combined monitor removes every dynamic monitor, and also reset every sub monitor. On `next`, it first evaluates the starting conditions of the dynamic monitors and instantiates them if necessary. Afterward, the `next` method on the registered sub-monitors is called. In the end, the combined monitor removes the dynamic monitors, which are *bailed out* or *finished*. Finally, the state of the combined monitor is calculated by the selected aggregation function.

8.3 Application Scenarios

In this section, we demonstrate the specification of reactive systems with GTTs and show how the TTMonitor tool can generate monitors from the GTTs using the presented approach. The chosen examples demonstrate the benefits of the approach in different application contexts for reactive systems. Due to space restriction, the table input files and monitors generated from them can be found in the companion material [Wei21] or on the companion website of the origin publication.[1]

[1]https://formal.iti.kit.edu/nfm2021/

```
1  struct gv_t { /* global variables */ };
2  const struct gv_t gv_t_default;
3  enum state_t { /* automaton states */};
4
5  template <typename io_t>
6  class TMonitor : public IMonitor<io_t> {
7    struct Token {
8      state_t state; gv_t globalVars; };
9    vector<Token> tokens;
10
11   public:
12   TMonitor() { reset(); }
13   void reset() override {
14     this->state(MonitorState::FINE);
15     tokens.clear();
16     // .. add starting tokens
17   }
18
19   void next(const io_t &input) override {
20     if(state() == ASSUMPTION_FAILED
21        && this->resetConditon(input)) reset();
22     // forward each token
23     // calculate monitor state
24   }};
```

Listing 8.2: The implementation skeleton of GTT-monitors

8.3.1 Cruise Control System

A CCS is a driver assistance system found in cars that accurately maintains the speed set by the driver by controlling the throttle-accelerator pedal linkage without driver intervention. If the driver uses the accelerator or the brake pedals, the system releases its control over the velocity. CCSs have already been formally studied [AT11; HKL09; Pre+18]. We follow the specification and Esterel implementation in [YB18]. There are nine input parameters to the system: *On*, *Off*, *Resume*, *Set*, *Speed*, *QuickDeccel*, *QuickAccel*, *Accel*, and *Decel*. The CCS returns three output values: the current operation mode (on, off, stand-by, disabled), the current target speed, and the value of the throttle. The GTT in Figure 8.3 describes those scenarios in which the CCS is switched on and should

#	ASSUME									ASSERT		⏲
	On	Off	Resume	Set	QuickDecel	QuickAccel	Accel	Brake	Speed	CruiseSpeed	CruiseState	
	BOOL	BOOL	BOOL	BOOL	BOOL	BOOL	BOOL	BOOL	FLOAT	FLOAT	ENUM	
1	FALSE	FALSE	FALSE	—	—	—	—	—	—	0	OFF	≥ 0
2	TRUE			FALSE				—	$> SpeedMin$	$= Speed$	ON	1
3	—				FALSE	FALSE	FALSE	FALSE				≥ 0
	Restart: $CruiseState = Off$											

Figure 8.3: GTT for the cruise control system which is restarted when the `CraseState` is set to `off`

maintain the current speed until either the brake or the accelerator pedal are pressed. This monitor becomes obsolete (i.e., it diverges) if the CCS is switched off, and restarts once the system is switched on.

8.3.2 Conveyor Belt Process

In this scenario, we demonstrate the features of dynamic monitors by specifying the material flow inside an automated manufacturing plant. The example is based on the Pick-and-Place-Unit (PPU) developed at the TU Munich [Vog+14]. The PPU was developed to demonstrate methods to manage the evolution of long-running hard- and software. More than 20 scenarios have been designed, and they demonstrate a variety of evolution scenarios typical for an automated production system. We use one scenario (scenario number 13) in which the PPU picks up workpieces from a deposit with a crane. If a workpiece is metallic, it is transported to the stamp to be engraved. Then the engraved workpiece is picked up again and is moved to the conveyor belt, where the workpieces are finally sorted on different ramps. Non-metallic workpieces are not engraved, and are directly moved to the conveyor belt. For optimization, the crane moves

#	ASSUME					ASSERT						⏲
	CranePos	CraneWP	WP@Magazin	StampState	WP@Conveyor	Crane	Vaccum	Stamp	Conv.Belt	Pusher1	Pusher2	
	ENUM	BOOL	ENUM	ENUM	BOOL	ENUM	BOOL	BOOL	BOOL	BOOL	BOOL	
1	MAGAZINE	FALSE	METAL_READY	—	—	STOP	—	—	—	—	—	1
2			—			PICKUP	TRUE					$[1, T_1]$
3		TRUE	EMPTY	FREE		STOP						1
4	—		—			MOVE_CW						$[1, T_2]$
5	STAMP					STOP						10
6						RELEASE						5
7						—	FALSE					1
8		FALSE		OCCUPIED				TRUE				1
9	—	—			—	FALSE		FALSE		—		—
10	STAMP	FALSE		READY		PICKUP	TRUE					$[1, T_3]$
11		TRUE		FREE		MOVE_CCW	TRUE					—
12	CONVEYOR					STOP	TRUE					—
13						RELEASE						5
14	—	FALSE				—	FALSE		TRUE	FALSE		1
15						—						T_4
16											TRUE	5

Figure 8.4: A GTT for describing the material flow in the PPU plant, which is instantiated when a new work piece appears at the magazine ($WP@Magazin \neq$ `EMPTY`).

non-metallic pieces to the conveyor belt while a metallic piece is being stamped.

Due to the parallel processing (stamping, transporting, and sorting) within the plant, a global specification of the input and output variables is hard to achieve. Instead, we can describe the plant by following the workpieces individually.

Note that the assumptions in Figure 8.4 encode the expected physical behavior of the environment. If they are violated, e.g., if a workpiece is not detected in time, the monitor raises the signal (*bail-out*), and this should be interpreted as a flag for an error in the environment. One possibility to deal with this is to deliver more explanations why a monitor diverges, as discussed in Section 8.4.

8.4 Discussion

Counting Repetitions. The automaton for constructing the monitors is generated from a normalized (unrolled) test table. Therefore, a row with a duration $[m, n]$ ends up in an automaton with $n \cdot d$ states, where d denotes the unrolling of overlying row groups. The introduced token-based simulation allows the use of integer variables for counting the repetition of rows and row groups. Using integer variables for counting have a limited impact on the performance; their use would decrease the code size and improve the branch prediction. Moreover, we could get rid of the restriction of nonrigid duration constraint, and allow the use of state or input variables in the duration column. Also, their use enables the usage of clock time instead of I/O cycles number and makes the generated monitors applicable for interactive systems.

Symbolical Representation of Global Variables. In Section 8.2.1 we have restricted the first occurrence of global variables to a form which describes an unambiguous value to bind. This restriction could be lifted, with a negative impact on the performance by using a symbolic representation, e.g. a BDD or a CNF formula. Instead of a concrete value, a token would hold a symbolic representation for each global variable. The constraints of a global variable in the monitored table cells are added to the token's symbolic representation and limits the value range of the global variable. The symbolic representation must be satisfiable (describing at least one possible value of the global variable) during monitoring. Moreover, every monitored constraint needs to be checked symbolically.

A simpler solution might be the use of multiple tokens. Instead of forcing the user to decide on one solution, we create a token for each adhering binding of the global variable for the equation. Back to our example of quadratic equation, we know there are at most two possible solutions, so we would create zero to

two tokens with different assignments. Note that this solution is only possible if the number of solutions is limited and rather small.

Assumptions are Assertions on the Environment. In some circumstances, an assumption violation is an indicator for a serious error, and not just a warning that the specification may not be suitable in the current situation. We make this observation in Section 8.3.2, where a disappearing work piece on a conveyor belt is rather unexpected and indicates either a broken sensor or a standstill. In this case a better explanation (with a more detailed cause) would help to distinguish between a failing assumption which is just a too restrictive or unsuitable specification, and absolute expected guarantees of the environment.

8.5 Related Work

Runtime monitoring has been applied to a wide variety of different systems ranging from programming languages such as Java [MN04a; CR05; Zee+07; CL02]. Likewise, the specification language used to describe the monitored properties are manifold. While some approaches use standard temporal logics such as LTL [Pnu77], MTL [Koy90], or STL [MN04b] (or variations thereof), others rely on specification languages such as JML [CL02]. Additionally, some approaches use their own specification formalism, e.g., [MN04a; PNW11]. In the following we only consider into monitoring of embedded systems.

Pike et al. [PNW11] presents an approach for "Runtime Verification for Ultra-Critical Systems" for distributed hard real-time systems. The authors designed a *stream*-oriented programming for the design and the encoding of the monitors, called *Copilot*. Such a Copilot monitor program is an event-based description containing a sequence of triggers, which itself consists of name, Boolean guard, arguments, and implementation body. If the guard of a condition is met, its implementation is executed. Copilot programs are compiled into C-programs. The authors successfully demonstrated their approach on a real-time air speed measuring system with five measuring units and four computation nodes. The result of each distributed monitor aggregates its result with the results of the other monitors in a fault-tolerant fashion.

D'Angelo et al. [DAn+05] also uses a stream-oriented specification language, called *Lola*, for their monitoring solution of embedded systems. A Lola specification is a set of equations and triggers over streams. A trigger is just a marker for a Boolean expression over streams, which adherence should result in a warning. For example, the following specification of [DAn+05, Example 5] states that "the

number of a's must always be less than the number of b's.".

$$s = s[-1,0] + \text{ite}((a \wedge \neg b), 1, 0) \quad +\text{ite}((b \wedge \neg a), -1, 0)\mathbf{trigger}(s \leq 0)$$

The generation of monitors may be difficult due to the dependency between the equations. Solving such a set of equations requires finding a fixpoint in which every equation is reduced to a constant. They demonstrate their monitors on the PCI bus protocol and a memory controller.

Bloem et al. [Blo+15] propose the construction of *shields* which are monitoring instances which are allowed to override the output of a reactive system under observation if the systems violate some formal constraints. To achieve this, they introduce a new notion of k-stabilization which captures the idea that a system can alter the output of a system for k steps to avoid the violation of given properties. For the specification of properties, Bloem et al. [Blo+15] uses *safety* automata $\mathcal{A} = (Q, \Sigma, \delta, s_0, F)$ and their accepted language. A safety automaton accepts a word $w \in \Sigma^\omega$ over alphabet Σ if every state s visited while processing w is an accepting state $s \in F$. Their main contribution is the synthesis of reactive systems (shields), which take over control of the output variables for k-cycles if the supervised system fails. To obtain such a shield, a synthesis problem is established via a two-party game. The winning region of the system (states in which there is a winning strategy) determines the possible valid strategy to react to the environment input.

Baader et al. [BBL09] presents a runtime verification approach which uses a combination of LTL and description logic, called $\mathcal{ALC}$-LTL, for the specification of the monitor. In $\mathcal{ALC}$-LTL, the basic propositions are $\mathcal{ALC}$-axioms in the description logic. The description logic allows the union, intersection, and negation as well as the universal and existential quantification. The monitor construction of an $\mathcal{ALC}$-LTL ϕ is reduced to the construction of two *Generalized Büchi Automaton* (GBA) for ϕ and $\neg\phi$. The final monitor is a Moore automaton which output expresses whether the specification is adhered to or violated $\neg\phi$, as well as it is currently unknown. The output is determined by the deterministic GBAs, in particular, by the presence of active runs in the automata. Using the Description Logic allows dealing with incomplete knowledge. The authors only give a rough idea, that their approach could be used in combination with medical ontologies, like *SNOMED CT*, to monitor the medical status of patients and alarm doctors if needed.

Grimm et al. [Gri+12] also use Description Logic, but now, for constructing monitors for industrial use cases. For this, they build a lightweight reasoning engine for the $\mathcal{EL}+$-fragment on a PLC system. This fragment only allows the implication, intersection, and existential quantification besides the typical use of concepts and relations. The approach allows modelling static aspects of

the systems, and to add (or remove) dynamical information over time to the knowledge. For example, $Fan \sqsubseteq \exists shows\ .\ Vibrations$ is added if a sensor recognizes vibration at the fan. If we can reason that a faulty state can occur from the current knowledge base, we raise an alarm. In contrast to [BBL09], no (relative) timing domain is considered.

8.6 Closing

We presented an approach for generating runtime monitors from GTTs, which can deal with nondeterminism and global variables. Moreover, we introduced the concept of dynamic monitors which are created/launched at runtime whenever a specified trigger event occurs. They make possible a local specification of parallel and multi-step processes. We show the applicability of the monitoring approach on concrete examples from the domains of automated production systems and embedded controllers. The approach has been implemented in TTMonitor, an open-source tool which generates monitor code in C++ from GTT specifications.

GTTs have two distinct kinds of constraints: assumptions and assertions. Depending on the type of constraints which fails, a failing trace is reported to either diverge (i. e., the specification does not cover it) or to reveal a flaw in the implementation. This principle can be refined further in future work by the introduction of several constraint categories in addition to assume and assert. This will allow the monitor to elaborate on the nature of failures even further and give more detailed feedback to the engineer. For instance, for each hardware component, a category could be introduced for the assumptions on its physical response behavior. If a failure is reported in this category, this will directly indicate that the hardware component has failed. Analogously, also for the assertions on software components.

Runtime monitors from GTTs are a valuable addition to the static verification. The static verification helps to detect early design flaws, and the runtime monitors help to ensure that the expected assumption of the verified GTTs are met during operation.

Chapter 9

Conclusion and Outlook

We have introduced the theoretical foundations and the decision procedures, and presented the evaluation of GTTs. In contrast to software testing, GTTs allow a formal specification and support the static or runtime verification of usage scenarios. Moreover, GTTs are applicable early in the development process; the only requirement for their application is the existence of the software. The shape and notions of GTTs are defined by several publications [Cha+18b; Wei+17; Bec+17]. This thesis is the first presentation of GTTs in a closed and combined form with all previously published features. The runtime verification was recently accepted. The verification backed up by C-verifiers were not presented previously.

This thesis focuses on the theoretical notions and the verification of GTTs. We explicitly left out other GTTs-related publications. In particular, [Cha+17; Cha+18a], and the applicability study in [Cha+18b] will be part of a different doctoral thesis [Cha21].

Overview. We use this chapter for the closing remarks of the functional verification with GTTs. We start with the discussion of the weaknesses and strengths of our approach (Section 9.1). Then, we beyond the current theoretical aspects of GTTs, and try to imagine what could be possible steps to overcome some of the presented limitations of GTTs. Firstly, we generalize the row groups to arbitrary jumps to between rows of the same or different test tables (Section 9.2). We call this new kind *meshed* GTT. Secondly, we replace GTTs completely with an arbitrary specification notion (Section 9.3), similar to a (UML) state diagrams, or an SFC: What is the required interface for the game?

In the common conclusion, we state how the idea of GTTs influences upcoming projects (Section 13.3). This chapter is mainly dedicated to ideas that go

beyond the features of GTTs.

9.1 Weaknesses and Strengths

Weaknesses in Theoretical Aspects. In the main chapter, we omit some smaller results that are unsurprising for an informed reader. For example, GTTs are not an ω-regular language description. ω-regularity means that a given notion can describe (or accept) every language, which can be described by a regular expression of ω. Note that ω-regularity coincides with accepted languages of nondeterministic Büchi automaton. In contrast to regular expressions, it is obvious that the union ($L+L'$) of two languages L, L' is not expressible in a GTT. Whereas the repetition and sequential composition exists directly (cf. Definition 5.7). Note for a comparison of languages and GTTs, we would use only the assertions of the GTT, and leave the assumptions empty.

Also, GTTs are not closed under intersection, union, or negation. Closed under an operation $\otimes$ means, that given two GTTs T_1, T_2, we can find a third GTT T_3 with $T_3 = T_1 \otimes T_2$ s.t. T_3 combines the conformance T_1 and T_2. For example, a system conforms to the GTTs T_1 and T_2 if and only if it conforms T_3 (where "$T_3 = T_1 \owedge T_2$"). Or for negation: a system does not conform a GTT T if and only it conforms "$\neg T$". These operations are possible for GTTs if we work with the Büchi-automata generated from GTTs, but finding a GTT is not always possible. The benefits of being closed under different operations are clear. For example, if GTTs would be closed under intersection, we can avoid verifying a system against multiple GTTs individually. Instead, we could construct a single (all-including) GTT, which may give us a performance advantage.

A simple construction principle by merging tables row-wise is blocked by the structure of the duration constraints, mainly by different nesting of row-blocks and different repetitions constraints. For intersection, we can establish such construction for tables with equal-structured duration constraints. This construction exploits the available universal quantification by introducing a new global variable t of type Boolean. Under the restriction of the time constraints, the tables T_1, and T_2 are merged row-wise. A merged cell receives the content $\mathrm{SEL}(t, c_1, c_2)$, where SEL is the case distinction (if-then-else), whether the constraint c_1 from the first table or c_2 from the second table is considered. As a system needs to conform every instantiation, it needs to conform $T_1 = T_3[t/\mathrm{false}]$ and $T_2 = T_3[t/\mathrm{true}]$.[1] This construction is not possible for union or negation (under the same assumption). For both cases, an existential quantification is needed. For example, if a system conforms to $T[\sigma]$ for all in-

[1] $T[t/s]$ denotes the substitution or instantiation of a global variable t with the term s.

stantiations σ, then it only needs one instantiation μ for $!T[\mu]$ to conform with. Existential quantification is discussed in Section 5.4.2.

Missing Support of Modular Verification. Whereas concrete test tables are used to describe a single finite test scenario, we can use GTTs to describe a family of (finite and infinite) test scenarios. But these scenarios might only be a subset of the full system behavior. Therefore, to fully describe a system we need to use multiple GTTs. From this, two issues arise: First, is a set of GTTs a complete specification of a system, or there any input sequences that lead to unspecified situations? Second, there a contradictory situation between two (or more) GTTs? This is the reachability of a situation in which any response of the system is invalid. This also corresponds to the question of realizability in program synthesis.

A consequence of the partial behavior description is a lack of supporting modular verification. Modular verification means we use the GTTs as a contract for a (reactive) subsystem in our reactive system, and during the verification, we would rather apply the contract of this subsystem rather than its actual implementation. Application of a contract means, that the actual implementation is abstracted by complemented nondeterminism, but contract-conform, behavior. Also, we need to prove that the actual implementation of the subsystems adheres to the applied contract. If the contracts are lightweight in contrast to the actual implementation, we receive two smaller verification goals.

But a modular verification is hard to achieve because of the required amount of GTTs for a full description, the raised issues, and the required tool support to maintain these GTTs. From the current perspective, we do not suggest GTTs for this case.

GTTs' Niche. GTTs are weak from a theoretical perspective (not closed under Boolean operations, and not ω-regular). Their strengths are comprehensibility and understandability. Both arise when the GTTs-specific features are triggered, which are (1) the use of the abbreviations, (2) a *dense* specification, and (3) sequential behavior. To the first point: The use of abbreviations allows to create a smaller and more concise specification, where the asserted or assumed conditions for each variable can easily be surveyed. Writing complete Boolean formulas into the table cells negates this advantage. To the second point: consider an LTL specification $\mathbf{F}in$, where we state that finally, the variable `In` becomes true. Such a specification fits well into LTL, but is tedious in a GTT (cf. Figure 9.1). Moreover, the cells of a GTT should be rather occupied with constraints than left empty. A sparse GTT which mainly contains only don't-cares "—" requires a

#	**ASSUME**	**ASSERT** **In**	⏲
1		—	≥ 0
2		=TRUE	1

Figure 9.1: A GTT equivalent to $\mathbf{F}In$ under strict or cooperative conformance.

lot of space for little information.[2] To the third point: GTTs makes it simple to state and understand sequential behavior. Row-groups are a valuable addition, and sometimes required to achieve concise results, but a deep nesting is also an obstacle for the readability. Thus, row groups should be used sparingly.

Practical Limitation. GTTs are not scaling well with the number of columns. Thus, specifications with many variables are hard to read (and to print in a publication). This is one reason for user-defined projection functions on columns. Note that we often needed to strip the presented GTTs in the size and the name of the variables to make them printable on a book page. The table rows are scaling far better, but not limitless. But for consolation, if a GTT is too large to be readable, a comparable formula in a temporal logic would also require its space and effort to be understood. But for large GTTs, we have the chance to increase the readability with better tool support. For example, spreadsheet applications allow to hide rows and columns or to fixate parts of the screen. Additionally, we can present the table in different graphical representations, e. g., state charts, or add additional annotations and comments.

To summarize, GTTs are excellent for the specification of sequential behavior with frequently changing constraints for a reasonable number of variables. GTTs are not a good replacement for invariants or (simple) LTL formulas.

Open Research Question. One of our claims is that GTTs are a comprehensible and understandable specification methodology. Currently, the main support is given by deductive and didactic argumentation: Engineers can start with their concrete test tables, and add new generalizations as they need and understand them. If they reach a point in which they are mentally overloaded by the complexity, they can easily go back to an earlier simpler version. This allows an adaptive and incremental verification process, that starts with one concrete scenario which is incrementally extended and generalized.

The previous claim needs support by empirical data. An experiment was performed by the project partner from Munich. In this experiment, students

[2] We use *dense* and *sparse* in reference to matrices.

were split into several groups, and had to fulfill specification tasks on paper. The goal was to compare comprehensibility and understandability of GTTs against Petri-Nets. Note that due to the experiment design (paper task) and participant group (students), there exists a bias which prevents a direct transfer of the results on the daily use of aPS engineers. The publication of this experiment is currently under revision [Cha+], and will be part of the companion thesis [Cha21].

For us, the questions on the theoretical foundation of GTTs are sufficiently answered. Hence, we mainly identify the following open research questions, which require empirical evaluation on real-world scenarios, to guide further necessity in the foundation and presentation of GTTs.

- Are GTTs understandable in comparison to other specification techniques?
- Which (set of) GTT features are required for industrial-used and are preferred by engineers?
- How should a tool for the specification and verification of GTTs should be designed?

9.2 Meshed Generalized Test Tables

In this section we add a new feature to GTTs: we allow arbitrary jumps between table rows, like `goto`-statements in programming languages. In the current version of GTTs, jumps are given implicit either by defining a row group, or by the order of the table rows. Now, the user can specify arbitrary jumps between rows in a single or multiple GTT. We call the new concept *meshed* GTTs. The jumps are specified by a drawn edge which is protected with a guard (Figure 9.2).

Structure and Semantics. In the definition of meshed GTTs, every jump is explicitly specified. Of course, not every jump, like jumps between consecutive rows, is later drawn in a graphical presentation. Formally, a meshed GTTs is similar to a graph of rows:

Definition 9.1 (Meshed GTT)**.** *A meshed GTT $T_{\mathcal{M}} = (R, r_0, E)$ is a tuple where*

- *R is a set of pre- and post-condition pairs (ϕ, ψ),*
- *$r_0 \subseteq R$ are the initial rows, and*
- *$E = R \times G \times R$ are the jumps between the rows.*

where $G := \{\texttt{pass}, \texttt{fail}, \texttt{miss}\}$ denotes the set of valid guards, and ψ_i, ϕ_i are Boolean formulas.

The guards on the jumps are limited to `pass`, `fail` and `miss`. These guards correspond to the transitions in the automaton, where a transition is taken when the row is successful adhered (`pass` with "$\phi \wedge \psi$"), or the assumption is violated (`miss` with "$\neg\phi$"), or analogue for the violation of assertions (`fail` with "$\phi \wedge \neg\psi$"). Of course, one can consider Boolean expression as guards, but for our purposes this formalization is sufficient.

It may not so clear, that this formalization of meshed GTTs also subsumes our formalization of GTTs in Definition 5.7. One reason is the dismiss of the duration constraints.

Proposition 9.2. *For every GTT $T \in \mathbb{T}$ exists a meshed GTT $T_{\mathcal{M}}$ such that an arbitrary system S conforms T iff S conforms $T_{\mathcal{M}}$.*

For the proof, we use the normalization of GTTs in Definition 6.1.

Proof. Let T_0 be the normalized GTT for the given table $T \in \mathbb{T}$. Thus all duration constraints are either $[1,1]$, $[0,-]$, $[0,1]$ or ω in T_0. To show our property we need to translate these time expressions into jumps. For the translation we can exploit the construction of the automaton.

Then, $T_{\mathcal{M}} := (R, r_0, E)$ can be constructed in the following way: R is the set of all rows in the normalized table T_0, r_0 are the first reachable rows (given by the successor function $succ(0)$ in Definition 5.8), and E is derived from $succ$:

$$E := \{(r, \texttt{pass}, s) \mid s \in succ(r) \wedge r, s \in R\} \ .$$

□

Note that for the construction we only require the guard `pass`, the guards `fail` and `miss` are an extension and allow the specification in cases of a constraint violation, for example, useful for the specification of fault handling in production systems.

9.2.1 Graphical Representation

A meshed GTT is very close to the automaton (Section 6.1.1). The only missing parts are the explicit sentinel and error state, and the closure of directly reachable rows (given by the successor function $succ$). But for the graphical representation, we do not want to slip into the graphical representation of the automaton with states and Boolean expression. Instead, we want to keep the nature of tables, but only with some extra edges.

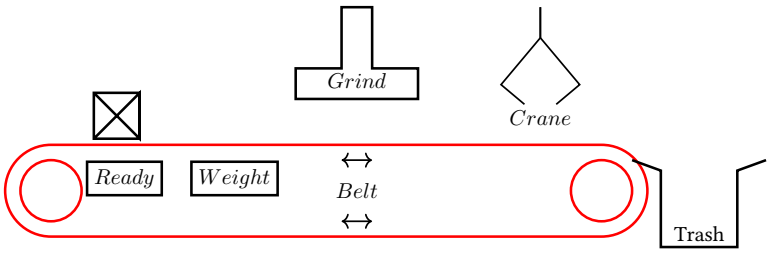

Figure 9.2: Grinding and sorting processing unit

Example. We want to illustrate a possible graphical appearance by an example. We consider a processing unit of workpieces, which ensures the specific weight of the workpieces. Too light workpieces are sorted in the trash bin, too heavy workpieces can be rescued by grinding them until they meet the weight requirement. Good workpieces are given to the next processing unit. Figure 9.2 shows the linear hardware setup on a conveyor belt. Workpieces are entering on the left side of the conveyor belt recognized by the sensor `Ready`. Besides this sensor, we have a weighting scale `Weight` measuring the weight of the workpieces. Transport between the detector and weighting scale is needed to obtain the weight. Further to the right side, we have the grinder (for reducing the weight), a crane (for picking up good workpieces), and the trash bin (which receives the bad workpieces).

Implicit Notions and Normalization. Figure 9.3 shows the specification of our example. For readability, we omit the guard on `pass`-edges and make it the default. Also, implicit jumps between consecutive rows are omitted. We draw edges for `miss` or `fail` differently, in the example we used a dashed line for `miss`-jumps. We allow the use of duration constraints on rows in the graphical presentation. Later, these rows are expanded during the normalization. We need to make clear how the incoming and outgoing jumps on the *aggregated* rows with duration constraints are applied on the expanded rows. We define: an incoming jump addresses a row only in its first iteration (regardless of the guard). Outgoing jumps are more differentiated. The `pass`-jumps respect the duration constraint, s.t. such a jump is only taken if the duration constraint of the (outgoing) row is met. The handling is done during normalization, only those expanded rows receive the outgoing jump if their associated iteration meets the constraint. Whereas, `fail` and `miss` jumps are always taken regardless of the duration constraints and the current row iteration. If a strong repetition appears in the duration constraint, all outgoing `pass`-jumps to different rows are invalid (other `miss`- and `fail`-jumps remain valid). Also, we add an implicit jump from the last rows of each table to the sentinel state if there is no explicit `pass` jump

given on the last rows. Thus, if a play reaches the bottom of at least one table, and there is no defined edge for the continuation, the system wins.

In our example in Figure 9.3, we use four different GTTs to express the different modes of the software: ready for a new workpiece, grinding to lose weight, deliver to trash, or picking up good workpieces. These tables are connected by multiple jumps. For example, if a new workpiece has arrived, it is transported to the weighing scale. Then three possible situations could appear: the weight is too light or too heavy or in range. Thus, we nondeterministically branch into the other three tables. These tables check if they are responsible for the current situation and then proceed with their remaining rows. Otherwise, the assumption is violated and the first row (and the table) is discarded. At the end of the grinding behavior, we use `miss`-jumps to jump to other tables if the workpiece is not too heavy anymore, and a `pass`-jump to repeat the grinding otherwise. Note that this table constellation runs forever, as every table has an outgoing `pass` jump in the last row.

To focus on newly introduced jumps, we used constants for the timing constraints, and avoid row groups. Of course, later can be easily added to the graphical representation as row groups are expanded during the normalization.

9.2.2 Semantics and Decision Procedure

The semantics of GTTs are still valid for the meshed GTT. We only need to reduce a meshed GTT into the set of unrolled instances $\mathrm{SP}(T_{\mathcal{M}})$. This set corresponds to all possible finite paths that reach the sentinel row, and all infinite paths over rows $r \in R$ with respect to the jumps defined in E. Global variables are quantified across all tables in a meshed GTT.

For reusing our decision procedure, we need to translate our meshed GTTs into an automaton, similar to Section 6.1.1. In comparison, the construction for meshed GTTs is simpler: Every $r \in R$ is a state s_r in the automaton. And each jump $(r, g, t) \in E$ becomes a transition between the corresponding states s_r and s_t where transition condition c depends on the guard g and the assume- (ϕ) and assert-condition (ψ) from the outgoing row r:

$$c = \begin{cases} \psi \wedge \phi & \text{if } g = \texttt{pass} \\ \psi \wedge \neg\phi & \text{if } g = \texttt{fail} \\ \neg\psi & \text{if } g = \texttt{miss} \end{cases}$$

Additionally, the implicit error and sentinel state with their implicit edges are added. This automaton needs to be encoded into formulas similar to Equations (6.2) to (6.4). Meshed GTTs are already supported by our verification tool GETETA.

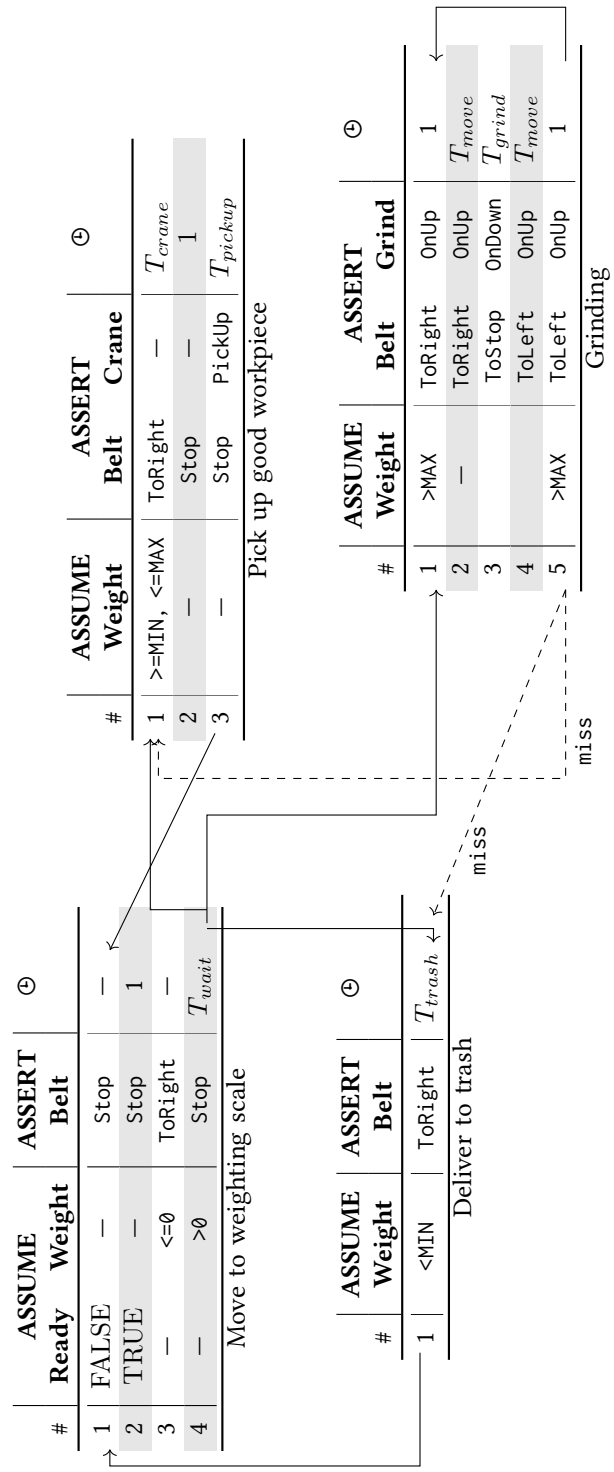

Figure 9.3: Example for a meshed GTT

9.3 Generalising the Game

Meshed GTTs are similar to the SFCs. The idea behind SFCs is to combine sequential program behavior by using edges with guards. The combination of sequential program behaviors is either exclusive (one of the successive behaviors is taken) or parallel (all successive behaviors are taken). In contrast, the sequential parts in meshed GTTs are the test tables and by the nondeterministic branching, we also obtain a parallel *execution*. This leads to the idea, why not using a completely different – maybe well-established – graphical model for the specification representation instead of our tables?

In this section, we generalize the game behind GTTs to obtain a specification language-neutral semantics. These semantics should then adaptable to other specification notions, e. g., SFCs, test tables, or state machines. We model that the specification notion has a state, e. g., for storing the current positions in a test table or the active steps in an SFCs. In the following, we use $s \in \mathcal{S}$ as a notion for this opaque *specification state*. Our generalized game relies on two functions, $contracts$ and $update$, which need to be defined accordingly to the specification notion.

Formalizing the Generalized Game. Let $\Sigma = InVar \cup StateVar \cup OutVar$ be the variable signature of the program to be verified, and Fml_Σ be the set of Boolean formulas over Σ. The function $contracts$ maps the current state $s \in \mathcal{S}$ of the specification to a sequence of formula pairs over Σ:

$$contracts\colon \mathcal{S} \to Seq(Fml_\Sigma \times Fml_\Sigma) \ .$$

$Seq(X)$ expresses the set of all finite sequences over the elements in X. Also, let $eog \in Fml_\Sigma \times Fml_\Sigma$ a constant as a signal that the end of the specification has been reached – similar to the end of the test table.

The function $update$ computes the next specification state by using the old state $s \in \mathcal{S}$, the result of the evaluation $r \in \mathcal{S}eq(R)$ of the given contracts and the chosen input values $i \in I$ and computed output and state values $o \in O$.

$$update\colon \mathcal{S} \times Seq(R) \times I \times O \to \mathcal{S} \ ,$$

where $\mathcal{R} = \{pass, fail, miss, eog\}$ is the result of the evaluation of a given pre- and post-condition pair. The function $eval$ defines the evaluation of such a pair $(\psi, \phi) \in Fml_\Sigma \times Fml_\Sigma$ on the observed sequences of input values $\bar{i}$ and output

$\bar{i} := \varepsilon$;
$\bar{o} := \varepsilon$;
while *true* **do**
  $i := C()$; $o := S(i)$;
  $\bar{i} := \bar{i} \cdot i$;
  $\bar{o} := \bar{o} \cdot o$;
  $c := contracts(\mathcal{O})$;
  `/* assumption:` $|c| > 0$ `*/`
  $r := \mathrm{map}(eval, contracts)$;
  if $eog \in r$ **then**
    return *System wins*;
  else if *only* $miss$ *in* r **then**
    return *System wins*;
  else if $fail \in r$ *and no* $pass$ *in* r **then**
    return *Challenger wins*;
  `/* Game goes on, as there is at least one` $pass$ `in` R `*/`
  $s := update(s, r, i, o)$;
end

Figure 9.4: Generalized game. C represents the nondeterministic challenger, and S the system, respectively.

$\bar{o}$:

$$eval(\psi, \phi) := \begin{cases} eog & : (\psi, \phi) = eog \\ pass & : \bar{i}, \bar{o} \models \psi \wedge \phi \\ fail & : \bar{i}, \bar{o} \models \psi \wedge \neg\phi \\ miss & : \bar{i}, \bar{o} \models \neg\psi \end{cases}$$

The positions in $r \in Seq(R)$ correspond to the contracts returned by the function `contracts`, thus, the nth entry in r is the evaluation of the nth contract.

The current input and output as arguments of the update function allow a more fine-grained decision of the successor state in the chosen specification notion.

Figure 9.4 shows the new game using the new vocabulary. Note that we use the ordinary map function to evaluate the sequence of contracts. The case distinctions correspond to the cases in Figure 5.1 adapted to the sequence of contract evaluations r. The game ends in favor of the system if the end of the specification is reached (denoted by eog), or all assumptions are `missed`. The

game continues only if at least one contract is passed—otherwise, the challenger wins.

It should be obvious, that the generalized game can easily be instantiated for GTTs using the generated automaton: The automaton states determine the active contracts where $contracts(s)$ returns a sequence of (ψ, ϕ) of the rows which corresponding automaton states are active in state s. If the sentinel state is active in s, eog is returned in the sequence. The function $update$ selects the successor state of each automaton state which corresponding contract `passed`. For `failed` contracts the error state is selected.

The generalized game has one *flaw:* What happens if the underlying specification says, that no contract is suitable for the current situation?. For GTTs, the system wins because this situation can only be reached if all input assumptions were missed in the previous round. We need further requirements on the specification notion.

Requirements for a Specification Notion. We can categorize the requirements into two categories: Requirements for the well-definedness of the game, and requirements to create a decision procedure.

The game definition requires that the specification notion is periodical executable, similar to the reactive system, and in each turn, a non-empty finite sequence of contracts is computed. The formulas in Fml_Σ can be evaluated given the history and current state of the system. Also, the specification notion must be causal (not depending on the future) and deterministic. Theoretical, the specification can be written in any programming language, and the contracts can be in any logic. Non-termination in $update$ or $contracts$, or in verification subject causes the game to be stuck, thus the system never loses, thus, weakly conforms the specification.

We need further restrictions for receiving an automatic verification procedure. We need to able to encode the specification notions into a (Boolean) transition system. Hence, the state space and input space of the verification subject and the specification notion need to be finite. Also, this enforces a bound on available contracts. Additionally, the interpretation of formulas Fml_Σ (in which the contracts are stated) needs to be decidable. But this restriction is not restrictive with respect to the finite variables.

Sequential Function Charts as a Specification Notion. For the last point of this section, we briefly elaborate on the idea of Sequential Function Charts (SFCs) as a possible specification notion. SFCs are a graphical programming language in the aPS domain, thus it should be easily understandable for engineers. A possible instantiation of the generalized game with SFC could be defined as

follows: We associate an assume- and assert-pair (ϕ, ψ) with each step in the SFC. If a step is active, the function $contract$ considers its associated contract as part of the returned sequence. For the update of the specification state, we would use the normal operational semantics of SFCs. In the variable signature, we introduce a new variable for each step X: $X.\mathtt{r} \in R$ which contains the outcome of the contract for the step named X. This specification notion allows the use of any element of the IEC 61131-3, especially variables, assignments, and timed actions. Of course, this is only one possible interpretation of SFCs as a specification notion in this framework. Others may exploit a stronger use of the timed actions or local variables.

Part II

Relational Verification

Chapter 10

Relational Test Tables

Motivation. Relational specifications allow the formalization interesting and practical-relevant properties by specifying a relation between two or more program runs (cf. k-safety hyperproperty in [CS08]). Two important applications of proving relational program properties are *regression verification* (which is also subject of our considerations in Chapter 12), and assurance of *secure information flow* (non-interference, cf Chapter 11).

Regression verification is a generalization of the equivalence proof between two programs, where two program revisions, often the old and new version, are shown to be related under a certain input relation [Bec+15]. With regression verification, we can recognize the introduction of unwanted behavior during the software evolution. For example, to describe the equivalence of two versions of a reactive program P and P', we may state that program runs of the two versions are state-wise equivalent in their output values if their input values are equivalent, formally expressible as

$$\forall i \in \mathcal{I}^{\omega}.\ \mathrm{B}(P)(i) = \mathrm{B}(P')(i) \ \ ,$$

where the program runs are given a set of traces $\mathcal{B}(P)$ and $\mathcal{B}(P')$ of the programs P and P' following in our notation in Section 5.1.

This notation is limited to the special case where equal input provides equal output—a rare case in software evolution. Releasing a new software reversion often brings changes: a bug was fixed, new optimization, or handling of the new hardware. In Section 7.3.2, we evaluated an evolution of the Pick-and-Place where a new behavior for optimization was introduced. With GTTS, we can express the new functional behavior, but we are not able to express or verify that offsides the new behavior the system behavior has not changed. We want to remedy this shortcoming on the relational specification of reactive systems with introduced *Relational Test Tables* (RTTS).

Formal verification for proving functional properties of reactive and PLC systems is well-studied but is rarely used in practice. One of the main obstacles is the lack of appropriated specification languages [Pak+16]. Relational verification is a promising field to reduce the amount of formal specification by using existing software. Instead of specifying how the system should behave, we use a similar system, e. g., the previous version, and specify the condition under which both should behave equal (or similar).

Non-interference is a different interesting relational property, which enforces that specified secret information does not influence the public-observable output [Den76]. Proving the absence of a secure information flow ensures that the software does not leak any secrets to an attacker. In a wider area, (non-secure) information flow can be used to detect dependent variables and software modules on a semantical level. Both properties (equivalence and information flow) are specified as a relation between two program runs.

Contribution and Overview. In this chapter, we propose an extension of GTTs: the *relational* test tables (RTTs). RTTs allow the specification of relational (and functional) properties of reactive systems in a practical and comprehensive formalism. They can be used for relations on any number $n \geq 1$ of program runs. They also are backward-compatible to GTTs ($n = 1$).

We make three extensions to the syntax of GTTs (Section 10.1). First, we introduce a reference to the variables in different program runs. Second, we lift the projection function in the column headers to multiple program runs. Third, we include the control columns, which allow intervening the regular program flow. The control columns require an augmentation of the original programs.

Besides the specification, we also present a decision procedure and the semantics of RTTs in Sections 10.2 and 10.3. For the verification of RTTs, we follow the approach of building product programs [BCK11], which reduces the problem of verification of a relational property to the verification of the functional property. The product program executes all original programs synchronously (also known as *lock-step*). The control columns allow us to break up this synchronous execution—allowing us to handle *stuttering* or repeated behavior. In our case, we reduce the verification of RTTs to the verification of GTTs.

Moreover, we show the applicability of RTTs with different examples from the domain of automated production systems. In Section 10.4, we specify and verify three change scenarios, and an information flow scenario. All examples have been successfully verified using a model checker.

10.1 Syntax

Barthe et al. [BCK11] describes the construction of product programs. This principle allows reducing the relational verification of k-safety properties to functional verification, for example with GTTs. But, this principle alone does not give us a comprehensible and expressive specification language. For example, a relational specification thrives on relations between the variables (e.g. $x < y$, $y \neq x + 1$). In GTTs, we only have a column for program variables, but not a column for expressing the relation between program variables properly. Also, the specification can only express a relation between (perfectly) synchronized programs (lock-step), because each reactive system makes a step at the same time.

Our introduced features focus on the syntactical side of GTTs to overcome these limitations (Section 10.1). Later in (Section 10.2), we use Barthe et al. [BCK11] to define the semantics of RTTs with the reduction to GTTs by building a product program from the augmented programs. In Section 10.1.3 we show the extension in action.

RTTs inherit all the syntactical elements of GTTs (Chapter 5). Especially, an RTT contains columns designated to input, state, and output variables of each underlying program run, but only one common duration column.[1] To tackle the previously stated limitation, we add the following syntactical concepts: (a) (Explicit) references to variables in specific program runs, including new abbreviations. (b) We introduce a new column type: the *control* column. This column type allows the break-up of the synchronous specification. (c) Lifting projecting functions in the column header to multiple program runs.

To make our notation of programs and their runs more clearly, programs are a syntactical construct given in form of source code, its semantics is transition within an execution cycle (Definition 5.1), which is lifted in Definition 5.2 to an infinite sequence of output values for a given infinite sequence of input values. This output sequence is generated successively by a program run. We rather use the newly introduced notation program run (than reusing the trace semantics) to make clear that we can manipulate the control flow of the program runs.

10.1.1 Relational References

An RTT specifies a relation between a list of program runs. Each program run is identified by its index position in the list, and additionally, by a user-defined name. For example, in the case of regression verification we often use *old* and *new* to identify the old and new software revision (see the example in Figure 10.2).

[1] The duration column is part of the specification, whereas the variables are part of their program runs.

Table 10.1: Constraints abbreviations for relational references.

Notation	Value if used in column for p»X
q»Y	Y in program run q
»Y	Y in the other program run
q»	X in program run q
»	X in the other program run
Y	Y in program run p

Program runs can origin from the same program (see Section 10.4.4), but all runs are executed independently of each other. The number of program runs, their names, and the corresponding program are denoted separately, and are not part of an RTT.

By using the name or index number of the program runs, we can identify variables explicitly. A variable X in a program run p is denoted by p»X, where p is either the name for the index of the program run. As variable names may be ambiguous, we use the fully-qualified variable names, like p»X, in the column headers. The expression grammar (Section 4.2.1) of GTTs are still valid, but extended s.t. p»X is valid variable identifier. For example, p»$X[-n]$ still refer to a previous value of X in the program run p of n cycles ago.

As the case of two program runs is very prominent, we introduce the abbreviation "»X" (the program run identifier is omitted) to refer to X in the "other" program run if there are only two program runs. The notation "», where the variable name is also omitted, references the same variable name in the other program run, i.e., » equals q»X if it is used in the table column for variable p»X and p, q are the only two program runs. Additionally, we keep the old notion: a simple name X refers to the variable X from the same program run. Table 10.1 shows an overview of the abbreviations for relational references in comparison to Table 4.3.

10.1.2 Control Column

We introduce a new column type: the *control* column. The control columns allow to manipulate the control flow in the program runs in such a fashion, that underlying program runs are not executed synchronously.

In an RTT, every program run has an omittable corresponding control column. The cells of a control column contain either control commands: PLAY (▶), PAUSE (⏸), BACKWARD (⏮). The PLAY-command enforces that the corresponding program run is normally executed—meaning that the underlying program is executed on the previous state and new output values are computed. For conve-

nience reasons, the play-command is the default if the value (or the complete column) is omitted. It can only supersede by a PAUSE-command. This default also makes every GTT a valid RTT. The PAUSE-command effects the opposite of the PLAY-command. If PAUSE is given, the corresponding program is not executed, and the state and output values of the previous cycle remain the same. The input values may change because they are controlled by the challenger (environment). PAUSE enforces stuttering of the underlying program.

The BACKWARD-command allows a resetting of the underlying program state to a previously seen state associated with a table row. If BACKWARD ($\blacktriangleleft_r$) with row number r occurs, then the state of the corresponding program run is set to the state that was present when r-th table row was entered. The reset occurs before the program is executed and only affects the state and output variables; not the input variables. As the current and future input values may diverge from the previously observed inputs, the future program states may also diverge, and the previously observed behavior may not be repeated. For example, BACKWARD to 0 ($\blacktriangleleft_0$) resets the program to its initial state, but afterwards different input can be chosen. Thus, the state and output variables differ from the original execution start.

We have identified these control commands as useful for our specification needs of evolution and security scenarios. But the set of control commands are not fixed and can be extended. For example, a fast-forward command, which lets the program executes n times in a cycle, could be helpful in cases where a program run takes multiple cycles for the computation, which is done in one cycle in the other in other program runs.

With control commands, we break up the synchronous development of the underlying program runs, in which not every program run calculates a new state in every turn. This allows relating program runs, which are not perfectly synchronized to each other, by compensating the deviation using control commands. As we see later, we convert the specification in the control columns into a specification of input variables of the augmented programs in Section 10.2.1. Thus, the underlying game remains synchronous; in each game round, the challenger and all programs need to emit an input or output, respectively.

10.1.2.1 Projection Functions in Column Headers.

In Section 4.2.2.2 we already introduce projection functions in the columns. Originally, these projection functions were introduced along with RTTs in [Wei+20] and later with this thesis back-ported to GTTs. This allows us to keep this section rather short.

A column is dedicated to a program variable, e. g., $p\text{»}X$, or a projection function. The dedication are noted in the column header. In GTTS, both dedications describe a function that projects the program state to a value X. New in RTTS is that we have multiple program runs. To address this, we already give fully-qualified variable in column headers. For projection function, we need to catch up. A projection function in RTTS is a function f that maps the history (up-to and including the current situation) of n program runs to an m-tuple:

$$f\colon (\bar{\sigma}_1, \ldots, \bar{\sigma}_n) \mapsto (v_1, \ldots, v_m) \ , \tag{10.1}$$

where $\bar{\sigma}_i \in (\mathcal{I} \times \mathcal{S})^*$ $(1 \leq i \leq n)$. Note that $\mathcal{S}$ contains also the output values like in Definition 5.10. The program variables are still a special case of a projection functions—they define the following projection function:

$$p\text{»}X \quad := \quad f(\bar{\sigma}_1, \ldots, \bar{\sigma}_n) = \sigma_p(X) \ .$$

The variable σ_p denotes the last state in the given history of the pth program run.

In GTTS, we allow the use of predicates in cells, which are then applied to the return value of the projection function (Section 4.2.2.2). The typical comparison predicates, like equality "=" or less-than-equals $\leq$ are built-in. Further predicates are definable by the user. The only requirement is that the arity of the predicate matches the size of the returned tuple. Together we obtain a nice feature: the relational column—a column that expresses the relation between variables. For an example, refer to **Press** column in the Figure 10.2.

Formally, let f be a projection between n program runs to an m-tuple of the column, where a predicate P with arity m is given in the cell without arguments. Then, this abbreviation is expanded to:

$$\begin{aligned} &\textbf{let } (v_1, \ldots, v_m) = f_c(\sigma_1, \ldots, \sigma_n) \textbf{ in} \\ &\quad P(v_1, \ldots, v_m) \ . \end{aligned}$$

Scalar values are silently lifted.

10.1.3 Example

We illustrate our extensions with an example in Figure 10.2. The given RTT expresses a change during the evolution of a stamping system for imprinting workpieces.

Stamping in the new software version of the system should have the same behavior as in the old version. But the new version is capable of error handling during the imprinting routine. If a workpiece is inserted into the stamp (input

#	CONTROL		ASSUME			ASSERT		⏲
	old	*new*	**WP**	*old*»WP	*new*»*Release*	**Press**	*new*»*State*	
1	▶	▶	=	FALSE	—	≥	», `Free`	—
2	▶	▶	=	TRUE	—	≤	»,`Stamping`	1
3	▶	▶	=	FALSE	—	≤	»	—
4	⏸	▶	=	—	—	>	`Error`	—
5	⏸	▶	=	—	TRUE	>	—	1

Figure 10.2: Example for an RTT

variable `WP` is true), the stamp is pressed against the workpiece (expressed by stamp pressure in the output `Press`), and the stamp signals when it is ready (`State`). The new revision is extended by a diagnostic sensor, which recognizes a failed imprint. If such an error is indicated, the stamp needs to be inspected and cleaned by an operator, who afterward releases the system from the error state.

Figure 10.2 shows an RTT capturing this behavior which is based on two program runs of the *old* and one of the *new* software revision. The column header "**WP**" represents a function which maps the `WP` variable of both program runs to a 2-tuple: $(\sigma_{old}(WP), \sigma_{new}(WP))$; analogue for column header "**Press**". The normal behavior in the new system version is (only) described in relation to the old version (Rows 1–3), where the WP variables in both runs need to equal, and the new stamping pressure (new»$Press$) needs to be lower than the old pressure (old»$Press$) when the stamp should *not* imprint, and vice versa when the stamp should imprint. In addition, the table describes the error handling behavior without referring to the old version (Rows 4 and 5).

In Row 1, the table states that both systems should signal a free stamp (variable `State`) until a workpiece is inserted. When a workpiece is present (Row 2), the stamping process starts (Š) for an unspecified amount of time (Row 3). Now, we enforce that the new stamp pressure is at least as high as in the old version. Thus, the new system imprints, when the old system would imprint, but possible with a higher pressure. Up until (and including) Row 3 we expect that both programs behave relative to each other in every step. When an error occurs during imprinting (Row 4), the equality between the old»$State$ and new»$State$ in the previous row would fail, as the old revision is not aware of errors and keeps pressing. We are proceeding with Row 4, where such equality is not required anymore. The old revision is paused (⏸ is given in the control column) until released by the operator indicated by new»$Release$ = TRUE (Row 5). The row group consisting of Rows 4 and 5 makes the error handling optional for cases without an imprinting error. During the error handling, we force, that the new stamp pressure is lower than the old stamp pressure during

⟨*table*⟩ ::= `relational table` ⟨*name*⟩ '`(`' ⟨*runs*⟩ '`)`' '`{`' ⟨*body*⟩ '`}`'

⟨*runs*⟩ ::= ⟨*name*⟩ ('`,`' ⟨*name*⟩)*

⟨*signature*⟩ ::= `var` ⟨*modifier*⟩* '`{`' ⟨*name*⟩ ('`,`' ⟨*name*⟩)* '`}`' ⟨*name*⟩ '`:`' ⟨*datatype*⟩
| `var` ⟨*modifier*⟩* ⟨*fqvar*⟩ [`as` ⟨*name*⟩] '`:`' ⟨*datatype*⟩
| `gvar` ...
| `column` ...
| `inherit_from` ...

⟨*fqvar*⟩ ::= [⟨*name*⟩] '`::`' [⟨*name*⟩]

⟨*row*⟩ ::= `row` [⟨*name*⟩] [⟨*time*⟩] '{' ⟨*control*⟩* ((⟨*fqvar*⟩|⟨*name*⟩) '`:`' ⟨*cell*⟩ ['`;`'])* '`}`'

⟨*control*⟩ ::= (`play` | `pause` | `back` '`(`' ⟨*name*⟩ '`)`') '`:`' runs ['`;`']

Figure 10.3: Additions and changes to the grammar Figure 6.8 for specifying RTTs

imprinting.

The complete specification is repeated infinitely often. Not all variables from the interface of both reactive systems have an own column. Some omitted variables are specified by indirectly, i. e., old»$State$ via the column of new»$State$.

10.1.4 Extending the Input Language

In Section 6.3.2, we define our input language for the textual presentation of GTTs as the input format of our verification pipeline. The new RTT features need also to be captured grammatically. In Figure 10.3, we give the new grammar as the difference to the grammar for GTTs in Figure 6.8. Meaning that the redefined non-terminals, i. e., ⟨*table*⟩, ⟨*signature*⟩ and ⟨*row*⟩, replaces their previously definitions. The Listing 10.4 shows the textual representation of the RTT in Figure 10.2.

The RTT-features inside the body are enabled with the keyword `relational`. Then, after the table name, we specify the amount and names of the program runs (⟨*runs*⟩). The next difference is the definition of the signature. The first change addresses the declaration of program variables which are referenced (and declared) with a program run and variable name (⟨*fqvar*⟩). Second, we have a new construct for combinatorial definition which allows us to define a program variable for multiple program runs at once. The other constructs of the

signature, for global variables, columns, and inheritance, remain unchanged.

The scheme for the addressing of variables also takes place in ⟨*row*⟩. Additionally, a new non-terminal ⟨*control*⟩ captures the specification of the control column. A control command is given on a row by the corresponding keyword (`play`, `pause`, or `back`) and a list of programs. The `back` keyword also takes the name of a row to which the program run should be reset to. When multiple BACKWARD-commands are given for the same program run, we consider this a mistake and the specification is not well-defined. If a program run is not explicitly paused, the underlying program of this run is executed for the particular row.

10.2 Decision Procedure

We define the semantics of RTT, i. e., what it means for a system to conform to an RTT, by reduction to the notion of conformance defined for GTT Sections 4.4 and 5.3. For this, we build the product program [BCK11] from augmented programs. The augmentation simulates the intended effects of the control commands.

We start with the construction of the decision procedure in section and afterward in Section 10.3 we discuss the semantics as related to the two-party game.

10.2.1 Program Augmentation and Product Program

In this section, we describe how we translate a given program P with input and outputs variables to a new program P', which encapsulate the behavior of P and also supports our control commands. Finally, we combine several augmented reactive programs into a single reactive product program, and resolve references to program runs to their variables in the product program.

In the remaining section, we assume $InVar_j$ and $StateVar$ are the sets of the input and state (incl. output) variables of the reactive program P_j to be augmented. Also, for the augmentation, we need to know the table rows which occur in a BACKWARD-command for the particular program P_j. We denote this set of *chapter marks* as $marks_j$.

We introduce additional input variables $stutter$, set_r, and $reset_r$ for every $r \in marks_j$ into augmented program P'_j. Also, we introduce for every table row $r \in marks_j$ a fresh copy of all states $StateVar$. In the following, we assume that these names do not clash with program variables in P_j. The new input

```
relational table rttexample( old, new ) {
  var {old, new} WP : BOOL
  var {old, new} Press : INT
  var new::Release : BOOL
  var new::State : ENUM

  column WP as old::WP, new::WP
  column Press as old::Press, new::Press

  group omega {
    row A [0,-] { WP: =; old::WP: false; new::Release: -;
                  Press: >=; new::State: ::,Free }
    row B [1,1] { WP: =; old::WP: true; new::Release: -;
                  Press: <=; new::State: ::,Stamping }
    row C [0,-] { WP: =; old::WP: false; new::Release: -;
                  Press: <=; new::State: :: }
    group [0,1] {
      row D [0,-] { pause: old;
                    WP: =; old::WP: -; new::Release: -;
                    Press: >; new::State: Error }
      row E [1,1] { pause: old;
                    WP: =; old::WP: -; new::Release: true;
                    Press: >; new::State: - }
    }
  }
}
```

Listing 10.4: The textual representation of the RTT in Figure 10.2.

$InVar'$ and $OutVar'$ are defined as

$$\begin{aligned} InVar'_j :=& InVar_j \cup \{stutter\} \cup \{set_r \mid r \in marks_j\} \\ & \cup \{reset_r \mid r \in marks_j\} \end{aligned} \tag{10.2}$$

$$StateVar' := StateVar_j \cup \bigcup_{r \in marks_j} StateVar_j^r \ , \tag{10.3}$$

where $StateVar_j^r = \{v^r \mid v \in StateVar\}$ is a set of fresh state variables for each chapter mark r.

Figure 10.5 shows the schema of an augmented program. In the core of every augmented program, P'_j is the invocation of the original program P_j. Around this invocation, we add augmentations, which are controlled by the newly introduce input variables.

If the input variable set_r is true, then the current state of the program variables $StateVar$ is stored into the designated copies v_r ($v \in StateVar \wedge r \in marks_j$). The input variable $reset_r$ triggers the reverse operation, and restore a previous stored state form the v_r variables. With the input variable $stutter$ we can avoid the execution of P_j, and the original program state $StateVar$ remains untouched.

If there are variables in $StateVar$ with complex or composed data structured, like arrays or records, we need to copy their complete state recursively. For programming languages with references or pointers, special care is needed to maintain them. In general, to apply the code augmentation, we need to be able to express a valid code fragment, which makes complete copies of the program state.

Translation of the Control Commands. We translate the control commands to input constraints of the augmented programs. The PLAY-command enforces $stutter = false$, otherwise on PAUSE-command $stutter$ needs to be true.

The input variables set_r and $reset_r$ correspond to BACKWARD-command. For every table row, which has a BACKWARD-command on the kth program run to the table row r, we translate this command into the input constraint $k\text{»}reset_r = \text{TRUE}$. On every other table row s, we add $k\text{»}reset_s = \text{FALSE}$ to the input constraints.

On the other hand, if a table row r is in the set of chapter marks $marks_j$ (it is a targeted by a BACKWARD-command), then we add the input constraint $k\text{»}set_r = \text{TRUE}$ in this table row for its first iteration. This may require an unfolding of the table row, in which the table row is split into two rows with equal input and output constraints. The first copy of the row receives the duration constraint "$[1, 1]$", and in the second row, the original duration constraint decreased by 1

Input: $InVar'_j$ in (10.2)
Data: $StateVar'_j$ in (10.3)
if set_r **then** // added for each $r \in marks_j$
  $v^r := v$ /* for all $v^r \in StateVar^r_j$ */ ;
end
if $reset_r$ **then** // added for each $r \in marks_j$
  $v := v^r$ /* for all $v^r \in StateVar^r_j$ */ ;
end
if $\neg stutter$ **then**
  invoke P_j;
else
  skip;
end

Figure 10.5: Scheme of an augmented program P'_j where P_j is the original program. set_r, $reset_r$ and $stutter$ are new input variables, and v^r new state variables.

accordingly. If the original table was skippable (duration constraint was ≥ 0), then both cloned rows are part of an optional row group. We also need to k»$set_r = false$ on all other tables rows s with $s \neq r$.

Construction of the Product Program. The next step is to weave the augmented programs $P'_1, \ldots, P'_n$ into a product program $P_\otimes$ (cf. [BCK11]):

$$P_\otimes(\bar{i}_1, \ldots, \bar{i}_n) := P_1(\bar{i}_1); \ldots; P_n(\bar{i}_n)) \ . \tag{10.4}$$

The product program is a sequential composition of the augmented program P'_j ($1 \leq j \leq n$). Note that the construction requires a renaming of input and state variables to avoid name clashes, which ensures the effects of the program runs are isolated from each other. Therefore,

Proposition 10.1. *Let $P_\otimes$ be a product program of $P_1, \ldots, P_n$, then*

$$\mathcal{B}(P_\otimes) = \mathcal{B}(P'_1) \times \ldots \times \mathcal{B}(P'_n)$$

The possible programs runs of the product program $P_\otimes$ corresponds to the Cartesian products of the program runs P'_j. This proposition makes the difference to our notion of a monitored program in Definition 6.7, in which the tester can abort the run of the product program globally.

In the last step, we need to rewrite our relational variable reference $p\text{»}X$ to refer to the corresponding variable in the product program, accordingly to a required variable renaming.

10.3 Conformance of RTTs

We define the RTT conformance for sequence of reactive systems (cf. Definition 5.12): x

Definition 10.2 (Relational Conformance)**.** *A sequence of reactive systems P_j ($1 \leq j \leq n$)* strictly conforms *to a RTT T if and only if the product program P' strictly conforms to the GTT T', i. e.,, it is a winning strategy in the two-party. Analogously, the sequence* weakly conforms *to T if and only if $P_\otimes$ weakly conforms to T' (its strategy never loses for T'), and it* cooperatively conforms *to T, if and only $P_\otimes$ cooperatively conforms to GTT T' (its strategy always reaches the end of every instantiation of T).*

The GTT T' is derived from the RTT T and matches the transformation done during program augmentation. In particular, the control commands are translated into input constraints, and the variable references are rewritten to point to the correct variable in the product program.

Recapitulate the game; in the relational settings, we have two parties: the challenger and the combination of systems (program runs). And it is not an $n + 1$-party game, where one party is the challenger and the other n parties are the program runs. Thus, when the combination of the systems loses, we do not differentiate which system is faulty in general. For particular scenarios, we would state a defect attribution. Consider the verification of evolution scenarios, if the system combination does not conform, we attribute the defect to the new revision, as we consider the old revision as given. For information flow verification (in which we use two program runs of the same program), it is clear which program is faulty, but unclear which program run.

10.4 Application Scenarios

In this section, we show the applicability of RTTs using scenarios of the Pick-and-Place Unit (PPU) community demonstrator (Section 7.3). The PPU is an automated production system built up by industrial hardware for researching the co-evolution of production systems. Thus, there are multiple evolution scenarios of this plant. For our following application scenarios, we selected representative scenarios, and sub-components of the demonstrator, e. g., the logic for controlling single hardware units.

Table 10.6: Runtime of the verification for our application scenarios given as a median of five samples

Scenario	Wall time [secs]	Size Model [bit]	Size Programs [LoC]
Regression and Delta Verification (Section 10.4.1)	5.86	321	406
Restart after an emergency stop (Section 10.4.2)	0.79	519	570
Exchange of Subsystem (Section 10.4.3)	1.79	767	835
Information Flow (w/ violation) (Section 10.4.4)	25.75	373	1758
(w/o violation)	114.62	373	1758

Statistics. All run times (wall clock) given below are a median of five samples where conformance to the RTT has been verified on an Intel Core i7-8565U, 16 GB RAM, with the model checker nuXmv 1.1.1 [Cav+14] with IC3 [BM07] for invariant checking. The stated lines of code do not include empty lines or comments. All verification artifacts are available in the companion material [Wei21] or on the companion website of the original publication.[2] The Table 10.6 summarizes the runtime statistics of the verification. Discussion of the verification is given in the sub-sections of each scenario.

Implementation. We have implemented the RTTs in our VERIFAPS verification framework for Programmable Logic Controller software. Figure 10.7 shows the pipeline, which takes for every program run the correspondence source code P_1 to P_n and the file with the textual representation (Section 10.1.4) of the RTT. The augmentation, as described in Section 10.2.1, is applied separately to the given program runs and requires additional information from the given test table. Also, the symbolical execution is applied on each program run separately; the product program is later constructed inside the model checker. For the symbolical execution, we remove complex program structures, for example, we unwind bounded loop, embed procedure calls, or unfold arrays and record data types as described in Section 6.3.1. The result is a logical model in SMV (Symbolic Model Verifier) for each program. The RTTs are encoded in a domain-specific language, and also translated into an SMV model by using the same techniques as for GTTs, i. e., the automata construction in Section 6.1. Theoretically, the given verification task is decidable: the model checker either verifies the compliance or returns a counterexample. Technically, due to the restriction of space and time, the verification is terminated without a result.

[2] https://formal.iti.kit.edu/formalise2020

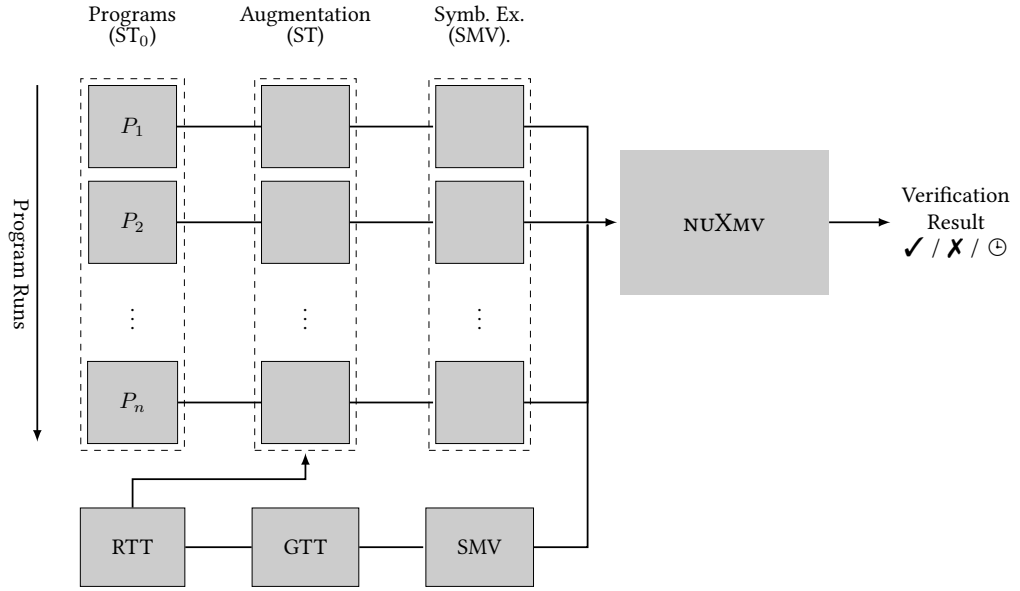

Figure 10.7: Verification pipeline for RTTs and ST_0 code

We have implemented this pipeline only for the model checker pipeline, but the principles can easily be adapted to the C-verifier pipeline. The main difference is that the monitored program consists of the product program, which also is the sequential combination of the programs $P_1, \ldots, P_n$, and the tester program.

10.4.1 Regression and Delta Verification

Scenario. The origin of this scenario is in Section 7.3.2, where we prove a weaker form. We only verify the newly introduce behavior functionally. In this scenario, we demonstrate a combination of regression verification and delta verification [Ule+16a] for two software revisions. While regression verification proves the equivalence for the common part of the system behaviors, delta verification ensures functional correctness for the differences. This scenario is based on the evolution step from the third to the fifth software revision of the PPU, which introduces an optimization for workpiece throughput: The new software revision makes use of the waiting time while a piece is stamped to deliver a new workpiece from the magazine to the conveyor belt. The old revision waits for the stamp to finishing the imprint.

#	CONTROL *old*	CONTROL *new*	ASSUME Level	ASSUME WPReady	ASSUME Position	ASSUME Carry	ASSUME _state	ASSERT Turn	ASSERT Lower	ASSERT Vacuum	◷
1			»	»	»	»	—	»	»	»	—p
2	⏸		Up	MetalReady	TRUE		Crane_Go_Up				1
3	⏸		—	—	—	—		Right			≥ 0
4	⏸				Magazine						1
5	⏸					FALSE		Stop	TRUE	On	≥ 0
6	⏸					TRUE			TRUE		1
7	⏸		Down			—			FALSE		≥ 0
8	⏸		Up								1
9	⏸		—					Left			≥ 0
10	⏸				Conveyor						1
11	⏸							Stop	TRUE		≥ 0
12	⏸										1
13	⏸									Off	1
14	⏸		Down						FALSE		≥ 0
15	⏸		Up								1
16	⏸		—					Left			≥ 0
17	⏸				Stamp						1
18	⏸							Stop			1
19		⏸									—p
20			»	»	»	»	»	»	»	»	1
21			»	»	»	»	—	»	»	»	—p

Figure 10.8: Combination of regression and delta verification. For presentation reason, we omitted program run reference in the column header. All columns belonging to the new program run, therefore "»" refers to the same variable in the other programs and states the equivalence.

Table. The RTT in Figure 10.8 contains rows for the following input and output variables: (a) `Level` indicating the position of the crane (`Up`, `Down`, `Unknown`), (b) `WPReady` signaling whether a (non-)metal work piece is ready at the magazine, (c) `Position` of the crane (`Magazine`, `Stamp`, etc.), (d) `Carry` signaling whether a work piece has been picked up, (e) the current `_state` of the internal state machine, (f) `Turn` determining the move direction of the crane (`Stop`, `Left`, `Right`), (g) the desired position of the suction cup (`Lower`), and (h) whether the suction cup should hold a work piece. The table specifies that the old (third) revision and the new (fifth) revision behave equally (Row 0, Row 19, Row 20), except for the phase in which the optimization occurs. During the optimization phase, the program run of the old revision pauses (Row 1 to 17) while the new program run moves the Crane to the Magazine, picks up the workpiece, delivers it to the Conveyor Belt, and moves the Crane back to the Stamp. This sequence is described as a functional specification. In Row 18, we pause the new program run and let the old program run until both runs are synchronized again on the same internal state $\_state$. This is required because the waiting duration for imprinting is hard-coded and cannot be changed.

Verification. We proved system conformance to the RTT for the function block for controlling Crane hardware. The old Crane software has 327 lines of code (LoC), the new version has 406 lines. Both function blocks have 15 input

and 6 output variables. Verification of weak conformance took 5.86 seconds in NUXMV. The state size of the model is 321 bits (software and table). For the verification, we decreased the waiting durations of the timers.

10.4.2 Restart After an Emergency Stop

Scenario. A crucial point of PLC software is their handling of emergency stops and also the recovery from it. The state may be only partially reset and obsolete variable values have persisted. in the reset of the variables. Because emergency stops are a rare event, such mistakes are hard to locate with testing, especially, as emergency stop needs to be handled in any program state.

In this scenario, we verify that a reset of function block Crane (implemented in SFC) does not lead to new behavior. To achieve this, we compare two program runs of this function block. The first function block is reset by the given input variable, the second run is restored to the initial state by a BACKWARD-command.

SFCs have vendor-specific support for resetting the current active steps (automata states). This reset is triggered by setting the variable *SFCReset* of an SFC instance to true. The reset does not affect the state variables. Hence, the engineers need to take care of the state manually.

Table. Figure 10.9 shows the RTT of this scenario, which is based on two runs a, b of the same program (function block). The column header I represents a function that projects the states σ_a, σ_b of the program runs a and b to their input variables (excluding $SFCReset$). Hence, the equality in Row 1 and Row 2 represents the equality of the 15 input variables. Analog, for the column **O** for all nine output variables.

This RTT states, that the outputs of both runs are always equivalent if their inputs (except $SFCReset$) are equivalent. If the program run b is reset by the flag $SFCReset$, we reset the complete state in a to its initial value. By assuring the equalities on the in- and outputs we prove that no new behavior occurs by the manually reset.

Note that this scenario is similar to proving the absence of an information flow. In this case, we forbid a flow $SFCReset$ into the observable outputs of the system.

Verification. We took the function block for controlling the Crane, which is originally programmed in SFC, from Scenario 13 of the PPU. The Structure Text for this function block has 570 LoC, and 16 input and 9 output variables. The verification of the 519 bit-sized model took 0.79 seconds in NUXMV.

10.4.3 Exchange of Subsystems

Scenario. The first scenario is regression verification between two programs. In this scenario, we verify an equivalence between three programs.

The developer exchanged the subroutine for the handling of the emergency stop. In the original version of the PPU, the program stops the movement of critical actuators by setting constant to the output variable, but jammed workpieces are hard to be removed by the operator when the actuator standstill. The new emergency stop handling allows a manual override of certain output variables to allow the operator to free the jammed workpieces. We prove that the old program version of the programs behaves like the new version in normal operation, and during an emergency stop the new program behaves in some output variables like the separated emergency subsystem. Additionally, every time the system enters the emergency behavior, the emergency subsystem needs to be reset, and hence the emergency stop should always react as it would be the first time it is triggered.

Table. The Figure 10.10 shows the specification for this scenario. The RTT is stated over three program runs, where n is the new program, o the old program, and e the separated emergency subsystem (that was integrated into n). The old and new programs receive equal inputs values and should return equal output values in normal operation ($EStop = \text{FALSE}$)[3]. During an emergency operation ($EStop = \text{TRUE}$), the new system should behave like a fresh isolated separated emergency system. This also requires that the emergency subsystem is correctly reset and accessed inside the new program.

The function $In(o, n)$ denotes a function which projects the three states of the three program runs to the input variables from the old system and the input variables of the new system. The result is a pair of two lists, and each list containing the input values in the lexicographical order of each program run.

[3]The PPU has multiple buttons to trigger an emergency stop. For the presentation, we decided to combine those into a single variable.

#	CTRL		ASSUME		ASSERT	⏲
	a	b	**I**	b»$SFCReset$	**O**	
1	▶	▶	=	a»$SFCReset$	=	≥ 1
2	⏮$_0$	▶	=	TRUE	=	1

Figure 10.9: This RTT specifies that resetting with $SFCReset$ results into the same behavior, as running the system from its initial state. The program runs a and b are from the same program.

#	CONTROL			ASSUME			ASSERT		⏲	
	o	n	e	n»$EStop$	$In(o,n)$	$In(n,e)$	$Out(o,n)$	$Out(n,e)$		
1				o», FALSE	=	—	=	—	≥ 1	
2			⏮$_0$	o», TRUE	=	=	—	=	1	∞
3				o», TRUE	=	=	—	=	≥ 0	

Figure 10.10: Regression verification between three programs: The new program (n) should behave like the old program (o), except during an emergency ($EStop = \text{TRUE}$), then it should be like separated emergency system (e).

Analogue, $In(n,e)$ for the input variables of the new and emergency program, and $Out(o,n)$, $Out(n,e)$ for the output variables. Thus, equality in the column $In(o,n)$, states that all input variables of the program run o and n should be equal.

Verification. We verified this on the Scenario 5 using the complete control software of the PPU. This includes the control for the Crane and also of the Magazine, and the Stamp. The original version of the PPU has 775 LoC and the implanted emergency system has 50 LoC. The verification takes 1.79 seconds in NuXmv. The model size is 767 bits.

10.4.4 Information Flow

Scenario and Table. In this scenario, we verify that there is no information flow from the configuration parameter for the suction pressure (`Pressure`) to the crane movement (`Turn`). Again, we use projections in the column header: $In_L(a,b)$ and $In_H(a,b)$ to hide the large amount of variables. The function $In_L(a,b)$ maps the 23 variables of both runs a and b together, which does are allowed to influence the `Turn` variable. In contrast, $In_H(a,b)$ maps the variables together, which contain the information which should non-interfere with the output, i. e., the secret information. In our case it is just the suction pressure:

$$In_H(a,b) := \quad f(\sigma_a, \sigma_b) = (\sigma_a(Pressure), \sigma_b(Pressure))$$

More precisely, the table in Figure 10.11 describes that the non-interference is only required after the initialization of the system (Row 2, `Init`). In all rows, we enforce that all input variables besides `Pressure` are equal in both runs. Row 3 expresses the non-interference property: For any two runs with arbitrary values for `Pressure`, the output `Turn` is the same. Therefore, `Turn` is only determined by the other input variables.

Unfortunately, a monitor function block stops the crane if the suction pressure is outside the expected range, i.e., if $Pressure \notin [1, 9999]$. Therefore, the

#	**ASSUME**		**ASSERT**		⏲
	$In_L(a,b)$	$In_H(a,b)$	$a\text{»}Turn$	$a\text{»}Init$	
1	$=$	$\neq$	—	`FALSE`	≥ 0
2	$=$	$\neq$	—	`TRUE`	1
3	$=$	$\neq$	$b\text{»}Turn$	—	$-_\infty$

Figure 10.11: Information flow property: the input variable $Pressure$ (given $In_H(a,b)$) should not have an influence on the output variable $Turn$. The control columns are omitted, as no intervention is required.

software does not conform to the RTT in Figure 10.11. This unintended outcome can be fixed by limit the $Pressure$ values on both program runs to be in the range $[1, 9999]$.

Verification. We used the complete fifth revision of the PPU, including the function blocks for all components. The product program has 1758 lines of code with 266 variables. For the interfering version, the model checker needs 25.75 seconds to find a counter-example. Proving conformance w.r.t. the fixed (non-interfering) specification takes 114.62 seconds. The size of the state space is 373 bits.

10.5 Conclusion

In this section, we present RTTs, an extension of the GTT for a powerful and comprehensible specification of relational and k-safety properties for reactive systems. Besides pure relational constraints, RTT allows the use of functional constraints, for example, to enforce certain behaviors. RTTs come with the introduction of the main feature: the control column.

Through the control column, we enable that the engineer can specify the relation between programs, that are now equal in a step-by-step fashion. Adding new commands to the control column requires only a new code-transformation. A code-transformation is often easier to implement and to understand (especially for an engineer or software developer), than a change in the automaton construction or the underlying properties in the verifier. Also, the idea of code augmentation is not exclusive for RTTs and should apply to other specification languages. By using product programs for the verification, we can build upon the semantics of the GTT without any changes to it.

We show the applicability and feasibility by verifying three change and an information flow scenario on isolated software components and the com-

plete control software of the PPU, an existing demonstrator for co-evolution in production systems.

Discussion. RTTs, as an extension of GTTs, have the same weaknesses and strengths as GTTs (Section 9.1). Additionally, they are often more complex due to the orchestration of the different programs. For simple relational specification, this overhead is not justified, and we often fall back to can simpler or customize languages. For example in Chapter 12, we consider an invariant for the specification of the regression verification which consists of three parts: a relation of input variables, a relation of output variables, and a condition under which these relations holds in all states (Section 12.1). Of course, such a specification is also expressible in an RTT with a single row, but then we are not exploiting the strength of RTTs. Like GTTs, the RTTs are well-suited when the specification changes over the execution time of the system. In particular, the niche for RTTs are the specifications where the relation between the program runs is not constant (in contrast to Section 12.1), and also we break the synchronized execution of the underlying partially. We conclude with RTTs we specify complex sequences of relations between programs.

Moreover, the specialized specification languages for the relational verification are rare, and often exists only for particular cases, like [SS14] for information flow in Java, or end up as an extension to a specification by adding the possibility to talk about other programs (`\call` for ASCL in [Bla+17]). Prior to this work, there did not exist a relational specification language for reactive systems.

The features of RTTs are not fixed. For example, one obvious improvement is to get rid of the rigidity of the control commands to enable specifications like "The system should be reset ($\blacktriangleleft_0$) if the previous output was zero". In such cases, the control commands depend on the program variable values. And as the control commands are just input variables, this is expressible in the derived GTTs, but not in the original RTT itself. During the verification of the application scenarios, we felt the need for better tool support, especially, a counter-example visualization which makes a failed proof-attempt of a relational property more understandable. For GTT, GETETA provides such counter-example preparation. For RTT, we need to deal with the different programs and program runs. Moreover, the visualization should support playing with the stuttering to find the correct alignment of the program runs.

Chapter 11

Provably Forgetting of Information

After we introduce RTTs, we use them to specify a novel property: forgetting of information in a reactive system. During the manufacturing process, confidential information is generated and aggregated that constitute business secrets; therefore they are the focus of attackers and require rigid protection. On the other hand, if we can prove, those business secrets are absented in a system, the effort for the protection for this system could be invested in different information, aspects, or systems.

11.1 Confidentiality in Automated Production Systems

In the era of the industrial revolution (IR4.0), information security becomes an increasingly important aspect of industrial manufacturing systems. As these systems should be more configurable and adaptable, the amount of software within these systems increases. Moreover, the manufacturing system and the enterprise resource planning system (ERP) need to share more information, e.g. the manufacturing system needs to announce finished workpieces, and the ERP configures the manufacturing system according to the customer's wishes of the next job. The information becomes a valuable target, either for violating the confidentiality or integrity of the manufacturing process.

Business Secrets. The configuration and processing information of the manufacturing system can contain very sensitive and crucial information about the

manufacturing process or the economic situation. For companies, the leakage of these secrets is a crucial threat to their business model, reputation, revenue, and therefore their existence. Also, the gathering of information is often a preparation for an attack on the integrity of software. For example, this is presumed indentation behind malware "Havex" campaign in 2014, which purposeful collected information of manufacturing systems [KR18, p. 115f.].

At least in Germany, these business secrets are protected by law if and only if the companies protect their secrets by using state-of-the-art methods themselves (cf. [Mül18] and § 2 Nr. 1 lit. b GeschGehG). Therefore, a company is interested to know in which components their data is stored to apply adequate protection measurements purposefully. The proposed approach helps to prove that specified information given by input values of the reactive system is eventually forgotten after a specified period.

Attacker Model. PLC can be a profitable target for an attacker. A PLC gathers and aggregates the sensor values, controls the actuator, and holds the configuration settings.

In this section, we want to protect the confidentiality of such information against an attacker. We assume the attacker is able to observe the current state of the running software, for example, by capturing maintenance access. We want to show that the knowledge gain for the attacker is limited. In detail, the attacker does not learn anything about the specified secrets before its successful intrusion.

This scenario is similar to forbidding an information flow from the incoming secret-classified information into the internal state of a system. But this property is too restrictive for manufacturing systems: The system needs to react to events and these events are just recognized by sensor values to handle the current situation. With our approach, we allow only a flow between of the secret information into the internal state, if the secrets are supplied in the last k cycles.

Introductory Example: Baffle Gate. We want to motivate our approach on a simple example given in Listing 11.1. Imagine a baffle gate used at metro stations or airports. In its default state, a baffle gate is blocked to prevent people from passing it. It becomes unblocked after a successful authorization for a defined period. After this span, the baffle gate blocks passage again. In our example, we consider the number of passed people (and therefore the number of successful authorization) as the secret of our system. This secret should not be revealed to an attacker, who can observe the state of the baffle gate.

The Listing 11.1 shows the controller software of the gate given as Structured Text code. The software has one input signal (`authorized`), which signals

```
1  FUNCTION_BLOCK BaffleGate
2
3    VAR_OUTPUT blocked : BOOL; END_VAR
4    VAR_INPUT authorized : BOOL; END_VAR
5    VAR CONSTANT T : UINT := 10 END_VAR
6    VAR wait : UINT; END_VAR
7
8    wait := max(wait - 1, 0);
9    IF authorized THEN
10         blocked := False;
11         wait := T;
12   END_IF
13
14   IF wait = 0 THEN
15       blocked := True;
16   END_IF
17 END_FUNCTION_BLOCK
```

Listing 11.1: Simple program to control a baffle gate. The baffle get is unblocked for T cycles after an user has authorized.

a successful authorization, and one output signal (`block`), which determines whether a passage is possible. The body of the function block is executed periodically.

With a classical secure information flow notion, we would forbid an information flow from the incoming (secret) `authorized` signal to the internal state of this program. This is far too restrictive as this input signal releases the gate, which affects the internal state and also output signal `blocked`. The system does not store the secret. We need to come up with a more relaxed notion of secure information flow. We notice, that after a certain amount of time (in our example $T = 10$ cycles), the system forgets about the previous `authorized` signal. Leading to the idea, that after the system has observed secret information, the system needs time to forget this information. One major motivation behind this idea is the observation, that the current sensor and actuator values are important for the software to understand the current situation in the manufacturing system and therefore to react properly. But the older information becomes, the more unimportant it is for the software and thus it should not have an influence anymore.

Contribution. In this section, we present a novel notion of *forgetting information* for the software reactive systems, which we formally specify using RTTs (Chapter 10). By using RTTs, we obtain an automatic verification pipeline for our property, which is based on state-of-the-art model checking. Informal, *forgetting information* means, that the software of the reactive system forgets a piece of specified information after n execution cycles. Therefore, the knowledge gain of an attacker by observing the state of the software is limited. We focus our approach on the software part of a reactive system. Therefore, we exclude the environment, e.g. the sensors and actuators of a manufacturing system, and other software components of the manufacturing system, like SCADA or ERP systems.

We demonstrate the application of the verification pipeline on the demonstrator system, which is a manufacturing system, originally developed by a third-party contractor for the demonstration of replay-attacks.

11.2 Related Work

Information flow on programming languages is a well studied area [Mur15; SM03]. In the core of secure information analysis, is non-interference of the secret to the public observable output. Dimitrova et al. [Dim+12] apply information flow properties on reactive systems. They invented an extension to linear temporal logic (LTL) to SecLTL, which include a new operator $\mathcal{H}_{H,O}\phi$ for specifying that the output O does not depend on the initial values of the secret variables H before the condition ϕ is satisfied.

Our quantification consideration (Section 11.3.4) are different from established quantified information flow [Smi09]. The later one tries to quantify the information leakage in bits, in contrast, we quantify the number of cycles needed to forget information and the number of different variables.

In the field of reactive systems, this paper seems to be the first work on ensuring that information is erased in and from the system. Simeonovski et al. [Sim+15] propose a framework for managing the "Right to Be Forgotten" introduced by the European Court of Justice in search engines. Erasing of secrets is an important topic in cryptography, as the cryptographic keys are highly sensible and valuable information. The presented work of [Cre+99] assumes that the attacker can access the hardware of the system. As a protection, the authors introduce a cryptographic primitive, called "erasable memory", which makes it possible to implement the essential cryptographic action of forgetting a secret. Diesburg et al. [Die+16] present "TrueErasure" a building block for secure deletion of sensitive information. The building block must be triggered by the software. In contrast, we verify that the information is erased from the system

state, defined by the software. Our state is a logical model, and in practice, the information might be still present in the memory or caches.

Outline. Section 11.3 consists of the explanation and formalization of the information forgetting property. We also show how this property can be verified. In Section 11.4 we explain our experiment, with an introduction of the software components, architecture, and information flow (Section 11.4.1). Also, we present the steps that were taken to obtain a verifiable program (Section 11.4.2), e. g., the removal of floating-point variables. Finally, in Section 11.5, we discuss our approach and its application on manufacturing systems.

11.3 Forgetting of Information

In this section, we build our approach for proving that a given software for a reactive system forgets a piece of specified information. As our approach expresses a kind of non-interference between variables, we are re-using existing secure information flow notions.

Notions of Non-Interference. An information flow exists if the information of a program variable h influences a different variable l. We also say, that variable h interferes with variable l. For considerations of confidentiality, the variable h (for *high*) contains the secret information and l (for *low*) the observable information by the attacker.

In our baffle gate example, the variable `authorized` would be *high* and the complete internal state is *low* (also including the secret `authorized` variable). Therefore, an attacker can learn directly the secret information. As motivated before, a classical non-interference notion that forbids a flow from high to low variables is too restrictive in our case.

For the presentation in this chapter, we assume that the complete state of the PLC is observable by an attacker, and also the secret is given in the form of input (sensor) values. After the formalization, it should be easy to lift these constraints and allow a more flexible notion, e. g., with a declassification of particular state variables or an arbitrary low-equivalence predicate [MR07].

11.3.1 Formalization Idea

We need to find a relaxed information flow property, that allows that a secret can be stored for a short time inside the state, and is eventually be forgotten later.

Let us make a thought experiment with two instances of our baffle gate. First, we run both systems for an arbitrary amount of time and different input signals, resulting in a different number of authorized and passed peoples. In this phase, the secret information is injected into both systems. Second, we synchronize the sensor inputs of both systems for a short amount of time, i. e., k cycles. Third, we stop both systems and compare their internal states. If the states are indistinguishable, then the number of positive authorizations is not derivable anymore.

In contrast to information flow, we introduce an *annealing phase*. During the annealing phase, the secret information needs to be superseded in the system. A system, which passes this experiment supersedes the specified secret information in at most k cycles. Thus, it can only leak secret information observed in the last k cycles. Finally, the knowledge gain of an attacker, observing a single state of the PLC, is limited to the secret information of the past k cycles. For manufacturing systems, the cycle times are rather small ($\leq 10ms$) and therefore the time window of the derivable information is rather short. The formalization is given in form of an RTT in Fig. 11.2.

11.3.2 Formalized Property

We formalize our thought experiment. Firstly, the experiment talks about two program runs of the same software, which makes our property 2-safety *hyperproperty* [CS10]. Secondly, our property is temporal, as time passes between the injection of the secret and comparing the states.

Proof Obligation for Information Forgetting. Figure 11.2 shows the RTT that captures our thought experiments. We define projections functions, denoted as $V^{\otimes}$, which maps a set of variables V into two tuples representing the values of the variables in V in the first and second program run. Formal expressed as

$$V^{\otimes} := \qquad f(s, s') = (\pi_V(s)\ ,\ \pi_{V'}(s')) \tag{11.1}$$

where $\pi_V(s) := (s(v_1), \ldots, s(v_n))$ (with $n = |V|$) denotes the projections of the program state $s \in \mathcal{S}$ to a tuple of the values of variables in V. We use $V^{\otimes}$ to project the low or high variables of the states into a pair of comparable values.

To capture our previous assumption, that the secret information can be observed via sensor values and the attacker can observe the complete state, we define $StateVar$ to be set of variable names containing the local state variables, analog $InVar_L$ for the low input, and $InVar_H$ for the high input variables. Thus, the projection $StateVar^{\otimes}$ maps the two program states to a tuple, where the first element matches the local state of the first program run, and the second

#	ASSUME			ASSERT	⏲
	$StateVar^{\otimes}$	$InVar_L^{\otimes}$	$InVar_H^{\otimes}$	$StateVar^{\otimes}$	
1	$=$	$=$	$-$	$-$	1
2	$-$	$=$	$-$	$-$	$-$
3	$-$	$=$	$=$	$-$	k
4	$-$	$=$	$=$	$=$	ω

Figure 11.2: Template of RTT for information forgetting with an annealing phase of length k

element for the second program run. The same is valid for $InVar_L^{\otimes}$ for the low input and $InVar_H^{\otimes}$ for the high output variables.

The RTT in Fig. 11.2 is defined for two program runs of the same program, e. g., the program of the baffle gate in Listing 11.1. Row 1 expresses that the local states and low inputs of both program runs need to be equal. The predicate "$=$" expresses equality and enforces that the first and second element of the tuple are equal. Our secret inputs $InVar_H$ can differ between both runs. The predicate "$-$" (don't-care) does not enforce any constraint. After the first execution, we allow that both internal states can differ, caused by the different values for secret inputs. The low inputs I_L remain equivalent. Row 2 is similar to Row 1, in contrast, we do not assume that the states are equal. We can apply the Row 2 arbitrary often, allowing us to reach all (reachable) pairs of states. Row 2 can also be skipped. The choice of whether we stay in Row 2 or go forward with Row 3 happens nondeterministically. Row 3 represents the annealing phase—enforcing that the secrets $InVar_H$ are equivalent between both program runs. After k cycle, the states of both program runs need to be equivalent (Row 4)—indicating that the secret previously injected is forgotten. Row 4 is applied infinitely often.

Lift the Assumption. Our assumptions can easily be lifted. First, we assumed that the secret is injected via the input variables, but the secret could also be in the initial configuration of the system. Then we have to exclude these state variables from the equality of the states in the column $StateVar^{\otimes}$. Second, the assumption that the complete state variable is available for the attacker is expressed as the equality on the state variables in Row 4. The equality relation represents the capabilities of the attackers, i. e., when two states are distinguishable for them. We can use different relations to express the different capabilities of the attacker, e. g., we can model that only the last two bits of a variable are accessible for the attacker.

Our formalization also allows us to encode a when-declassification [MR07], allowing an information leakage after certain events has occurred. To achieve

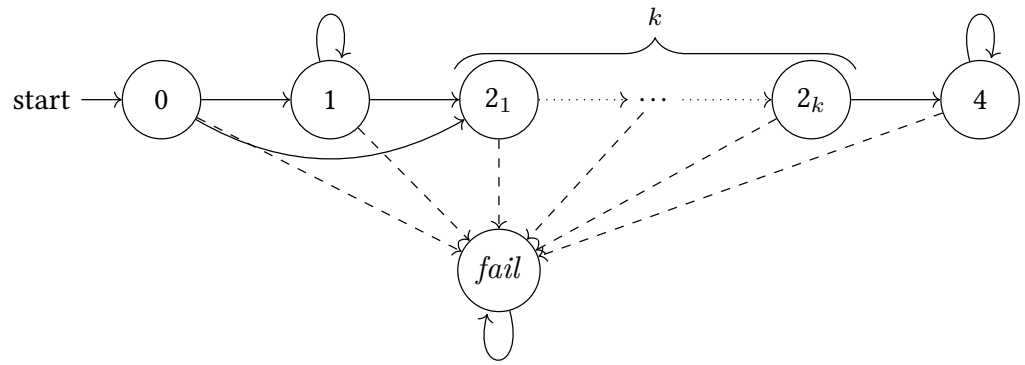

Figure 11.3: Automaton checking the RTT in Figure 11.2. The states correspond to the table row. A solid line represent that the assertion and assumption of the row are adhered. The dashed lines are triggered if only the assertion is violated.

this, we need to encode the negation of the events in the assumption side of the table to exclude these runs (or plays) from our verification.

11.3.3 Verification

In general, the verification of RTTs is described in Chapter 10. Here, we describe briefly how the verification works for our given RTT in Figure 11.2.

The verification need the program, and the RTT instantiated with the variable sets for the states variables ($StateVar$), low and high input variables ($InVar_L$, $InVar_H$). Both programs and the RTT are translated into the model-checker format SMV (Figure 10.7). For this, we simplify the given program, for example, we unwind the bounded loops and unfold composite data structures (records), and via symbolic execution, we gain the single-static assignment form which we encode into SMV (Section 6.3.1).

The translation of the RTT into SMV is done via the construction of a nondeterministic automaton. The states of the automaton (Figure 11.3) encodes the table row of the RTT (in comparison to Figure 6.3). If an automaton state is active, the corresponding row is active. The outgoing edges are triggered, when the constraints of this row hold: If assumption and assertion constraints hold, the test proceeds with the successor rows, nondeterministically. If an assumption is violated, the run is discarded. And if only the assertion is violated, the error state $fail$ is activated.

The universal quantification over the traces of the two program runs is handle by construction *self-composition* [BDR04] in the model-checker.

A system provably forgets the specified information (conforming to the

RTT) if and only if for every input sequence, there exists an infinite run in the automaton, that does not end in the error state $fail$. This property can be encoded as an invariant and verified efficiently (Section 11.4).

11.3.4 Quantification of Security

Our presented formalism is a quantification of security, because we quantify *how fast* (in the number of cycles) information is forgotten and *how much* (in the number of variables) information is forgotten. Both numbers are on an ordinal scale—hence we use them to compare the security of systems. A system which forgets *more* information *faster* is more secure.

The computed numbers can be considered from the view of *risk assessment*. A risk is formed by two factors: the entry probability and the costs in the event of damage or loss. Our approach does not prevent that an attacker can successfully capture information of a PLC system. But if a successful attack occurs, the attacker sees a limited and known amount of information. Therefore, if a system forgets more information faster, it has a lower risk, because of the reduced costs, whereby the entry probability keeps the same. On the other hand, we do not have an interval scale, as it is invalid to state, that a system is two times more secure than another system if it forgets the same information two times faster. For the cost assessment, it is crucial which information is kept in the system.

11.4 Experiment

In this section, we show the application of our property on a real-world example, which was developed by an industrial third-party contractor in charge of the Fraunhofer IOSB, and designed to demonstrate replay attacks in industrial communication networks [Pfr+16].

Program to be Verified. Our subject for this experiment is a demonstrator of an aPS that main controls a color wheel (Figure 11.4). The demonstrator consists of a PLC, an HMI interface, several network components, and a motor rotating a color wheel. The PLC software controls the rotation of the color wheel, either immediately by inputs from the HMI, or automatically by an operator-configured sequences.

We select the number of wheel turns as the business secret. Thus, our goal is to verify that the number of turns is not stored within the state of the software.

The PLC supports two modes: automatic and manual operation. The mode is selected by an integrated HMI. In the automatic mode, the PLC executes a user-defined a sequence of steps. A step consists of a target position (angle),

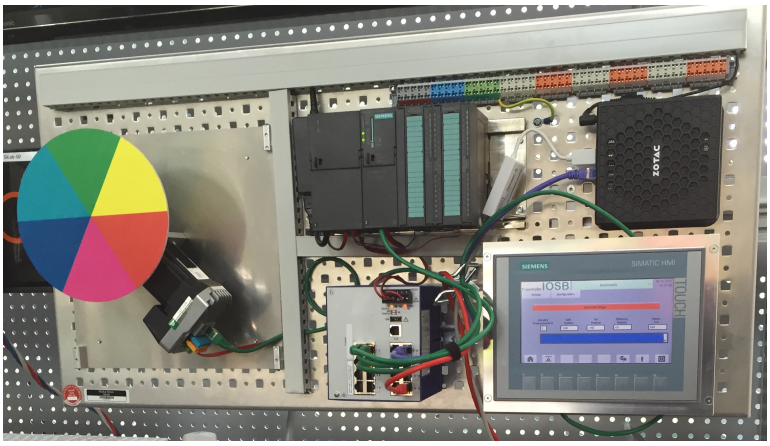

Figure 11.4: Hardware components of the system to be verified. Image provided by Fraunhofer IOSB

velocity, acceleration, deceleration, and waiting time. The PLC drives the wheel to the target position with the defined parameters. If the position is reached, it waits for the defined waiting time and then proceeds with the next step. Depending on the configuration, the system leaves the automatic mode after the sequence is completely executed or restarts with the first step. The automatic mode can be paused or aborted. In the manual mode, the users can interact with the system more directly via the HMI. The user can stop and spin the wheel in both directions with a user-defined, or predefined velocity. Also, the manual mode allows setting the reference position of the wheel.

Overview. In the remainder of this section, we give an overview of the structure of the software (Section 11.4.1). We identify the verified fragment and needed preparation steps (program transformations, Section 11.4.2). We close this section with the verification results.

11.4.1 Software Architecture

The software is implemented in Structured Text and consists out of 16 user-defined data types, two function blocks, two functions (initialization and communication with HMI) and the main program[1].

The Fig. 11.5 visualizes the internal architecture and the program flow. The main program is executed cycle-wise every n ms. The components com-

[1] Additionally, there are seven auxiliary functions, mostly for converting to and from external sensor values.

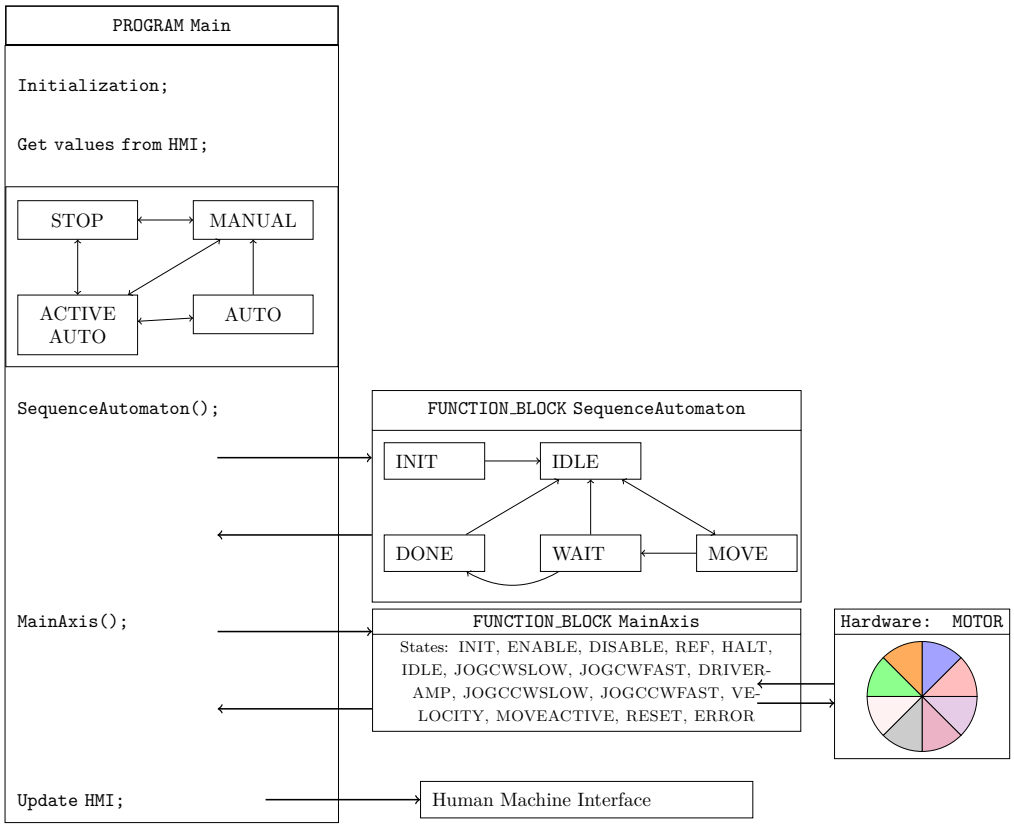

Figure 11.5: Architecture of the software consisting out of four structural elements: main program, sequence automaton, main axis control and HMI.

municate via variables in the global state. For example, the function block `Sequence Automaton` sets the mode of `MainAxis` directly and `MainAxis` sets the values for the HMI. `Initialization()`. This function ensures a correct initialization of the global state. Mainly it ensures that the String variables holding the error messages are properly set and all arrays are filled. The initialization function is executed once, i. e., in the first cycle. In the second step, the current values from the HMI are transferred to the global state. Third, the main program determines the operation mode, either STOP, MANUAL or AUTOMATIC. [2] The fourth step invokes the function block `Sequence-Automaton`, which only handles the automatic mode. This automaton decides whether the motor needs to move, the target position is reached, or the waiting time is elapsed and the next step should be executed. These decisions are based on the sequence of user-defined

[2]The automatic mode is split into a mode for pre-selection of the auto settings (AUTO) and executing the automatic mode (ACTIVE AUTO).

entries within the global state. A distinct internal variable describes the current state of the sequence execution (cf. Fig. 11.5). The call to `MainAxis` triggers the most important part of the software: the motor control. There are 15 modes defined in this function block. The mode variable is set internally or externally by the main program or the sequence automaton. A mode determines the calls of the driver function blocks with specified parameters.[3] For example: if the mode is HALT, the driver parameters are set to stop the motor, and at the end of this function block the driver is called. Erroneous and success calls are handled by `MainAxis` by jumping to the IDLE or ERROR mode. The program code of `MainAxis` before the verification preparation is given in [Wei19, Appendix B].

The program sizes are: `Initialization` has 54 LoC, Program `Main` 97 LoC (reading from HMI 40 LoC, operation mode 45 LoC), Function Block `MainAxis` has 362 LoC, Function Block `SequenceAutomaton` has 65 LoC, and writing to HMI has 81 LoC.

11.4.2 Preparations for Verification

For the verification, we concentrate on the function block `MainAxis`. But before the verification, we need to apply program transformations to bring this function block into a supported shape for the symbolic execution and the model checker. In the remaining chapter, we do not distinguish between state and output variables of the function block, and consider the output variables as part of the state.

The starting point is the original implementation of the function block `MainAxis`, the global state, and the auxiliary functions. We start by simplifying the function block into ST_0 (Definition 6.9). Secondly, we need to apply simplifications customized for the given software. In detail these are:

- We remove assignments to `dScratch` and `VSObj_McFaultDescription`. The first location is a global variable that is never read but is written. The second one holds a String value of the current error cause in the HMI, and unsupported by the model checker.

- The model checker cannot handle floating values. Therefore, we transform variables of type REAL to INT. Additionally, we need to remove the conversion functions `REAL_TO_INT` and `INT_TO_REAL` with the identity function. We apply the same for the used (and not needed anymore) rounding of values (“`(x/1000)*1000`”).

[3] The driver function blocks are an extension of PLCOpen Motor Control and not defined in this project.

- In the last transformation, we slice the program to remove all variables, that are neither read nor written, and remark the remaining variables as input and output according to their reading and write access.

The resulting program for the verification code is 421 LoC.

11.4.3 Result

The complete transformation pipeline is implemented in our verification library for VERIFAPS. After the translations and simplifications, the state space in the model checker is 566 bits large (270 bits input, 296 bits state). For the specification, we have $|StateVar| = 32$ $|InVar_L| = 51$, and $|InVar_H| = 1$. The complete variables sets $InVar_L$, $InVar_H$ and $StateVar$ are given in the appendix of [Wei19]. We instantiated our property (Fig. 11.2) with $k = 2$ for the annealing phase. For the verification, we used NUXMV 1.1.1 [Cav+14] on an Intel® Core™ i5-6500 (3.20GHz) with 16 GB RAM. For efficiency, we start both systems in arbitrary equal states, instead of the initial states. This is an over-approximation and reduces the diameter of the search space.

The system does not adhere to our property (Fig. 11.2). NUXMV finds a counterexample in 1.85 sec (median, $n = 3$). So there exists a run that does not lead to an erasure of the secret information.

Fixing the Leak. Inspecting the counterexample reveals how the different supplied velocities, given via `ActStep.rVelocity` variable, are result into different values in the state variable `Move-Axis1.-Velo-city` after $k = 2$ cycles of equal input. Further, we can prove that all the other variables do not infer with secret anymore, by using the same formalization but exclude `MoveAxis1.-Velo-city` from a set of state variables $StateVar$. Hence, this variable is the only leakage.

We propose, a fix for the leak, in which we overwrite the variable at the end of the cycle, after the execution of `MoveAxis1`. This step changes the behavior of the program and afterward a correct behavior has to be validated. The fix is easy: We added two assignments at the end of the code (cf. [Wei19, Appendix B]), which overrides the input parameters of an underlying driver function block:

```
1 MoveAxis1.Velocity := 0; MoveAxis1.Execute := FALSE;
```

The first assignment overrides the velocity, s. t. the variable does not leak this information. The second assignment disables the command execution of the

Table 11.6: Runtime of the model-checker for proving or finding a counter-examples of the information forgetting for various annealing phases k and scenarios (A) original leaky version, (B) original leaky version proving all other variables do not leak, and (C) fixed (non-leaky) version.

$k =$	2	3	5	7	10
(A)	3.39 sec	2.95 sec	2 min 52 sec	9 min 24 sec	2 h 50 min
(B)	57.40 sec	45.82 sec	3 min 32 sec	10 min 29 sec	2 h 36 min
(C)	40.93 sec	29.74 sec	3 min 5 sec	10 min 46 sec	1 h 33 min

instance `MoveAxis1` of the Function Block `MC_Move-Relative` [TC211] in the next cycle. The driver function blocks are called at the end of the Function Block `MoveAxis` for sending commands to the motor controller of the color wheel. Setting `Execute` prevents that the controller that the velocity of 0 is sent to the controller in the next cycle. If a new velocity needs to be set, the `MoveAxis1` re-enables the execution and also sets the velocity.

Model Checking Runtime. The Table 11.6 gives an overview about the runtime: for finding the counterexample in the original leaky program (A), proving that only `MainAxis1.-Velocity` leaks (B), and proving the fixed version (C). Sample size is $n = 2$. The standard derivations were rather small (less than 20 seconds), so we omit them. We include the run times for a set of different annealing-phase lengths k to show the performance impact by selecting different lengths. The different lengths are redundant as our notion is monotone: If a system forgets the information in k cycles, then it also forgets the same information in $k' > k$ cycles. The parameter k highly influences the depth search space, as it determines the number of unwinding the system definitions before a counterexample can be found to strengthen the to be constructed inductive invariants in IC3.

11.5 Discussion

Our approach has limitations and pitfalls that are given from the design decision. In this section, we discuss and illustrate these limitations w.r.t. our experiment.

Limitation of verifying software. Our approach is focused on the software of the reactive system. Especially in the experiment, our focus is more narrowed, as we consider only the single Function Block `MoveAxis` as it is the most complex and critical software part inside this software project, and deals finally with

the sensors and actuators. Hence, every control-command to control the motor passes this piece of code.

It was out of our scope whether other parts of the cyber-physical system adheres to the property. This includes the other PLC software parts (human-machine-interface (HMI), Function Blocks of the motor driver, etc.), the underlying operation system or the hardware, and even the physical environment. All of these parts are not (completely) accessible from the PLC software, but might be observable by an attacker.

For example in the demonstrator, the attacker would gain access to the complete user-defined program sequence, containing the information of the current the segment, its position, velocity, etc. From these program sequence, an attacker might guess an estimation of the previous amount of turns, but also an estimation of the future amount of turns.

Moreover, information may be stored inside the physical plant itself (in the actuators) and are fed back to the PLC via sensors. Without a suitable environment model, information flow in the physical plants is not traceable.

Nonetheless, it is possible to take the internal PLC and the cyber-physical aspects into consideration for verification, if precise models for them would be available.

Verification on the PLC level. In our experiment, we prove the privacy on the second lowest level (PLC level) of the automation pyramid. Below is *field or electronic level*, containing the sensors and actuators, and on the upper levels are the SCADA system, the manufacturing enterprise system (MES), and the enterprise resource planning system. The upper levels are gathering information from the lower levels, and thus they store the business information which we tried to forget on the PLC level.

Nonetheless, verification of forgetting information PLC is needed and beneficial. Due to their real-time requirements, the protection of PLC against attacks is hard to achieve without threatening the functionality and safety aspects. The upper levels are built with standard, more powerful, PC components that can be protected with standard equipment and mitigation strategy. On the lower side, the benefits of our approach require that the sensors and actuators are not infiltrated.

Single Observable State. We limit the leakage in our attacker model to one PLC software state. In practice attacks expand over several days to months, while an attacker may see every state of the system. Our approach keeps still useful: the attackers can not guess information, which is lying past their infiltration. But in practice, the attackers would assume that the infiltration was not discovered,

and so attacked industrial system runs in the same way (program, configuration, workpieces, etc.) as before its infiltration. Given them a possibility to guess the (forgotten) historical forgotten information, but with uncertainty.

Program Transformation. For the verification we apply some program transformation, i. e., demoting floating-point variable to integer variables, removing string-, and unused variables. These transformations can be critical to the property as they alter the control and information flow. For example, code lines could become unreachable using integers instead of floating-point arithmetic. In contrast, symbolic execution and other simplification, like structure unfolding, are uncritical as they are not changing the semantic of the program, special the set of reachable states remain the same. In general, these program transformations need a justification individually for every verification attempt. For example, in our experiment the floating-point arithmetic just occurred for translation between the values of the HMI and the inputs for the motor driver.

11.6 Conclusion

In this chapter, we present a notion and formalization to describe the forgetting of information in a reactive system. This notion is a relaxed variant of an information flow property in which we give a system a period (annealing phase) to forget the flowed secret information. The instantiation of the notion requires the length of the annealing phase, and the specification of high and low information in the system. A software that dependently forgets the specified information, e.g. business secrets, is *not* protected against successful intrusion, but in case of an intrusion, the number of leaked secrets are reduced. As our approach focuses on the software of the system, thus in practice further investigations are needed to cover the deployment context of the software, especially, cyber-physical aspects of the physical environment. With an experiment, we show the feasibility of our notion on a real demonstrator system, implemented by an industrial contractor.

Additionally, with this property, we demonstrate the power of RTTS: the focus on the sequential shape, and the overtime changing relation between variables.

Chapter 12

Modular Regression Verification

Reactive software driving technical systems is often in operation for long periods of time, sometimes for many years or even decades. Guarantees regarding its correctness must be ensured over the entire system lifetime, and the software must go along and maintain quality through all hardware and software evolution steps. Testing might help to identify software flaws and to increase confidence in the correctness of the system. But can only cover a small amount of the possible scenarios.

A solution to this problem is *regression verification*: Instead of using two separate specifications for two revisions, the two revisions are compared directly to each other, where the old revision serves as a functional specification for the new one. In Section 10.4, we have already investigated how to specify (and verify) complex evolution scenarios with RTTs, in which the new revision should behave similarly to the old revision (and other systems). In this chapter, we go a step back and use simpler specification model, called regression verification contracts, which are one-rows RTTs (Section 12.1).

The idea for regression verification of aPS software goes back to [Bec+15], which shows that the resulting proof obligations (for regression verification) can be discharged in some cases, but that the size of the system may make the verification approach suffer from the potential problem of state space explosion. Even for our rather simple case study of the Pick-and-Place (PPU, Section 7.3), which has less complexity than real-world scenarios, proving equivalence with the approach described above took up to a day of computing time.

As a response to this challenge, this chapter presents a technique to modularize regression verification by decomposing the verification condition into

smaller subgoals which can be regression-verified individually. The novelty in comparison to existing model checking modularization approaches is not that individual programs are decomposed into manageable fragments, but that the programs are split into pairwise blocks combined to be verified relationally.

The modular verification approach is embedded into a new regression verification algorithm which combines different lightweight (syntactical) and more heavyweight regression verification analyzes.

Modularizing the verification has multiple gains: Firstly, it reduces the state space of proof obligations, allowing them to be more feasible for model checking. Moreover, it introduces a locality principle: Parts of a program not touched at all by a refactoring can be factored out and equivalence be proven by simpler, syntactical techniques. For modules that occur more often, the verification effort can also be reduced since they only need to be analyzed once.

Contribution and Overview. In this chapter, we present a sound modularization technique in Section 12.2 for the regression verification of reactive system software; it requires that relational specifications of subroutines are given (by the user). In Section 12.3, we present a new algorithm for regression verification which orchestrates a collection of diverse heavy- and lightweight verification techniques making the new modular analysis more powerful in practice. We implemented the algorithm for PLC software, and demonstrate the feasibility of our approach on the PPU. The evaluation and the results are discussed in Section 12.4.

12.1 Formal Equivalence Relations

We briefly repeat the regression verification notions from [Bec+15], as these notions form the base of our modularization approach.

When we consider regression verification formally, we need to set two program behaviors into relation. The first notion that comes to mind is *perfect equivalence*, which requires that the behaviors of two PLCs programs P and Q are identical, i.e., that they produce the same output when presented with the same input trace. Formally, we can reuse our definition of the trace semantics (Definition 5.10), and receives two functions $\mathrm{B}(P)$ and $\mathrm{B}(Q)$, are equal:

$$\text{for all } \bar{i}, \bar{i}' \in I^* : \; i = i' \implies \mathrm{B}(P)(\bar{i}) = \mathrm{B}(Q)(\bar{i}') \;\; . \tag{12.1}$$

However, they may very well differ on the chain of memory states reached in their traces, i.e., P and Q need not be identical. Perfect equivalence is a very strict notion for evolution scenarios, as it does not allow any behavioral

difference between the old and new revision. Still, it is useful to prove that a *software refactoring* maintains the system behavior.

In many evolution cases, behavioral differences must be taken into consideration to capture intended changes, like bug fixes or performance optimizations. The differences can be handled with the more flexible notions of *conditional and relational equivalence*. They extend perfect equivalence in two ways: Firstly, conditional equivalence allows us to filter scenarios that should not be included in the equivalence analysis using a predicate τ on the input values. Secondly, in relational equivalence one can replace the equalities in (12.1) by different relations that express the equivalence between input ($\approx_{in}$) and output ($\approx_{out}$) values:

$$\text{for all } \bar{i}, \bar{i}' \in I^* : \ \tau(\bar{i}, \bar{i}') \wedge \bar{i} \approx_{in} \bar{i}' \implies \mathcal{B}(P)(\bar{i}) \approx_{out} \mathcal{B}(Q)(\bar{i}') \ . \quad (12.2)$$

The triple $C = (\tau, \approx_{in}, \approx_{out})$ that parameterizes (12.2) is called a *semantical regression verification contract* for P and Q. Perfect equivalence EQ is a special case of a regression verification contract with $EQ = (true, =, =)$. This generalizes the ideas of design-by-contract [Mey92] for single program properties to multi-program analyses. The condition (12.2), which we denote as $RV(C, P, Q)$, defines when the contract C is satisfied by the programs P and Q. Note that this triple can also be presented as an RTT with one strong-repeated row, where τ and $\approx_{in}$ are on the assumption side, and $\approx_{out}$ is on the assertion side.

12.2 Modularization

Modularization is a technique to split up the program code into individual separate modules with defined interfaces. The effects of a module are limited to a specific scope, allowing a separate analysis. Wherever one module calls another module, the effects of the call can be abstracted rather than to include the full module implementation. Thus, the complexity introduced by the control flow and internal state of the submodule is invisible in the caller module.

We present a decomposition rule which allows us to exploit the modularization of reactive software to break down the regression proof obligation $RV(C, P, Q)$ into simpler proof obligations.

12.2.1 Motivational Example

Consider the plant in Figure 12.1 representing an assembly line with a conveyor belt B and two processing stations s_1 and s_2 (e.g., a drill and a stamp). A detector d at the beginning of the conveyor belt recognizes the arrival of a workpiece W. Once a workpiece has arrived, the automatic process starts, and W is moved

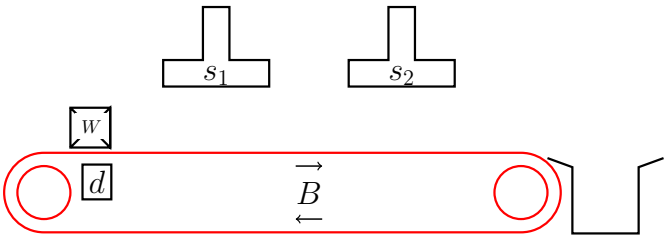

Figure 12.1: Schematic of the plant consisting of a conveyor belt B with two processing stations s_1 and s_2.

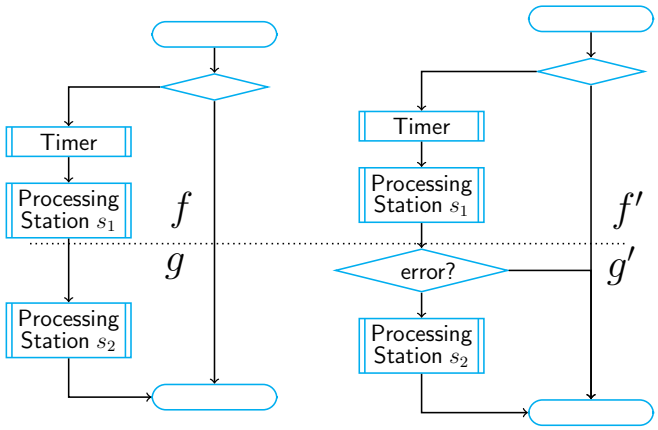

Figure 12.2: Sketch of the program flow of the motivating example: the original revision on the left and the adapted revision on the right.

from left to right, passing both processing stations, and eventually falling into the basket at the end of the belt.

In the original software revision, every workpiece is unconditionally processed by both processing stations. While a piece is being processed, the conveyor belt halts for a defined amount of time. Let us assume that experience has shown that the process at s_1 may occasionally fail. The software has hereupon been adapted, and, after the revision, the plant can recognize workpieces for which s_1 has failed. If a faulty workpiece leaves s_1, the second processing station should be skipped and the piece should be sent to the output basket directly.

Software Structure. Figure 12.2 shows a sketch of the program flow of the main program for both revisions. The difference is that a branching statement has been introduced after s_1. The modules "Timer" and the code for the processing stations remain unchanged.

Regression Verification and Modularization. Obviously, both software revisions behave differently when a faulty workpiece occurs. To apply regression verification, a regression verification contract is required that specifies when both revisions should behave equally. In this example, the two revisions behave equally if no faulty workpiece occurs. The contract for this example would therefore encode in the filter predicate that no faulty workpiece is ever detected.

The non-modular approach for regression verification in [Bec+15] does not exploit the fact that the subroutines for controlling the hardware components remain unchanged. The full code of both programs is encoded for the translation against the regression contract with a model checker. The evaluation [Bec+15] (revisited in Section 12.4) shows that some evolution scenarios cannot be solved in a reasonable amount of time.

With the approach that we introduce here, we are able to replace the implementation of the modules in the encoding by their contracts, and can hence lower the verification effort by this abstraction which can thus become an enabler for the regression verification for larger programs.

This abstraction does not come for free. For a successful abstraction, sufficiently strong contracts that imply the necessary properties must be found. Finding them automatically may be as difficult as the whole program analysis itself. In the presented approach the user has to come up with suitable contracts.

12.2.2 Formalisation

The goal of this section is to look at composed programs and to introduce an inference rule that allows one to modularize regression verification proofs for such programs. Let therefore the two programs P, Q be implemented as a composition of two subprograms, say $P = f; g$ and $Q = f'; g'$. We have introduced programs as functions and the semicolon operator is the forward composition of functions (i.e. $(f; g)(x) = g(f(x))$).

For the modular analysis, it must be possible to identify the similar subprograms in P and Q that then become the corresponding parts between the two revisions. In the example from Section 12.2.1, for instance, the two programs can be split into two subprograms along the dotted line.

If one pair of corresponding subprograms can be verified in isolation (in this example g and g') for a contract C_g, this result can be used for the verification of the relation of the remainder programs where g and g' can be abstracted by (uninterpreted) placeholder function symbols x and x' which stand in for the programs g and g'. As a precondition in this proof obligation, we may assume the regression verification contract C_g for x, x' without knowing the exact functionality of g and g'.

The inference rule for the verification of $RV(C,\ f; g,\ f'; g')$ for a regression verification contract C has two premises which encode (1) that C_g is a valid regression verification contract for g and g' and (2) that the two programs satisfy contract C under the modular assumption that g and g' satisfy C_g.

$$\frac{RV(C_g, g, g') \qquad \forall x, x'.\ RV(C_g, x, x') \rightarrow RV(C,\ f; x,\ f'; x')}{RV(C,\ f; g,\ f'; g')} \tag{12.3}$$

12.2.3 Modularization for Conditional and Relational Equivalence

In this section, we present how the modularization rule (12.3), formulated over functions, can be concretely used for the regression verification of reactive programs. We start with the definition of a very general concept of a reactive programming language with frame structures, then introduce the decomposition rule, and close this section with remarks on properties of the rule.

Programs. We consider simple loop-free programs, containing assignment- and if-statements. Additionally, we introduce a `frame`-construct for marking program parts which should be modularized. Programs are constructed by the grammar

$$\begin{aligned} \langle Prg \rangle \rightarrow\ & \langle name \rangle := \langle expr \rangle \mid \langle Prg \rangle\ ;\ \langle Prg \rangle \\ & \mid \texttt{if}\ (\langle expr \rangle)\ \{\ \langle Prg \rangle\ \} \mid \texttt{frame}(\langle name \rangle)\ \{\ \langle Prg \rangle\ \} \end{aligned} \tag{12.4}$$

in which the $\langle name \rangle$ denotes identifiers and $\langle expr \rangle$ side-effect-free expressions.

The set for programs produced by Prg is rather abstract and limited. However, it is expressive enough to encode reactive programs without (unbounded) loops. Programs in the low-level language (12.4) can, e. g., be constructed from more complex program languages like Structured Text or C by unwinding (bounded) loops and arrays, unfolding record data types and inlining procedure calls. In particular, we consider programs that are similar to ST_0 (Definition 6.9).

Frames and the Scope of Variables. Frames structure the otherwise unstructured programs into modules. During the translation from input programs into the low-level language (12.4), structuring elements from the source language, like function-blocks or method invocations, are translated into frames. Frames can also be manually added by a user—to enable the handling of complex code refactorings which took place across the boundaries of the structural elements in the source code, e. g., when a computation from inside a method is pulled out to the method caller.

For a sound abstraction and modularization, the scopes of variables must be restricted, and the `frame` constructs mark these scopes. With every frame identifier N we associate three disjoint sets of variables: input (in_N), state ($state_N$) and output (out_N) variables. Every variable v occurring inside a frame named N must belong to one of them. The variables in these categories are constrained as follows: Input variables are only read within the frame but may be written from outside the frame. For state variables read and write access inside the frame is allowed, but any access outside the frame is forbidden. Output variables are write-only within the frame, and read-only outside the frame. Global variables do not fit into this scheme but can be encoded into it by an automatic program transformation. The program transformation introduces a new input and output variable for each global variable, which occurs in the frame. The global variable is assigned to the input variable at the beginning of the frame. The effect of the frame on a global variable is captured in the output variable, which is assigned to the global variable after the frame. Therefore, such variable categorization can always be established.

In a modularization step, frames will be replaced by an abstraction using their contracts. The variables play an important role then: They manifest the interface at which the frame is abstracted for modular treatment. The input variables must adhere to a precondition on the entry of the frame, the state variables can be removed from the program when the frame is abstracted, and the output variables assume values which adhere to a postcondition for the frame.

It is important to note that frame identifiers can occur on several frames within the same program. This models the case that multiple operations are invoked on the same module within a program. This happens, e.g., if the same function-block is invoked twice in an IEC-61131 context, or if a (stateful) procedure is called multiple times from the original program.

To make the abstraction of frames sound, we forbid a partial replacement. A replacement is partial when not all frames with the same identifier vanish after the abstraction. The simple case is to replace all sub-frames with the identifier at once. But there cases requiring more care: Consider the situation in Listing 12.3. If we abstract Frame B, we also remove only one of two C frames, hence a single C frame keeps in Frame A. This abstraction is unsound because the remaining C frame operates on a completely different state space. Note that the regression contract contracts do not describe anything behavior on the state variables. In our example, the variable `o` is always even after the second C frame in the original, but after the abstraction the then-case of the if-statement becomes reachable. Moreover, it is still allowed to abstract Frame A completely.

Frames that modify the same variables must have the same identifier, and all frames with the same identifier must have the same code and the same variable

```
1  frame(B) {
2    frame(C) { s := s + 1; o := s;}
3  }
4  frame(C) { s := s + 1; o := s;}
5  if(o  2 != 0)  <unreachable>;
```

Listing 12.3: A frame constellation in which it is forbidden to abstract Frame B as this makes `<unreachable>` reachable.

signature. This is not a restriction: If different functionalities access the same variables (e.g., different methods of an object in an object-oriented setting), programs can be refactored such that all frames contain the same integrated code that implements all functionalities. An additional parameter together with a case a distinction is used to decide the concrete functionality in each frame.

Specification and Verification. For both modular functional and modular regression verification, one needs contracts for the abstraction. In Section 12.1 we have already encountered the concept of regression verification contracts on the semantic level. We will refine this notion now to program entities. Let two loop-free programs P and Q be given. A *regression verification contract* is a triple (ϕ, α, ω) of three formulas: the functional precondition ϕ, the relational precondition α and the relational postcondition ω. The semantics of these regression verification contracts are semantical contracts (Section 12.1). The formula ϕ evaluates to the filter predicate τ, and the interpretation of α and ω are the input and output equivalence relations.

The programs P and Q operate on disjoint sets of variables such that their statements programs cannot interfere with each other's state spaces, and are only connected in formulas within contracts. We can therefore use the sequential composition $P\,;Q$ to obtain the effects of their independent executions. The proof obligation which needs to be verified reads, written as a Hoare triple [Hoa69],

$$\{\phi \wedge \alpha\}\ P\,;Q\ \{\omega\}\ . \tag{12.5}$$

In Section 12.3 we will describe efficient techniques to encode such proof obligations for decision procedures.

Modularization Rule. Let in the scenario introduced above, f and g be frame identifiers such that a frame for f occurs in P and a frame for g occurs in Q. For

modular treatment, we need to look at the programs that abstract from the code of inner frames within their enclosing programs (as a parallel to the replacement of x for g in (12.2)):

Definition 12.1 (Factor program)**.** *Let P be a program according to* (12.4) *and f be an identifier. The frames for f in P all have a unique occurrence number i. The factor program P/f is then derived from P by replacing each frame i for identifier f with the following sequence of statements:*

1. $\texttt{in}_i := \texttt{in}$ *for every input variable* in
2. $\texttt{count}_f := \texttt{count}_f + 1$
3. $\texttt{out} := \texttt{out}_i$ *for every output variable* out

The freshly introduced variable $\texttt{count}_f$ for the factored frame f is used to bookkeeping about the number of invocations of f during a run of the program, and is needed to make the upcoming modularization rule sound.

In non-regression program verification, modularized subprograms are often replaced by an obligation to show the precondition of the block and an assumption of the postcondition afterward. Since in regression verification, we deal with two programs at a time, all we can do in the local context is to *remember* the values of all invocations for a global, program-spanning argument to take them into account. The following inference rule does precisely that. Instead of proving (12.5), one can show the two formulas that together imply it: (a) f and g together satisfy a regression verification contract $(\phi^{fg}, \alpha^{fg}, \omega^{fg})$, and (b) the factored programs satisfy the original regression contract. The intermediate variables in_i and out_i introduced by the factor program allow us to specialize a formula and set it into the context of one concrete call-site of the frame identifier. For a formula γ over the variables of P and Q, the instantiated formula $\left[\gamma\right]_{i,j}$ denotes the formula in which all occurring variables from P have been replaced by the counterpart of the i-th invocation and all variables in Q with the variables of the j-th invocation. For perfect equivalence $\varepsilon = (in^f = in^g \rightarrow out^f = out^g)$, the instantiated formula $\left[\varepsilon\right]_{1,2}$ would read $in^f_1 = in^g_2 \rightarrow out^f_1 = out^g_2$.

Definition 12.2 (Modular regression verification)**.** *For two programs P and Q (with disjoint variables) and frame identifiers f and g, let π_f and π_g denote the programs which are inside the corresponding frames f and g and let n (m) be the number of occurrences of f (g) in P (Q). For a regression verification contract $(\phi^{fg}, \alpha^{fg}, \omega^{fg})$ for π_f and π_g the inference rule*

$$\frac{\{\phi^{fg} \wedge \alpha^{fg}\}\ \pi_f\,;\pi_g\ \{\omega^{fg}\} \qquad \{\phi \wedge \alpha \wedge \kappa \wedge \Gamma\}\ P/f\,;Q/g\ \{\omega \wedge \kappa\}}{\{\phi \wedge \alpha\}\ P\,;Q\ \{\omega\}}$$

with $\Gamma = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} \left[\phi^{fg} \wedge \kappa \wedge \alpha^{fg} \rightarrow \omega^{fg}\right]_{i,j}$ *and* $\kappa = (\texttt{count}_f = \texttt{count}_g)$ *is called the modularity rule.*

The assumption Γ of the second premise couples the variables modelling the invocations of f and g. Whenever the input values for invocation occurrences i and j satisfy the precondition $\left[\phi^{fg} \wedge \alpha^{fg}\right]_{i,j}$ of the regression verification contract, the relational postcondition $\left[\omega^{fg}\right]_{i,j}$ is known to hold on the output values.

This rule is quite similar to the differential assertion checking approach using mutual function summaries by Lahiri et al. [Lah+13], but is applied here to frames with potentially more than one invocation and in the context of reactive systems in which the programs are called repeatedly. To allow for that, additional checks (encoded using the counting variables $\texttt{count}_f$ and $\texttt{count}_g$ in κ) have to be included that ensure that the number of invocations of the two abstracted frames are the same in both programs.

Soundness and Completeness. This modularization rule is sound. We miss a formal proof but want do give an argumentation on the critical part of this rule.

An induction proof on the number of coupled invocations (captured in the program variables $\texttt{count}_f$ and $\texttt{count}_g$ introduced for this reason) can be conducted. In particular, the equality of both counter variables ensures that the coupling invariant holds on $P\,;Q$. This coupling invariant exists due to the proof of the regression contract $\{\phi^{fg} \wedge \alpha^{fg}\}\ \pi_f\,;\pi_g\ \{\omega^{fg}\}$ of the subframes f and g, and couples the internal states of both subframes together. Moreover, this coupling invariant is (a) relational (as it talks about program states of f and g), (b) inductive over the calls of f and g, and is relative ([BM07]) to ϕ^{fg} and α^{fg}. The coupling invariant is also valid on $P\,;Q$ (in which f and g are embedded) if the following three conditions hold: (1) f and g are called synchronized (κ), (2) they are only called with similar inputs α^{fg}, and (3) only in the context of ϕ^{fg}. By this invariant, we are allowed to assume ω^{fg} holds for each invocation pair of f and g.

The approach is not complete since we require that both systems invoke their frames equally often. There are systems which fulfill a regression contract but do not have this property. Then this approach can currently not be applied. Also, the completeness can be destroyed by the abstraction of the output variables of the subframe. The given output relation ω^{fg} can be too weak, and yield new behavior in the parent frame which is not triggered by the original implementation. Hence, ω cannot be shown. For example, consider that the subframe only returns a constant value. The abstracted version instead would return an arbitrary value

```
function Reve(f, f'):
    Input: Two frames f, f'
    Data: A regression verification contract (φ, α, ω) for f and f'.
    Output: true iff f and f' together satisfy the contract
    if check cache for (f, f', φ, α, ω) then
        // earlier results are cached
        return cached result;
    end
    if (φ, α, ω) = (true, =, =) then
        // only applicable for perfect equivalence
        return if true EqualSource(f, f');
        return if true EqualSE(f, f');
    end
    return if true EqualSmt(f, f', φ, α, ω);
    return if true EqualAbstraction(f, f', φ, α, ω);
    return EqualityMC(f, f', φ, α, ω);
```

Figure 12.4: Algorithm to check the equivalence of two frames

(but equal value), that allows the parent-frame to execute original unreachable code areas.

The rule is compositional, in the sense that it can be applied recursively on the resulting proof obligations.

12.3 The Algorithm

In this section, we construct a new regression verification algorithm for reactive software that combines a number of different modular and non-modular verification techniques. The algorithm takes two programs and a regression verification contract as input and checks if the programs satisfy the relational specification. We assume both programs have a top-level frame with the identifier `main` that contains all program statements. The algorithm works recursively, comparing first the outermost frames, trying to establish equality from going top to bottom in the program structures, recursively verifying the equality of enclosed subframes.

The algorithm orchestrates different checkers and runs them in sequence returning on the first positive result. In the orchestration, we call the more syntactical, faster, but imprecise checkers first before falling back to more powerful, and more precise, but slower checkers. All checkers are sound: If they report that

frames conform to their contract, then this is the case. They are not necessarily complete, and some checkers are only applicable on a restricted set of cases, for example, perfect equivalence. The full algorithm, shown in Figure 12.4, *is* complete as the last checker $EqualityMC$ uses heavyweight model checking without abstractions and is complete.

The following sections briefly introduce the involved checkers.

12.3.1 Conformance by Syntactical Congruence

In case, that a contract specifies perfect equivalence EQ, the checker *EqualSource* checks equivalence via a comparison of the syntax trees of the two source code artifacts. During parsing the source code into a syntax tree, the code is normalized, in particular, comments and whitespaces are removed, keywords are capitalized, ...). Identical normalized source code implies equal software behavior. In the soundness argumentation, we claim that a successful verification of $RV(C, P, Q)$ implies an inductive coupling invariant over the state variables. Of course, this is true for *EqualSource* checker, but it may not be obvious to see this invariant. Given a regression obligation $RV(EQ, P, Q)$, if the *EqualSource* successfully terminates (syntax trees are equal), then the (inductive coupling) invariant is the equivalence between the states of the programs of P and Q. Note that this equality check also implies the equality initial values of the state variables, which is required to established the coupling invariant before the first execution of both frames.

Despite its severe restrictions, this method is a fast and useful checker, especially for frames resulting from often reused standard library procedures. These libraries functions, e. g., timer function blocks, counters (cf. Sections 4.3.1, 4.3.2 and 7.1) are usually not touched by the application engineer.

12.3.2 Conformance by Symbolic Execution

Checking equality by comparing the source code is very restricted, and fails, e. g., if two independent lines are swapped, or an irrelevant new variable is introduced. Frames A and B in Figure 12.5 shows this situation. The next checker in the orchestration is *EqualSE*, and is able to handle such cases. It is still a syntactical checker; hence, it is also only able to handle perfect equivalence. This checker is based on symbolic execution to compute the symbolic results of a frame.

The result of the symbolic execution of a frame f is a function $F\colon Var \to Expr$ which maps every state and output variable to an expression which is the aggregation of all assignments to the variable in f. The term $F(v)$ computes to the value of v at the end of the frame and may depend on the input and state variables of f.

```
1 FRAME(A) {
2   out1 := s;
3   out2 := in;
4   s := in;
5 }
```

```
1 FRAME(B) {
2   out2 := in;
3   out1 := s;
4   s := in;
5 }
```

```
1 FRAME(C) {
2   out := s;
3   s := in;
4 }
```

```
1 FRAME(D) {
2   out := t;
3   t := in;
4 }
```

Figure 12.5: Four different frames (cf. Equation (12.4)) with input variable `in`, output variables `out1` and `out2`, and state variables `s` and `t`. The Frames A and B are equal and can be checked with *EqualSE* but not with *EqualSource*. Also, Frames C and D are equal, but requires *EqualSE* to infer coupling invariant $s = t$.

One possibility to show perfect equivalence between two frames f and f' is to establish syntactical equality between the symbolic execution results for all output variables. The equality must also be checked for those state variables which occur in the aggregated expressions of output variables to guarantee that the following cycles will produce equal output. In detail, we establish the equality of the state variable as the coupling invariant between the execution of both frames. Note that the initial value is not part of the symbolic execution $F(v)$ of a variable v. Therefore, we need to additionally check the equality of the initial value for each pair of state and output variables to bootstrap the coupling invariant.

Inference of Coupling Invariant. Thus far, we described the case where all input, output, and state variables have the same name in both frames. As an example, consider the situation of Frame C and D in Figure 12.5. To make this analysis more flexible, we allow arbitrary one-to-one mappings of variables between frames where the correspondence of input and output variables is given by a conjunction of equalities between variables in α and ω in the regression verification contract. For state variables, the mapping needs to be inferred. This mapping is also the coupling invariant of both frames, which implies the equality of the match output variables given in ω. In our example with Frame C and D, the inferred mapping matches the state variable s (in Frame C) to the state variable t in (Frame D). Such state mapping is similar to the syntactic unification

Input: α and ω of the regression contract, and the symbolic execution F, F' of the frames f, f'
Data: M is a set of variable pairs v/v' which can be assumed as matched, and Q is a set of variable pairs, which need to be unified.
Output: returns $\bot$ (a matching could be established) or $\top$ (a matching of the state and output variables is established in M)
Function *match(t, t')*
  if $t = t' \vee t/t' \in M$ **then**
    `// Already matched terms`
  else if t, t' *are state variables and each not matched in* M **then**
    `// Match (state) variables, but add them also to the queue.`
    $Q := Q \cup \{t/t'\}\,; M := M \cup \{t/t'\}$;
  else if $t = f(t_1 \ldots, t_n)$ *and* $t' = f(t'_1, \ldots, t'_n)$ **then**
    `//` t, t' `are the same function application, go into recursion.`
    $match(t_i, t'_i)$ for each $1 \leq i \leq n$ abort if $\bot$ occurs;
  else
    abort with $\bot$;
end
$M := \{v/v' \mid$ for all pairs (v, v') defined by α and $\omega\}$;
$Q := \{o/o' \mid$ for all pairs (o, o') defined by $\omega\}$;
while $Q \neq \emptyset$ **do**
  choose $o/o' \in Q\,; Q := Q \setminus \{o/o'\}$;
  $\mathrm{match}(F(o), F'(o))$;
end
return $\top$;

Figure 12.6: The algorithm to infer a mapping of the state variables for a given regression contract of two frames f, f'.

of terms in the first-order logic. An unification is a substitution, s.t. which maps all terms (in a given set) to the same term. In our example, we could simply replace all occurrences of t with s in Frame D, and receive the same program.

For first-order, we are the most general unification is computed the Robinson Algorithm [Rob65] for a given set of terms. Here, the situation is a little bit different: First, we need to consider a pre-existing mapping of the input variables given in α, and only need to "unify" matched outputs defined by ω.

Figure 12.6 shows the algorithm to infer such mapping on the state variables. The algorithm can be split into two parts: the match function and the queue-based iteration. The algorithm maintains two sets: Q containing variable pairs o/o' which need to be checked whether they match, and M the set of variables pair, that can be considered as matched.

The main-loop iterates over the remaining unchecked pairs, these are initialized with the pairs output variables of f and f' defined by ω. But, as the algorithm is executed, new pairs s/s' of state variables are added by the *match*-function. If all pairs are matched, the algorithm terminates successfully, and the matching M is a valid coupling invariant for the output and state variables. (Also, we need to check the initial value for each of the pairs separately.)

The *match*-function recursively compares two given expressions t, t'. If both expression are already equal or matched, no further actions are required. Otherwise, both terms are unequal. If t, t' are variables and t and t' are not already matched differently (in M), we mark them as matched by adding the pair t/t' to M and also we need to ensure this (by adding the pair also to the queue Q) If the t and t' are function applications of the same function, we can recursively match the arguments. The *match*-function fails if either the expression t and t' are not comparable (e. g., different function applications $t = +(1, 1)$ and $t' = g(1, 1)$ or different literals $t = 1$ and $t' = 2$) or t (or t') is already matched with a different variable. If this function fails, the complete algorithm terminates with $\bot$. Our inference algorithm can only infer one-to-one-mapping between state variables.

Note that we give the algorithm in a simplified form (only consider prefix function applications). The implementation need also to consider other expressions, like infix or ternary operators. Furthermore, the mapping can be lifted from equalities over variables to equalities over expressions. But this requires more considerations in the *match*-functions, and may also be combined with the following SMT-based checker to ensure the soundness of the inferred mapping.

We can summarize: The checker *EqualSE* is able to show the equality of $o = 2 * i + s$ and $o' = 2 * i' + t'$, where s, t' are state and i, i' are input variable. A matching needs to include the equality $s = t'$, and $i = i'$. Moreover, the equality of $i = i'$ (input variables) must be justified by the given regression contract ($\alpha \models i = i'$). Due to its syntactical nature, this checker is incomplete, e.g., the equality between $o = 1 + 1$ vs. $o' = 2$, cannot be handled.

12.3.3 Conformance by Reduction to SMT

If these last syntactical checkers fail or are not applicable, the first semantical checker is triggered. This checker is backed up by a reduction to a Satisfiability Modulo Theories (SMT) problem using the previously computed symbolic exe-

cution results $F(v)$ and $F'(v')$ of the given frames. This checker is not limited to perfect equivalence, but can be used for arbitrary regression verification contracts.

The checker *EqualSMT* verifies an inductive relational invariant χ over the state variables of the two frames. In the simplest form we show that any state variables s and s' in f and f' evolve identically (i.e. $s = s'$). The formula to be checked for satisfiability is then

$$\left(\bigwedge_{v \in V} v^+ = F(v)\right) \wedge \left(\bigwedge_{v \in V'} v^+ = F'(v)\right) \wedge \phi \wedge \alpha \wedge \chi \wedge \neg\left(\omega^+ \wedge \chi^+\right) \quad (12.6)$$

where the sets of variables V and V' contain all output and state variables of f and f'. Variable v^+ holds the result of the symbolic execution for v (via the function F or F'). It differs from v to distinguish variables before the execution from after it. A predicate χ^+ results from χ by replacing v with v^+. If this formula is not satisfiable, χ is an inductive invariant for the frames and, additionally, they conform to the regression verification contract (ϕ, α, ω).

As an example, consider the following contract $(true, i = i', o = o')$ for $o = 2 * i + s$ and $o' = t' + 2 * i'$. The instantiated SMT formula (12.6) for this example is

$$\underbrace{(o^+ = 2 * i + s \wedge s = s^+)}_{v^+ = F(v)} \wedge \underbrace{(o'^+ = t' + 2 * i' \wedge t' = t'^+)}_{v'^+ = F'(v)}$$
$$\wedge \underbrace{i = i'}_{\alpha} \wedge \underbrace{s = t'}_{\phi} \wedge \underbrace{o = o'}_{\omega} \wedge \neg(\underbrace{o^+ = o'^+}_{\omega^+} \wedge \underbrace{s = t'}_{\chi^+}) \ ,$$

where o and o' are the output variables, s and t' state variables, and i and i' input variables, respectively. The relational invariant χ has been chosen as $s = t'$ in the example. It is a parameter of the checker, and in general non-trivial to infer. In our implementation, we use the equality of equally named state variables for χ, but our algorithm in Figure 12.6 can also be used to infer a candidate for χ. In a further SMT verification condition (not shown here), it has to be shown that the initial memory states (cf. Section 2.2) of f and f' initially satisfy the coupling invariant χ.

12.3.4 Conformance by Modular Abstraction

The checker *EqualAbstraction* is the checker that exploits the modularization rule introduced in Definition 12.2. Therefore, given two frames f, f', this checker starts with abstracting the top-level frames inside f and f', and uses Figure 12.4 for checking contract conformance of inner subframe pairs.

We assume that the subframes in f and f' are collected in pairs and that each frame pair is specified with a regression verification contract. Let g be a subframe in f, and g' in f', respectively.

After the body of all subframes have been abstracted, we obtain the two factor programs f/g and f'/g' of both original frames together with a regression verification contract that has additional assumptions and post-conditions.

The regression verification algorithm is called recursively for $Reve(g, g')$ of each subframe pair and for $Reve(f/g, f'/g')$.

The modularization rule may be applicable to several subframes. In our implementation, we eagerly apply it to all specified subframe combinations. The recursive procedure is applied recursively and exhaustively, but will eventually terminate since the frames are always finitely nested in a program.

If the modular abstraction step fails, it produces a counterexample (a finite trace, see Section 12.3.5) which may describe a genuine flaw in the system, or it may be spurious if a regression verification contract does not hold on the sub-frame pair or is not strong enough to serve as a suitable abstraction in the proof.

12.3.5 Conformance by Model Checking

The final checker is the most precise and most powerful one and encodes the verification condition into a model checking problem. This checker makes use of the non-modular regression verification approach by Beckert et al. [Bec+15] and verifies a regression verification contract specification between two complete frames f, f' without using abstraction. More precisely, the verification target is a problem in which an invariant (derived from the regression verification contract) for the system consisting of the two compared frames must be verified. Experience has shown that invariant-inferring techniques like the IC3 [BM07] approach (in particular the implementation within the model checker nuXmv [Cav+14]) work quite well for this type of regression verification problems.

Since the state space is finite, this checker is theoretically complete, i. e., returns within finite time for any input. However, experience shows that it can take hours or even days until the model checker comes back with a result. The modularization technique and the combination with simpler techniques in $Reve$ have been devised to reduce the use of such heavy-weight verification, and hence reduce the time needed for regression verification challenges.

The model checker returns either that the invariant has been proved (implying correctness of the contract), then inductive invariant could be inferred by IC3 (Section 2.3). Or the checker produces a counterexample, which is a concrete trace, i. e., finite sequence of assignments of input, state, and output variables for both frames exemplifying the violation of the contract. We currently do not

provide tool support, but these values can be used as inputs for a simulation of the reactive system like it is present in many modern IDEs for reactive software.

12.4 Evaluation

In this section, we show the applicability of our new regression verification algorithm on selected scenarios of the Pick-and-Place Unit (PPU) community demonstrator (cf. Section 7.3,[Bec+15]. The PPU is a down-scaled model of a manufacturing plant employing industry-level hardware components that has been designed for researching the management of the evolution (hardware and software) of automated manufacturing systems. Therefore, there are multiple evolution scenarios, with software and/or hardware changes, of this plant. We selected representative evolution scenarios to cover different situations.

12.4.1 Selected Evolution Scenarios

We briefly explain the three selected evolution scenarios. The software revisions correspond to the different scenarios of the PPU in [Vog+14, Fig. 48].

Revision 1 vs. Revision 2 A new sensor is introduced for detecting metallic workpieces as a preparation for the next evolution. The software mainly changes the Crane module, but changes on the top-level module are needed to route the sensors to this submodule. An influence on the system behavior is not expected: Both revisions are perfectly equivalent.

Revision 3 vs. Revision 5 Revision 5 introduces an optimization which allows using the waiting time during stamping to transport workpieces which do not need to be stamped to the conveyor belt. The optimization is only triggered if workpieces of different types are present (metallic and non-metallic). If only metallic workpieces are present, the two revisions behave perfectly equivalently. The workpiece type can be determined by the program using the input variable $CapacitiveSensor$. We obtain a regression contract $(CapacitiveSensor = true, =, =)$ which intuitively formalizes that the old and new revisions behave equivalently (equal inputs give equal outputs) under the condition that the sensor variable $CapacitiveSensor$ is $true$ in every cycle.

Revision 12 vs. Revision 13 In the old revision, the position of the crane is measured with three switches (with Boolean sensor values *OnConveyor*, *OnMagazin* and *OnStamp*). These are replaced by a single angular sensor. We need

to define a relation R between the three boolean sensor values and the angle position

$$\begin{aligned}(16160 < \mathit{AnalogPosition} \wedge \mathit{AnalogPosition} < 16260) &= \mathit{OnConveyor} \wedge \\ (24290 < \mathit{AnalogPosition} \wedge \mathit{AnalogPosition} < 24390) &= \mathit{OnMagazin} \wedge \\ (8160 < \mathit{AnalogPosition} \wedge \mathit{AnalogPosition} < 8260) &= \mathit{OnStamp}\end{aligned}$$

which serves the relational precondition in the regression verification contract $(\mathit{true}, R, =)$.

12.4.2 Results

Table 12.7 summarizes the performance of the verification. The runtimes are shown for each checker on a frame. The first column describes the compared revisions and modules, where *Main* or *Crane* denotes the regression verification between the corresponding frames of both revisions. *Main/** denotes the frame with all subframes factored out. For convenience, Table 12.7 only shows the first and second level of nested frames. In particular, the frequently used timer module is hidden.

"Non-Modular Total" is the comparison reference value of applying the non-modular approach as in [Bec+15] with our pipeline. In comparison, "Modular Total" gives the overall runtime of the modular pipeline. Both total columns state the runtime measured from the command line. Hence, they include the work needed to prepare the programs (parsing, symbolic execution, etc.). In contrast, the checker runtimes are given in milliseconds and are measured internally. A checker is skipped (marked with a dash (–) in the table) if either it was not capable of proving the regression contract, or a checker invoked earlier was able to solve this case. Note that for the comparison of Rev 12. vs. Rev 13 ("12 vs.13 −SE"), we have disabled $\mathit{EqualSE}$ to evaluate the modularization rule, because we want to demonstrate the capabilities of the decomposition rule. $\mathit{EqualSE}$ can solve this comparison directly in half a second (cf. "12 vs.13 +SE" in Table 12.7). The lines of code do not include empty lines or comments and cover both code modules. Also, the number of variables (#Vars) is the sum of input, state, and output variables of both frames.

The runtimes (wall clock) are the median of three samples, computed on an Intel Core i7-8565U, 16 GB RAM, using the model checker nuXmv 1.1.1 [Cav+14] with IC3 for invariant checking, and z3 4.8.8 for solving the SMT instances. The time-out was set to 1 hour. Our algorithm implementation is single-threaded. All verification artifacts and a link to the source code are available in the companion material [Wei21] or online[1].

[1] http://formal.iti.kit.edu/isola20

Table 12.7: Results of the regression verification algorithm.

Rev./Module	Runtime							Code Size	
	Non-Mod.	Modular	Checkers [ms]						
	Total [s]	Total [s]	Src	SE	SMT	Modul.	Classic	LoC	#Vars
1 vs. 2	11.34	2.27						744	136
Main			0	98	108	933	–	744	136
– Main/*			0	17	–	–	–	174	203
Crane			0	52	56	756	–	415	51
– Crane/*			1	80	51	–	601	403	207
Magazine			0	21	–	–	–	234	38
3 vs. 5	823.36	9.49						1,605	256
Main			–	–	296	6,765	–	1,605	256
– Main/*			–	–	74	–	3,540	294	364
Crane			–	–	106	2,989	–	810	74
– Crane/*			–	–	85	–	2,890	768	376
Stamp			0	–	–	–	–	402	56
Magazine			0	–	–	–	–	240	44
12 vs. 13 −SE	t/o	36.85						4,808	520
Main			–	–	593	27,324	–	4,808	520
– Main/*			–	–	63	–	7,904	453	1,250
Conveyor			0	–	–	–	–	468	50
Crane			–	–	177	16,074	–	1,326	77
– Crane/*			0	–	153	–	15,904	1,284	631
Pusher			3	–	–	–	–	2,144	154
Stamp			0	–	82	4,801	–	403	57
Stamp/*			0	–	–	–	4,680	375	639
Magazine			0	–	61	–	2,892	241	45
12 vs. 13 +SE	t/o	9.56						4,808	520
Main			0	397	–	-	–	4,808	520

12.4.3 Discussion

The evaluation shows a huge speed-up against the previous non-modular approach from [Bec+15]. It shows the potential of modularization to enable the handling of large reactive systems. For a fair comparison, we repeated the experiments of [Bec+15], but we use the default bit-width for integers on PLC languages, and also we did not reduce the blocking time of the used timers. Rev. 12 against Rev. 13 ran into a time-out, [Bec+15] gives a clue that the verification can take more than 22 hours. Most of the performance should result from abstracting these timers, which are used to wait a particular amount of time. During this time span, the system stutters partially, resulting in long phases of forwarding searches in IC3.

12.5 Conclusion

In this chapter, we have motivated and presented a new verification rule for the modular decomposition of regression verification proof obligations for reactive system software. Moreover, we have integrated the rule into a novel regression verification algorithm which orchestrates five different regression verification approaches into one proof technique. Thanks to the modularization, simpler equality checkers allow one to show properties more easily on subproblems.

The evaluation indicates a tremendous performance improvement: Modularization can allow regression verification proofs to run orders of magnitudes faster.

Complex Contracts with RTTs. In contrast to RTT, we used a simplified specification for our regression contracts. It is possible to extend our modularization approach to stateful specifications like RTT.

Let us consider the situation, that an RTT describes the relation between two sub-frames. The RTT couples the input and outputs of both frames together, similar to our regression contract, but now depending on its current state. The verification that the sub-frames conform to the RTT is already covered by the verification pipeline for RTTs. But to use the RTT as an abstraction in the factor programs, we need adaptions. First, we need to encode automaton (representing the RTT) in the verification target such that it asserts the relation on the inputs constraints, and assumes the relation on the output constraints. Moreover, the automaton is not part of one of both program runs (old and new revision). Moreover, the automaton is more like an observer over both runs. Second, the difficult part is, that a transition on the automaton needs to be invoked every time the sub-frame pair is called, and this can happens multiple times in each execution cycle. Therefore, the RTT is not synchronized with the execution cycle of both program runs.

Using RTTs seems feasible on the technical level, but on the conceptual level they have disadvantages: First, they need a state. The state is something we want to get rid of. Also, it is not excluded that the verification target of the factor programs is larger in the model size than the original programs. Second, an RTT is (often) not a complete specification covering every possible input sequence. Therefore, we need multiple RTTs for a sub-frame pair, but this arises different conceptual problems: the under- and over-specification of sub-frames. Under-specification states that there might still exist inputs sequences that are missed by all given RTTs. Hence, we would need to add more RTTs. But if we have too many RTTs, it could easily happen that these are contradictory to each other s.t. there exists no system conforming all RTTs for every input sequence.

Multiple Contracts. We considered that only a single contract for each frame pair is available. This is sufficient for our experiments on the PPU. In this section, we sketch the extension of our approach for multiple contracts on frame pairs. We distinguish between two usage scenarios: Firstly, all occurrences of a sub-frame are abstracted with the same contract or, secondly, the contract is individually selected for each occurrence. In the first case, our decomposition rule in Definition 12.2) remains applicable for the selected contract c without further changes to the rule. We only have to insert the correct formulas $(\phi_c^{fg}, \alpha_c^{fg}, \omega_c^{fg})$ of the selected contract c. In the second case, we need to reconsider Γ, and the regression verification on sub-frames. In particular for the factor programs, each occurrence of f in P and g in Q is replaced with the same nondeterministic behavior, only the relational contracts, which bind the input and output values together, are different. This affects Γ which encapsulates the possible and applicable contracts. Γ becomes aware of the k contracts:

$$\Gamma = \bigwedge_{c=1}^{k} \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} \left[\phi_c^{fg} \wedge \kappa \wedge \alpha_c^{fg} \rightarrow \omega_c^{fg}\right]_{i,j}$$

Now, we enable the different contracts on the "right side" of our decomposition rule, but we need proof that the contracts are adhered to by the sub-frames ("left side" of the rule). We cannot prove each contract individually, because the contracts can be used in arbitrary order in the factor programs. Hence, we need to construct a combined contract which expresses this arbitrary usage:

$$\left\{\bigvee_{c=1}^{k} \phi_c^{fg} \wedge \alpha_c^{fg}\right\} \; \pi_f \,;\, \pi_g \; \left\{\bigwedge_{c=1}^{k} \left(\mathrm{prev}\left(\phi_c^{fg} \wedge \alpha_c^{fg}\right) \rightarrow \omega_c^{fg}\right)\right\}$$

The premise states that at least one function precondition ϕ_c^{fg} and input relation α_c^{fg} of the same contract c have to hold. After the execution of $\pi_f \,;\, \pi_g$, the conclusion enforces the output relation ω_c^{fg} holds when the corresponding precondition and input relation of the same contract c hold in the previous state before execution of both sub-frames (denoted with prev).[2] Due to the new regression goal, we prove $\pi_f \,;\, \pi_g$ for every possible combination of contracts. This might be an over-approximation of the actual use case of the frames, in which for example, two contracts are always applied alternately. The example is expressible with RTTs. Note that it is sufficient to consider the subset of required or used contracts in the application of the decomposition. Therefore, not all available contracts of the sub-frames might be necessary for the verification, and can be left out in the construction of the regression goal. But it might be more

[2] ϕ and α are formulas over the input variables which are normally read-only, hence prev is not required, but for clarity explicit state this circumstance.

efficient to follow a verification strategy that tries prove all available regression contracts for frequently used frames once, than to multiple prove regression goals with individual subsets with the required contracts.

Future Work. We rather see the need for (user-specified) regression contracts for the sub-frame as a drawback of our decomposition technique. Therefore, before investigating the use of RTTs for these contracts, we tend to reduce the specification effort. In most cases, these regression contracts seem to be automatically inferable, e. g., by using heuristics, symbolic execution, or Horn solvers.

In our implementation, we have not used any sophisticated strategy to decide whether a frame should rather be kept inlined or be abstracted. The implementation tries to abstract all allowed frames at once, which seems to be a good strategy considering our experiments. But a more restrictive selection could bring a further advantage. To develop such a strategy further experiences are needed, especially in cases in which abstraction decreases the verification performance.

Chapter 13

Conclusion

13.1 Summary of the Thesis

Specification Languages. In this thesis, we presented generalized and relational test tables, two novel table-based specification languages. Moreover, we provided decision procedures for these specification languages along with experiments and evaluation.

The table-based specifications have advantages: They are currently used as concrete test tables in the testing of automated production systems. Engineers can start with concrete tables which they can successively generalized with the features of GTTs. If there exists software with similar behavior, the software can be used for specification by switching to RTTs. The upgrade from a GTT to RTT is straight forwarded through their shared syntax and semantics. In general, GTTs and RTTs offer a structured view of constraints on variable and their change over time, that is hardly found in established specification languages.

The test tables inherit the weakness of concrete test tables, that they are designed to cover only a family of typical behaviors of a program. It is hard to achieve a full system specification with test tables. Therefore, GTTs and RTTs are not useful for compositional verification. For a detailed discussion, refer to Sections 9.1 and 10.5.

Forgetting of Information. Additionally, we used RTTs to formalize a new security property: the forgetting of information. This specifies whether a secret is finally forgotten after a specified period. This is the first notion of this kind for reactive systems. Such a verified system does not prevent successful intrusions (which also may happen on a completely different level, e. g., maintenance access), but limits the knowledge gain of the attacker. The biggest weakness of

our approach is the limitation to the PLC software as they might be leaks through the hardware, operation system, or the physical environment. Nonetheless, our property provides a useful assessment of the confidentiality of a reactive program, as it provides a quantifiable and comparable scale of confidentiality.

Modular Regression Verification. We presented a novel contract decomposition rule and a new recursive algorithm for regression verification. The decomposition rule allows to split up large regression verification goals into smaller sub-goals, by using contracts of software modules instead of implementation of the module. The smaller sub-goals can become quite simple, so simple that simple and fast equality checkers become useful. These fast checks range from simple source code comparison to the unification of programs, and SMT-based coupling invariant verification. The new algorithm is tremendously fast, sound, and complete by orchestrating all simple checkers, the decomposition rule, and the fallback of the complete model-checker The disadvantage of this approach is the requirement of the relational specification for each pair of software modules, which should be abstracted.

All our contributions are implemented and publicly available under the VERIFAPS at https://github.com/verifaps/verifaps-lib.

With our contributions, we follow the idea of reducing the obstacle for engineers to apply formal verification during the development. Either by introducing a new specification language (GTTS), by exploiting existing programs for the specification (RTTS, regression verification) or by improving the verification performance. Despite the thesis' focus on automated production systems, the presented contribution can also be applied to reactive systems of other domains. With a different interpretation, the test tables are applicable to event-based reactive systems.

13.2 Future Work

The story of the test table is not finished with this thesis, besides follow-up projects they are limitations on these specification languages which need to be overcome.

Rigidness of Time Constraints. In contrast to timing diagrams (Section 3.2), the time constraints on GTTS and RTTS are rigid. This means, that only concrete intervals are permitted, which cannot depend on program or global variables. The reason is that we need to unwind the table rows for the static verification because we can be in a row at different iterations at once. Additionally, using program variables lead to new issues in the semantics, e. g., what should happen

if a program variable (in a time constraint of an active row) changes its value? Should the current value considered, or the value at the entry table row? For dynamical verification, we can lift this limitation by using a token that captures the current table row and row iteration. The drawback is we do not know the upper limit of these tokens, hence, we may require an arbitrary amount of memory.

A possible extension can be copied from timing diagrams, (cf. [Fis99]) which permits a new signature of variables exclusively for modeling time constraints between events. For GTT, we could also introduce new variables which are only usable in time constraints and also bounded. The new variables would give us the possibility of stating requirements like "the amount of row iterations of the first and second row are equal".

From GTTs to Contract Automata. In the future project (see below), we are planning to go ahead with the shape of generated automata from GTTs. These automata have the characteristic, that on their edges are contracts. Currently, in the form of pre- and post-condition of the row. We want to use these automata kind, and develop it into a useful specification on its own. This means, our specification is an automaton, which has different contracts on its edges which determines the next possible states. A contract is an n-tuple with an assertion on each party (e. g., challenger and system) and a time constraint. If a contract is fulfilled from all parties and also the time constraint is fulfilled, the automaton selects the incoming state as active. Otherwise, the error state for the violating party is selected. Of course, nondeterminism and global variables will be present.

These *contract automata* allow a more general and complete specification of software modules. This is the key for compositional verification. Additionally, we hope for a better feeling of control over the specification and more expressiveness. On the other hand, a large automaton is harder to read and understand. These issues need to be tackled with a graphical notation and solid tool support.

Note that contract automata are an extension of our meshed GTTs (Section 9.2), and the generalized game (Section 9.3).

Better Tool Support. Our library VERIFAPS as a backend aims at verification experts and not the engineers as the end-users. Nonetheless, it offers quite suitable support. For example, GTTs can be translated for debugging into spreadsheet files. A spreadsheet file allows the user to insert a sequence of input and output values, and the inserted formulas of the constraints in the GTT give feedback on whether and when constraints are violated. Such a spreadsheet can also be generated where the output values are computed—a given program is encoded into spreadsheet formulas. VERIFAPS also visualizes counter-example

of failed proofs attempts of GTTS. A counter-example is visualized by printing the simplified program (Section 6.3.1) for each cycle with inlined values for each assignment and branch condition. Also, for each cycle, the values of the input and state variables are printed at the beginning and the value of the output variables at the end. Additionally, for each cycle, the state of the automaton is mapped to the table, showing which rows were active, and which assumptions or assertions failed (down to the violated columns).

But for the engineer, VERIFAPS do not other any easily accessible interface. A prototype STVS (Figure 6.4) was creating in a students project and support the state of GTTS presented in [Bec+17], but is not adapted to newer GTTS-features. On the other hand, to be accessible for the engineers, we have to bring our approaches into their daily tools for development and testing. This is one goal of a submitted DFG transfer project.

13.3 Follow-up Projects

Furthermore, there are concrete plans for two follow-up projects for the further development and research on the presented approaches.

Transfer Project. A DFG transfer project was submitted in which we bring our approaches to the engineering practice. The characteristics of a transfer project are the participation of industry partners. In our case, the industrial partner is an engineering company of automated production systems. In this project, we want to make functional, regression, and relational verification feasible in the daily engineering practice by lowering the obstacles and lower the complexity.

We use regression verification as a tool for the software assessment in variant management. Often, the industry does not use explicit variant management, in which a single product tailored to the individual needs of a customer, is generated or assembled from a larger software project containing the various inter-compatible components. Instead, the variants are maintained individually and independent of each other—often by copying the latest previous software version and adapting it. In case of an issue, all variants have to be patched individually. Moreover, it is often not clear, which variants are affected by an issue and thus require a fix.

Regression verification should help in the construction of the 150%-models of variants. A 150%-model of software is a common way to describe variants with their common and specific software pieces. In this scenario, regression verification can be applied as an equivalence checker between software variants to find identify the common behavior. Also, regression verification helps in the

management of patches. Issues are always found in specific variants, and often they are patched on this single variant. To avoid, a diverge in the variants, the issue and the patch must be evaluated on the other variants. With regression verification, we can recognize whether other variants suffer under the same issue and whether the suggested patch solves the issues on these variants, too.

As patches are a local change in the code basis, it is not very handy to investigate and validate a complete PLC software (again). Therefore, we investigate lightweight syntactical difference analyses to recognize which parts of the verification subjects need to be revalidated with regression verification. This technique of shrinking the software to a minimal amount of code is called *program slicing*. In contrast to traditional program slicing techniques for single programs, we need to build a program slicing that considers two programs at once.

Besides regression verification, we also plan to apply GTTs on industrial examples including the adaption to their programming dialects and domain. We consider the use GTTs for expressing the functional behavior of the local patches or patched area. Currently, we plan with limited use of RTT-specification of the engineer. We plan to provide an ensemble of testers for different equivalence and similarities of variable groups or single variables. These ensembles might be defined as RTTs, but will stay hidden from the engineer.

Software-defined Car. In a different proposal, we bring the idea of GTT-like functional specification in the automotive industry. For this, we extend flexibility and expressiveness of GTTs, especially, to cover full-system specifications. To achieve this, we follow the idea of the contract automata (the automata generated from GTTs), and extend the GTTs as a frontend, or introduce new frontends for the specification, e. g., state machines. In the core, such an automaton determines the currently available contracts (pre- and post-conditions) for each time point. Additionally, the frontend should permit the specification of variants.

We plan to use the contract automata for the static verification, as well as, the dynamic verification via generated monitors. In contrast to the partial behavior description by the GTTs, contract automata allow modular and compositional verification.

Final Remark.

We believe that formal methods have shown and will show a substantial difference in the development of software and reactive systems. Especially, formal verification can guarantee strong safety and – currently not well-established – security requirements. Also, formal verification requires to be accompanied

by a rigorous development process and runtime checks to bring the previously proofed guarantees into the operation, and carry them over the decades.

Although, their advantages, they are currently not present in the practice of the development. With this thesis, we provide a new kind of table-based formal specification languages that can capture functional and relational properties. We think these languages are easier to learn and to adapt in the development practice, mainly because their predecessors are already used.

Appendix A

Glossary

In this appendix, we gather the used notions across this thesis to allow an easy lookup.

Name of Tools and Proper Names.

aPS stands for automated production system, used synonymously for automated manufacturing system or plant

CodeSys is an IDE for the development of IEC 61131-3 software.

eldarica is a solver for Horn-clauses which also provides a frontend for C-programs, see [HR18] and used Section 6.2.

FBD Function Block Diagram—a graphical programming language of IEC 61131-3 (Section 2.2).

geteta is the implementation of the verification pipeline for GTTS (Chapter 6 and Section 6.3).

IC3 is technique for checking invariants (Section 2.3).

IL Instruction List is a textual assembler-like programming language of the IEC 61131-3 standard (Section 2.2).

LD Ladder Diagram is a graphical programming language of IEC 61131-3 (Section 2.2).

nuXmv is a symbolical model checker, see [Cav+14].

SeaHorn is a modern C-program verifier, see [Gur+15] used in Section 6.2.

SFC Sequential Function Chart is a graphical automata-like programming language of IEC 61131-3 (Section 2.2)

ST Structured Text is a textual Pascal-like programming language of IEC 61131-3 (Section 2.2).

ST_0 is a reduced Structured Text dialect and our intermediate representation (Definition 6.9).

STVS is a graphical interface for the GETETA and the verification of GTTs (see Figure 6.4).

VERIFAPS is our verification library for aPS (Sections 1.1 and 6.3).

Basic Notations.

Kripke structure is a graph-like mathematical description for systems, defined in Section 2.3.

$\mathbb{N}$ denotes the set of natural numbers.

$Seq(X)$ denotes the set of all finite sequences other the set X ($Seq(X) = \bigcup_{i \leq 0} X^i$).

Notations for Reactive Programs.

reactive program Source code (Syntax) which is repeatedly executed (Definition 5.1) by a reactive system.

reactive system A system, which periodically triggers the reactive program (Definition 5.2 and Section 2.1).

$InVar, OutVar, StateVar$ are the sets of input, output or state variables of a reactive system or program. $\Sigma = InVar \cup OutVar \cup StateVar$ denotes the complete variable signature.

$\mathcal{I}, \mathcal{O}, \mathcal{S}$ are the sets of the input values, output values or state values of a reactive system or program. These sets are the cartesian product of the value domains of each input, output or state variable.

Notations for Generalized Test Table.

ϕ, ψ are often used as placeholder for a symbolic cell constraints $\mathfrak{E}$. ϕ is the assumption or pre-condition, and ψ is the assertion or post-conditon.

τ is used as a time constraints $\mathfrak{T}$.

$\mathrm{SP}(T)$ denotes the set of symbolical plays for a GTT T (Section 5.3.2).

$\widehat{\mathrm{SP}}$ denotes the current state of the game regarding to the remaining possible plays (Figure 5.1). Simetimes $\widehat{\mathrm{SP}}^k$ used with $k \in \mathbb{N}$ to denote a particular round k in a play.

pre, $post$ are indices to be used on state variables in cell content (Definition 5.4) to denote whether the state variable is evaluated in the state before or after the execution of the system.

next is a function in cell constraints (Definition 5.5) which denotes that the state variable is evaluated in the latest state (cf. Definition 5.10).

$\mathfrak{E}$ denotes the set of all cell constraints (Definition 5.5).

$\mathbb{T}$ denotes the set of all GTTs (Definition 5.7).

$\mathfrak{T}$ denotes the set of all time expressions (Definition 5.6)

$\mathcal{M}(T)$ the monitor of GTT T (Definition 8.1)

$succ(r)$ denotes the successor rows of r. We use $succ(0)$ to denote the initial reachable rows. (Definition 5.8)

Regression Verification

(ϕ, α, ω) is a semantical regression (verification) contract (Section 12.2.2).

$RV(C, P, Q)$ is a regression proof obligation or goal for a given regression contract C and two reactive programs P, Q (Section 12.2.2).

Appendix B

Source Code

B.1 Function Block `MinMaxWarning`

```
1  TYPE
2    OperationMode : (Learn , Active);
3  END_TYPE
4
5  PROGRAM MinMax
6
7  VAR CONSTANT
8      WAIT_BEFORE_WARNING : INT := 10; // Amount of cycles outside
           before given a warning.
9      WAIT_AFTER_WARNING : INT := 5; // Amount of cycles inside,
           before withdraw warning.
10 END_VAR
11
12 VAR_INPUT
13   mode : OperationMode; // learning or active mode
14   learn : BOOL; // True iff current signal should be learnt
15   I : INT; // input signal
16 END_VAR
17
18 VAR_OUTPUT
19         W : BOOL;
20         Q : INT;
21 END_VAR
22
```

```
23 VAR
24         lower : INT := 32766; // minimal teached value
25         upper : INT := -32767; // maximal teach value
26         cntQuench : INT := 5; // remaining cycles for withdraw
               warning
27         cntHeat : INT := 10; // remaining cycles to signal
               warning
28 END_VAR
29
30
31 IF mode = OperationMode#Learn THEN // go into learning
32         IF learn THEN
33                 lower := MIN(lower, I);
34                 upper := MAX(upper, I);
35         END_IF
36         Q := 0;
37         W := FALSE;
38 ELSE
39         IF upper < lower THEN
40                 Q := 0;
41                 W := TRUE;
42         ELSE
43                 Q := MIN( upper, MAX( lower, I) );
44                 IF I <> Q THEN
45                         cntHeat := cntHeat - 1;
46                         cntQuench := WAIT_AFTER_WARNING;
47                         IF cntHeat = 0 THEN
48                                 W := TRUE;
49                         END_IF
50                 ELSE
51                         cntQuench := cntQuench - 1;
52                         cntHeat := WAIT_BEFORE_WARNING;
53                         IF cntQuench = 0 THEN
54                                 W := FALSE;
55                         END_IF
56                 END_IF
57          END_IF
58 END_IF
59
60 END_PROGRAM
```

Listing B.1: Function Block `MinMaxWarning` for clamping a value into a learnt range

B.2 Function Block LinRe

```
1  TYPE
2    OpMode : (Op , Teach);
3    TeachStatus : (Ok, NoTeachPoints, Teaching, InvalidTeachPoints,
         InvalidInputValue);
4  END_TYPE
5
6  PROGRAM LinRe
7
8  VAR CONSTANT
9    SENSORINPUT_MIN : INT := 0;
10   SENSORINPUT_MAX : INT := 4095;
11   TEACHTIMEOUT : TIME := TIME#20s0ms;
12   ICONST : INT := 1000;
13 END_VAR
14
15 VAR_INPUT
16   TPSet : BOOL; SensorInput, TPy : INT; OperationMode : OpMode;
17 END_VAR
18
19 VAR_OUTPUT SensorOutput : INT; END_VAR
20
21 VAR
22   initial : BOOL;
23   lastmode : OpMode;
24   Status : TeachStatus;
25   teachstep : INT;
26   x_temp, y_temp, t, m, x1, y1, x2, y2 : INT;
27   timeout  : TON;
28 END_VAR
29
30 IF NOT initial THEN
31   x1 := 0;
32   y1 := 0;
33   x2 := 0;
34   y2 := 0;
35   Status := TeachStatus#NoTeachPoints;
36   initial := TRUE;
37 ELSE
38   IF OperationMode = OpMode#Op THEN
```

```
39      IF Status = TeachStatus#Ok AND SensorInput >= SENSORINPUT_MIN
            AND SensorInput <= SENSORINPUT_MAX THEN
40        SensorOutput := m * SensorInput / ICONST + t;
41      ELSE
42        SensorOutput := 0;
43      END_IF;
44    ELSIF OperationMode = OpMode#Teach THEN
45      IF lastmode <> OperationMode THEN
46        teachstep := 0;
47        timeout(IN := FALSE);
48      END_IF;
49
50      CASE teachstep OF
51        0:
52           Status := TeachStatus#Teaching;
53           timeout(IN := TRUE, PT:=TEACHTIMEOUT);
54           IF timeout.Q THEN OperationMode := OpMode#Op; END_IF;
55
56           IF TPSet AND SensorInput >= SENSORINPUT_MIN AND
                 SensorInput <= SENSORINPUT_MAX THEN
57             y1 := TPy;
58             x1 := SensorInput;
59             teachstep := 1;
60             timeout(IN := FALSE);
61           END_IF;
62        1:
63           IF TPSet = FALSE THEN
64             teachstep := 2;
65           END_IF
66        2:
67           timeout(IN := TRUE, PT:=TEACHTIMEOUT);
68
69           IF timeout.Q THEN OperationMode := OpMode#Op; END_IF;
70
71           IF TPSet AND SensorInput >= SENSORINPUT_MIN
72               AND SensorInput <= SENSORINPUT_MAX THEN
73             y2 := TPy; x2 := SensorInput;
74             teachstep := 3; timeout(IN := FALSE);
75           END_IF;
76         3:
77          IF x1 > x2 THEN
78            x_temp := x1;  y_temp := y1;  x1 := x2;  y1 := y2;
79                 x2 := x_temp;  y2 := y_temp;
```

```
80          END_IF;
81
82          IF x2 - x1 <> 0 THEN
83            m := (y2 - y1) * ICONST / (x2 - x1);
84            t := y1 - m * x1 / ICONST;
85            Status := TeachStatus#Ok;
86          ELSE
87            Status := TeachStatus#InvalidTeachPoints;
88            OperationMode := OpMode#Op;
89          END_IF
90        END_CASE
91    END_IF
92  END_IF
93    lastmode := OperationMode;
94  END_PROGRAM
```

Listing B.2: Function Block LinRe for the linear interpolation of sensor values

List of Figures

1 A GTT for the MinMaxWarning function block Appendix B.1. iv
2 RTT for the equal behavior after reset v

3.2 Example specification from [DBM15, Figure 3] describing a Mealy automaton. 25
3.4 An example of a timing diagram from [Fis99, Figure 1.]. 29
3.5 A specification for an elevator: $at\$i$ is true if the elevator is at the ith floor. 32

4.1 Example for a CTT with two input variables, two output variables covering 10 cycles of the system. 40
4.2 Example for a generalized test table with a global variable p. 42
4.5 A GTT for Function Block RS 48
4.7 A GTT for the TP function block. 50
4.8 Example of a nested GTT for a conveyor belt. 51
4.9 An example GTT for a solar thermal system. 51
4.10 Three GTTs where each GTT describes exactly two CTTs and has a different interpretation in the conformance. 53

5.1 Game between challenger and system w.r.t. a GTT T 68
5.2 GTT illustrating the difference between strict and weak conformance 69
5.3 A GTT with a global variable p, without any compliant system. . . . 75
5.4 A GTT with a global variable p specifying the identity function for the first cycle. 75
5.5 Two GTTs with a GTT-local variable p, which can be assigned. 77

6.1 Model-Checking Pipeline 80
6.2 A normalized table and the successor relation on its rows. 81

6.3 Sketch of the automaton generated for the GTT of Figure 4.9 84
6.4 Screenshot of STVS 87
6.5 Horn-based verification pipeline for GTTS 87
6.7 Preprocessing for ST_0 97
6.8 Grammar of the input language for GTTS. $\langle cell \rangle$ and $\langle expr \rangle$ are defined in Definition 5.4 and $\langle time \rangle$ in Definition 5.3 100

7.2 GTTS for CTU 105
7.4 GTT for Function Block DEBOUNCE, where meta-variable T represents an arbitrary waiting time, and is instantiated to retrieve a valid GTT. 107
7.5 Schematic view of the investigated function block with a state machine describing its operation. 109
7.6 Concrete test table of analog sensor function block 109
7.7 Generalized test table of function block for linear re-scaling, where L is the linear regression, see (7.1). 110
7.8 Three GTTS for the specification of the MinMaxWarning's behavior . . 111
7.9 The Pick-and-Place Unit in a medium-sized configuration consisting a stack, crane, stamp, and the conveyor belt with different ramps and pushers. Figure provided by Institute of Automation and Information Systems from Technical University of Munich (TUM) 113
7.10 GTT for the emergency stop behavior of the PPU 114
7.11 A GTT for the crane-as-buffer maneuver to bypass the Stamp and improve overall workpiece throughput. 116
7.12 A GTT specifying the PLC software's behavior for sorting black work pieces. 117
7.13 A GTT specifying the PLC software's behavior for sorting white non-metal work pieces. 117

8.3 GTT for the cruise control system which is restarted when the CraseState is set to off 134
8.4 GTT describing a material flow 134

9.1 A GTT equivalent to $\mathbf{F}In$ under strict or cooperative conformance. . 142
9.2 Grinding and sorting processing unit 145
9.3 Example for a meshed GTT 147
9.4 Generalized game. C represents the nondeterministic challenger, and S the system, respectively. 149

10.2 Example for an RTT 161
10.3 Additions and changes to the grammar Figure 6.8 for specifying RTTS 162
10.5 Scheme of an augmented program 166
10.7 Verification pipeline for RTTS and ST_0 code 169

10.8 Combination of regression and delta verification. 170
10.9 RTT for equal behavior after reset 172
10.10 RTT for the regression verification between three programs 173
10.11 RTT for a information flow property 174

11.2 Template of RTT for information forgetting with an annealing phase of length k 183
11.3 Automaton checking the RTT in Figure 11.2 184
11.4 Hardware components of the system to be verified. 186
11.5 Architecture of the software consisting out of four structural elements: main program, sequence automaton, main axis control and HMI. 187

12.1 Schematic of the plant consisting of a conveyor belt B with two processing stations s_1 and s_2. 196
12.2 Sketch of the program flow of the motivating example: the original revision on the left and the adapted revision on the right. 196
12.4 Algorithm to check the equivalence of two frames 203
12.5 Four different frames *EqualSE* 205
12.6 The algorithm to infer a mapping of the state variables for a given regression contract of two frames f, f'. 206

Listings

3.3 Example specification of a stop watch in CocoSpec. 28
4.4 Function Block RS 48
4.6 Function block PT 49
6.6 Sketch of the monitored program. 93
6.9 Textual representation of Figure 4.7 101
7.1 Function Block counting-up on rising edges. 105
7.3 Function Block DEBOUNCE 106
8.1 The C++ interface of GTT-monitors 132
8.2 The implementation skeleton of GTT-monitors 133
10.4 The textual representation of the RTT in Figure 10.2. 164
11.1 Simple program to control a baffle gate. The baffle get is unblocked for T cycles after an user has authorized. 179
12.3 A frame constellation in which it is forbidden to abstract Frame B as this makes <unreachable> reachable. 200
B.1 Function Block MinMaxWarning for clamping a value into a learnt range 227
B.2 Function Block LinRe for the linear interpolation of sensor values 229

List of Tables

3.1 "Number of studies related to different programming languages in 5-year periods From: An overview of model checking practices on verification of PLC software" [Ova+16, Table 13] 22

4.3 Constraint abbreviations from [Bec+17]. X is the name of the variable that the cell corresponds to; n, m are arbitrary expressions of type integer; α, β are abbreviations or formulas. 43

7.14 Statistics for the verification of weak conformance. CPU times are the median of five samples if the verification was successful. Otherwise, "unsafe" denotes that the verifier detected an error, "t/o" the time-out of 10 minutes was reached, "oom" out-of-memory or "err" the verifier was not able to parse the given input file. 120

7.15 Alternative verifications for Function Block `MinMaxWarning`. Table gives runtimes for the verification of strict conformance with nuXmv and weak conformance with ic3ia. Runtimes are cpu time given as median of five samples. Note that weak conformance verification with ic3ia have no bound on the global variables. 121

10.1 Constraints abbreviations for relational references. 158

10.6 Runtime of the relational verification for our application scenarios . 168

11.6 Runtime of the model-checker to prove the forgetting of information 190

12.7 Results of the regression verification algorithm. 212

Bibliography

[AT11] Jagannath Aghav and Ashwin Tumma. "ESTEREL IMPLEMENTATION AND VALIDATION OF CRUISE CONTROLLER". In: *Computer Science, Engineering And Applications (CCSEA)*. 2011, pp. 128–141. DOI: 10.5121/csit.2011.1214.

[Arm+02] Roy Armoni et al. "The ForSpec Temporal Logic: A New Temporal Property-Specification Language". In: *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by Joost-Pieter Katoen and Perdita Stevens. Vol. 2280. Lecture Notes in Computer Science. Springer, 2002, pp. 296–211. DOI: 10.1007/3-540-46002-0_21.

[BBL09] Franz Baader, Andreas Bauer, and Marcel Lippmann. "Runtime Verification Using a Temporal Description Logic". In: *Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings*. Ed. by Silvio Ghilardi and Roberto Sebastiani. Vol. 5749. Lecture Notes in Computer Science. Springer, 2009, pp. 149–164. DOI: 10.1007/978-3-642-04222-5_9.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BDR04] G. Barthe, P. R. D'Argenio, and T. Rezk. "Secure information flow by self-composition". In: *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* 2004, pp. 100–114. DOI: 10.1109/CSFW.2004.1310735.

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Relational Verification Using Product Programs". In: *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. Ed. by Michael J. Butler and Wolfram Schulte. Vol. 6664. Lecture Notes in Computer Science. Springer, 2011, pp. 200–214. DOI: 10.1007/978-3-642-21437-0_17.

[Bar+19] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. "Verifying Relational Properties using Trace Logic". In: *CoRR* abs/1906.09899 (2019). arXiv: 1906.09899. URL: http://arxiv.org/abs/1906.09899.

[Bar+18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. "Introduction to Runtime Verification". In: *Lectures on Runtime Verification - Introductory and Advanced Topics*. Ed. by Ezio Bartocci and Yliès Falcone. Vol. 10457. LNCS. Springer, 2018, pp. 1–33. DOI: 10.1007/978-3-319-75632-5_1.

[Bau+04a] Nanette Bauer, Sebastian Engell, Ralf Huuck, Sven Lohmann, Ben Lukoschus, Manuel Remelhe, and Olaf Stursberg. "Verification of PLC Programs Given as Sequential Function Charts". English. In: *Integration of Software Specification Techniques for Applications in Engineering*. LNCS 3147. Springer, 2004. DOI: 10.1007/978-3-540-27863-4_28.

[Bau+04b] Nanette Bauer, Ralf Huuck, Ben Lukoschus, and Sebastian Engell. "A Unifying Semantics for Sequential Function Charts". English. In: *Integration of Software Specification Techniques for Applications in Engineering*. LNCS 3147. Springer, 2004, pp. 400–418. DOI: 10.1007/978-3-540-27863-4_22.

[Bec+17] Bernhard Beckert, Suhyun Cha, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. "Generalised Test Tables: A Practical Specification Language for Reactive Systems". In: *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings*. Ed. by Nadia Polikar-

pova and Steve Schneider. Vol. 10510. Lecture Notes in Computer Science. Springer, 2017, pp. 129–144. DOI: 10.1007/978-3-319-66845-1_9.

[Bec+19] Bernhard Beckert, Jakob Mund, Mattias Ulbrich, and Alexander Weigl. "Formal Verification of Evolutionary Changes". In: *Managed Software Evolution*. Ed. by Ralf H. Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin. Springer, 2019, pp. 309–332. DOI: 10.1007/978-3-030-13499-0_11.

[Bec+15] Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. "Regression Verification for Programmable Logic Controller Software". In: *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*. Ed. by Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi. Vol. 9407. Lecture Notes in Computer Science. Springer, 2015, pp. 234–251. DOI: 10.1007/978-3-319-25423-4_15.

[BK11] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16.

[BBK12] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. "Arcade.PLC: A Verification Platform for Programmable Logic Controllers". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, 2012, pp. 338–341. DOI: 10.1145/2351676.2351741.

[Bit01] Friedemann Bitsch. "Safety Patterns - The Key to Formal Specification of Safety Requirements". In: *Computer Safety, Reliability and Security, 20th International Conference, SAFECOMP 2001, Budapest, Hungary, September 26-28, 2001, Proceedings*. Ed. by Udo Voges. Vol. 2187. Lecture Notes in Computer Science. Springer, 2001, pp. 176–189. DOI: 10.1007/3-540-45416-0_18.

[Bjø+15] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. "Horn Clause Solvers for Program Verification". In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte. Vol. 9300. Lecture Notes in Computer Science. Springer, 2015, pp. 24–51. DOI: 10.1007/978-3-319-23534-9_2.

[Bla+17] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, and Virgile Prevosto. "RPP: Automatic Proof of Relational Properties by Self-composition". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 391–397. DOI: 10.1007/978-3-662-54577-5_22.

[BB11] Jan Olaf Blech and Sidi Ould Biha. "Verification of PLC Properties Based on Formal Semantics in Coq". In: *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. Ed. by Gilles Barthe, Alberto Pardo, and Gerardo Schneider. Vol. 7041. Lecture Notes in Computer Science. Springer, 2011, pp. 58–73. DOI: 10.1007/978-3-642-24690-6_6.

[Blo+15] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. "Shield Synthesis:: Runtime Enforcement for Reactive Systems". en. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. LNCS. Springer, 2015, pp. 533–548. DOI: 10.1007/978-3-662-46681-0_51.

[BHK18] Dimitri Bohlender, Daniel Hamm, and Stefan Kowalewski. "Cycle-bounded model checking of PLC software via dynamic large-block encoding". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. Ed. by Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir. ACM, 2018, pp. 1891–1898. DOI: 10.1145/3167132.3167334.

[BK18] Dimitri Bohlender and Stefan Kowalewski. "Design and Verification of Restart-Robust Industrial Control Software". In: *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings*. Ed. by Carlo A. Furia and Kirsten Winter. Vol. 11023. Lecture Notes in Computer Science. Springer, 2018, pp. 47–68. DOI: 10.1007/978-3-319-98938-9_4.

[BK20] Dimitri Bohlender and Stefan Kowalewski. "Leveraging Horn clause solving for compositional verification of PLC software". In: *Discret. Event Dyn. Syst.* 30.1 (2020), pp. 1–24. DOI: 10.1007/s10626-019-00296-8.

[Böm+20] Thomas Bömer, Karl-Heinz Büllesbach, Michael Hauke, Stefan Otto, and Christian Werner. *IFA Report 1/2020, Praxisgerechte Umsetzung der Anforderungen für sicherheitsbezogene Embedded-Software nach DIN EN ISO 13849-1*. Tech. rep. Sankt Augustin: Institut für Arbeitsschutz der Deutschen Gesetzlichen Unfallversicherung (IFA), 2020.

[BM07] A. R. Bradley and Z. Manna. "Checking Safety by Inductive Generalization of Counterexamples to Induction". In: *Formal Methods in Computer Aided Design, 2007. FMCAD '07*. 2007, pp. 173–180. DOI: 10.1109/FAMCAD.2007.15.

[Bra11] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". English. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Vol. 6538. LNCS. Springer, 2011, pp. 70–87. DOI: 10.1007/978-3-642-18275-4_7.

[Bra12] Aaron R. Bradley. "Understanding IC3". In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Vol. 7317. Lecture Notes in Computer Science. Springer, 2012, pp. 1–14. DOI: 10.1007/978-3-642-31612-8_1.

[Bra+11] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. "An incremental approach to model checking progress properties". In: *International Conference on Formal Methods in Computer-*

Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011. Ed. by Per Bjesse and Anna Slobodová. FMCAD Inc., 2011, pp. 144–153. URL: http://dl.acm.org/citation.cfm?id=2157677.

[BL90] J. Richard Buchi and Lawrence H. Landweber. "Solving Sequential Conditions by Finite-State Strategies". In: *The Collected Works of J. Richard Büchi*. Ed. by Saunders Mac Lane and Dirk Siefkes. New York, NY: Springer New York, 1990, pp. 525–541. DOI: 10.1007/978-1-4613-8928-6_29.

[Büc90] J. Richard Büchi. "On a Decision Method in Restricted Second Order Arithmetic". In: *The Collected Works of J. Richard Büchi*. Ed. by Saunders Mac Lane and Dirk Siefkes. New York, NY: Springer New York, 1990, pp. 425–435. DOI: 10.1007/978-1-4613-8928-6_23.

[Bur+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. "Symbolic model checking: 10^{20} States and beyond". In: *Information and Computation* 98.2 (1992), pp. 142–170. URL: http://www.sciencedirect.com/science/article/pii/089054019290017A.

[Cav+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. "The nuXmv Symbolic Model Checker". In: *CAV 2014*. Vol. 8559. LNCS. Springer, 2014, pp. 334–342.

[Cha21] Suhyun Cha. "Application concept and evaluation of a formal specification approach usable by engineers for retrofitting production automation by software changes". submitted. PhD thesis. Technical University of Munich, 2021.

[Cha+19] Suhyun Cha, Mattias Ulbrich, Alexander Weigl, Bernhard Beckert, Kathrin Land, and Birgit Vogel-Heuser. "On the Preservation of the Trust by Regression Verification of PLC software for Cyber-Physical Systems of Systems". In: *17th IEEE International Conference on Industrial Informatics, INDIN 2019, Helsinki, Finland, July 22-25, 2019*. IEEE, 2019, pp. 413–418. DOI: 10.1109/INDIN41052.2019.8972210.

[Cha+17] Suhyun Cha, Sebastian Ulewicz, Birgit Vogel-Heuser, Alexander Weigl, Mattias Ulbrich, and Bernhard Beckert. "Generation of monitoring functions in production automation using test specifications". In: *15th IEEE International Conference on Industrial Informatics, INDIN 2017, Emden, Germany, July 24-26, 2017*. IEEE, 2017, pp. 339–344. DOI: 10.1109/INDIN.2017.8104795.

[Cha+] Suhyun Cha, Birgit Vogel-Heuser, Alexander Weigl, Mattias Ulbrich, and Bernhard Beckert. "Table-based formal specification approaches for control engineers – empirical studies of usability". In: *IET Cyber-Physical Systems: Theory & Applications* (). submitted.

[Cha+18a] Suhyun Cha, Alexander Weigl, Mattias Ulbrich, Bernhard Beckert, and Birgit Vogel-Heuser. "Achieving delta description of the control software for an automated production system evolution". In: *14th IEEE International Conference on Automation Science and Engineering, CASE 2018, Munich, Germany, August 20-24, 2018*. IEEE, 2018, pp. 1170–1176. DOI: 10.1109/COASE.2018.8560588.

[Cha+18b] Suhyun Cha, Alexander Weigl, Mattias Ulbrich, Bernhard Beckert, and Birgit Vogel-Heuser. "Applicability of generalized test tables: a case study using the manufacturing system demonstrator xPPU". In: *Automatisierungstechnik* 66.10 (2018), pp. 834–848. DOI: 10.1515/auto-2018-0028.

[Cha+16] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. "CoCoSpec: A Mode-Aware Contract Language for Reactive Systems". In: *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*. Ed. by Rocco De Nicola and eva Kühn. Vol. 9763. Lecture Notes in Computer Science. Springer, 2016, pp. 347–366. DOI: 10.1007/978-3-319-41591-8_24.

[CR05] Feng Chen and Grigore Roşu. "Java-MOP: A Monitoring Oriented Programming Environment for Java". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. LNCS. Springer, 2005, pp. 546–550. DOI: 10.1007/978-3-540-31980-1_36.

[CL02] Yoonsik Cheon and Gary T Leavens. "A Runtime Assertion Checker for the Java Modeling Language (JML)". en. In: *Software Engineering Research and Practice (SERP)*. 2002.

[CGH97] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. "Another Look at LTL Model Checking". In: *Formal Methods Syst. Des.* 10.1 (1997), pp. 47–71. DOI: 10.1023/A:1008615614281.

[Cla+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. "Temporal Logics for Hyperproperties". In: *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 2014, pp. 265–284. DOI: 10.1007/978-3-642-54792-8_15.

[CS08] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008*. 2008, pp. 51–65. DOI: 10.1109/CSF.2008.7.

[CS10] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210. DOI: 10.3233/JCS-2009-0393.

[Cre+99] Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. "How to Forget a Secret". In: *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings*. Ed. by Christoph Meinel and Sophie Tison. Vol. 1563. Lecture Notes in Computer Science. Springer, 1999, pp. 500–509. DOI: 10.1007/3-540-49116-3_47.

[DAn+05] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. "LOLA: Runtime Monitoring of Synchronous Systems". In: *Temporal Representation and Reasoning (TIME)*. IEEE, 2005, pp. 166–174. DOI: 10.1109/TIME.2005.26.

[Dan+16] Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta, and Sergio Mover. "Infinite-State Liveness-to-Safety via

Implicit Abstraction and Well-Founded Relations". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9779. Lecture Notes in Computer Science. Springer, 2016, pp. 271–291. DOI: 10.1007/978-3-319-41528-4_15.

[DBM15] Dániel Darvas, Enrique Blanco Viñuela, and István Majzik. "A formal specification method for PLC-based applications". In: *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*. 2015, pp. 907–910.

[DFB15] Dániel Darvas, Borja Fernández Adiego, and Enrique Blanco Viñuela. "PLCverif: A tool to verify PLC programs based on model checking techniques". In: *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*. Melbourne, Australia, 2015, pp. 911–914.

[DMV16a] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. "Conformance checking for programmable logic controller programs and specifications". In: *11th IEEE Symposium on Industrial Embedded Systems, SIES 2016, Krakow, Poland, May 23-25, 2016*. IEEE, 2016, pp. 29–36. DOI: 10.1109/SIES.2016.7509409.

[DMV16b] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. "Formal Verification of Safety PLC Based Control Software". In: *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*. Ed. by Erika Ábrahám and Marieke Huisman. Vol. 9681. Lecture Notes in Computer Science. Springer, 2016, pp. 508–522. DOI: 10.1007/978-3-319-33693-0_32.

[DVM16] Dániel Darvas, Enrique Blanco Viñuela, and István Majzik. "PLC code generation based on a formal specification language". In: *14th IEEE International Conference on Industrial Informatics, INDIN 2016, Poitiers, France, July 19-21, 2016*. IEEE, 2016, pp. 389–396. DOI: 10.1109/INDIN.2016.7819191.

[Den76] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5 (1976), pp. 236–243. DOI: 10.1145/360051.360056.

[Die+16] Sarah Diesburg, Christopher Meyers, Mark Stanovich, An-I Andy Wang, and Geoff Kuenning. "TrueErase: Leveraging an Auxiliary Data Path for Per-File Secure Deletion". In: *ACM Trans. Storage* 12.4 (2016). DOI: 10.1145/2854882.

[Dil+94] Laura K. Dillon, George Kutty, Louise E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. "A Graphical Interval Logic for Specifying Concurrent Systems". In: *ACM Trans. Softw. Eng. Methodol.* 3.2 (1994), pp. 131–165. DOI: 10.1145/192218.192226.

[Dim+12] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. "Model Checking Information Flow in Reactive Systems". In: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*. Ed. by Viktor Kuncak and Andrey Rybalchenko. Vol. 7148. Lecture Notes in Computer Science. Springer, 2012, pp. 169–185. DOI: 10.1007/978-3-642-27940-9_12.

[EMH18] Marco Eilers, Peter Müller, and Samuel Hitz. "Modular Product Programs". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 502–529. DOI: 10.1007/978-3-319-89884-1_18.

[Fer+14] Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Víctor M. González Suárez, and Jan Olaf Blech. "Modelling and Formal Verification of Timing Aspects in Large PLC Programs". In: *Proceedings of the 19th IFAC World Congress*. Cape Town, South Africa, 2014, pp. 3333–3339.

[FJ97] Konrad Feyerabend and Bernhard Josko. "A visual formalism for real time requirement specifications". In: *Transformation-Based Reactive Systems Development*. Ed. by Miquel Bertran and

Teodor Rus. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 156–168.

[Fin+19] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. "Monitoring hyperproperties". In: *Formal Methods in System Design* 54.3 (2019), pp. 336–363. DOI: 10.1007/s10703-019-00334-z.

[FK09] Bernd Finkbeiner and Lars Kuhtz. "Monitor Circuits for LTL with Bounded and Unbounded Future". In: *Runtime Verification (RV) 2009*. Ed. by Saddek Bensalem and Doron A. Peled. Vol. 5779. LNCS. Springer, 2009, pp. 60–75. DOI: 10.1007/978-3-642-04694-0_5.

[Fis99] Kathi Fisler. "Timing Diagrams: Formalization and Algorithmic Verification". In: *J. Log. Lang. Inf.* 8.3 (1999), pp. 323–361. DOI: 10.1023/A:1008345113376.

[GS13] Benny Godlin and Ofer Strichman. "Regression verification: proving the equivalence of similar programs". In: *Softw. Test. Verification Reliab.* 23.3 (2013), pp. 241–258. DOI: 10.1002/stvr.1472.

[Gor19] Mattias Gorenflo. "Semantic-Preserving Transformations of Sequential Function Charts into Efficient Structured Text". B.Sc Thesis. 2019.

[GGS21] Ohad Goudsmid, Orna Grumberg, and Sarai Sheinvald. "Compositional Model Checking for Multi-properties". In: *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*. Ed. by Fritz Henglein, Sharon Shoham, and Yakir Vizel. Vol. 12597. Lecture Notes in Computer Science. Springer, 2021, pp. 55–80. DOI: 10.1007/978-3-030-67067-2_4.

[Gri+12] Stephan Grimm, Michael Watzke, Thomas Hubauer, and Falco Cescolini. "Embedded $\mathcal{EL}^+$ Reasoning on Programmable Logic Controllers". In: *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II*. Ed. by Philippe Cudré-Mauroux et al. Vol. 7650. Lecture Notes in Computer Science. Springer, 2012, pp. 66–81. DOI: 10.1007/978-3-642-35173-0_5.

[Gur+15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. "The SeaHorn Verification Framework". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 343–361. DOI: 10.1007/978-3-319-21690-4_20.

[GST16] Ofer Guthmann, Ofer Strichman, and Anna Trostanetski. "Minimal unsatisfiable core extraction for SMT". In: *FMCAD 2016*. 2016, pp. 57–64. DOI: 10.1109/FMCAD.2016.7886661.

[Hal98] Nicolas Halbwachs. "Synchronous Programming of Reactive Systems". In: *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*. Ed. by Alan J. Hu and Moshe Y. Vardi. Vol. 1427. Lecture Notes in Computer Science. Springer, 1998, pp. 1–16. DOI: 10.1007/BFb0028726.

[HBS12] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. "Incremental, Inductive CTL Model Checking". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 532–547. DOI: 10.1007/978-3-642-31424-7_38.

[Hau+17] Michael Hauke et al. *IFA Report 2/2017, Funktionale Sicherheit von Maschinensteuerungen –Anwendung der DIN EN ISO 13849–*. Tech. rep. Sankt Augustin: Institut für Arbeitsschutz der Deutschen Gesetzlichen Unfallversicherung (IFA), 2017.

[HKL09] C. Heitmeyer, J. Kirby, and B. Labaw. "Tools for formal specification, verification, and validation of requirements". In: *Conference on Computer Assurance (COMPASS)*. IEEE, 2009, pp. 35–47. DOI: 10.1109/CMPASS.1997.613206.

[Hei+05] C. L. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. "Tools for constructing requirements specifications: The SCR toolset at the age of ten". In: *International Journal of Computer Systems Science and Engineering* 20.1 (2005), pp. 19–35.

[Hei+98] Constance L. Heitmeyer, James Kirby, Bruce G. Labaw, and Ramesh Bharadwaj. "SCR*: A Toolset for Specifying and Analyzing Software Requirements". In: *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*. Ed. by Alan J. Hu and Moshe Y. Vardi. Vol. 1427. Lecture Notes in Computer Science. Springer, 1998, pp. 526–531. DOI: 10.1007/BFb0028775.

[HJ07] L. Heitmeyer and R. D. Jeffords. "Applying a Formal Requirements Method to Three NASA Systems: Lessons Learned". In: *2007 IEEE Aerospace Conference*. 2007, pp. 1–10. DOI: 10.1109/AERO.2007.352764.

[HOW14] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. "Online Monitoring of Metric Temporal Logic". en. In: *Runtime Verification (RV)*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Vol. 8734. LNCS. Springer, 2014, pp. 178–192.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[HR18] Hossein Hojjat and Philipp Rümmer. "The ELDARICA Horn Solver". In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. Ed. by Nikolaj Bjørner and Arie Gurfinkel. IEEE, 2018, pp. 1–7. DOI: 10.23919/FMCAD.2018.8603013.

[IEC61131-3] International Standard. *IEC 61131-3: Programmable controllers – Part 3: Programming languages*. 2014.

[KTV12] B. Kormann, D. Tikhonov, and B. Vogel-Heuser. "Automated PLC Software Testing using adapted UML Sequence Diagrams". In: *14th IFAC Symposium of Information Control Problems in Manufacturing* (2012), pp. 1615–1621.

[Koy90] Ron Koymans. "Specifying Real-Time Properties with Metric Temporal Logic". en. In: *Real-Time Systems* 2.4 (1990), pp. 255–299. DOI: 10.1007/BF01995674.

[KT14] Daniel Kroening and Michael Tautschnig. "CBMC – C Bounded Model Checker". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 8413. LNCS. Springer, 2014, pp. 389–391.

[KR18] C. Kurz and F. Rieger. *Cyberwar – Die Gefahr aus dem Netz: Wer uns bedroht und wie wir uns wehren können*. C. Bertelsmann Verlag, 2018.

[Lah+13] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. "Differential assertion checking". In: *ESEC/FSE 2013*. ACM, 2013, pp. 345–355. DOI: 10.1145/2491411.2491452.

[Lam+99] S. Lampérière-Couffin, O. Rossi, J.-M. Roussel, and J.-J. Lesage. "Formal Validation of PLC programs: a survey". In: *ECC*. 1999.

[LPB15] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. "General LTL Specification Mining (T)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. Ed. by Myra B. Cohen, Lars Grunske, and Michael Whalen. IEEE Computer Society, 2015, pp. 81–92. DOI: 10.1109/ASE.2015.71.

[Len18] Daniel Lentzsch. "Modular Regression Verification for Programmable Logic Controller Software". M.Sc Thesis. 2018.

[Lju+10] O. Ljungkrantz, K. Åkesson, M. Fabian, and C. Yuan. "A formal specification language for PLC-based control logic". In: *2010 8th IEEE International Conference on Industrial Informatics*. 2010, pp. 1067–1072. DOI: 10.1109/INDIN.2010.5549591.

[Lju+12] Oscar Ljungkrantz, Knut Akesson, Chengyin Yuan, and Martin Fabian. "Towards Industrial Formal Specification of Programmable Safety Systems". In: *IEEE Transactions on Control Systems Technology* 20.6 (2012), pp. 1567–1574. DOI: 10.1109/tcst.2011.2169262.

[MN04a] Oded Maler and Dejan Nickovic. "LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)". In: *Software Engineering and Formal Methods (SEFM)*. Ed. by Yassine Lakhnech and Sergio Yovine. 2004, pp. 152–166. DOI: 10.1007/978-3-540-30206-3_12.

[MN04b] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". en. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems (FTRTFT)*. Ed. by Yassine Lakhnech and Sergio Yovine. Vol. 3253. LNCS. Springer, 2004, pp. 152–166. DOI: 10.1007/978-3-540-30206-3_12.

[MR07] Heiko Mantel and Alexander Reinhard. "Controlling the What and Where of Declassification in Language-Based Security". In: *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*. Ed. by Rocco De Nicola. Vol. 4421. Lecture Notes in Computer Science. Springer, 2007, pp. 141–156. DOI: 10.1007/978-3-540-71316-6_11.

[Mey92] Bertrand Meyer. "Applying "Design by Contract"". In: *IEEE Computer* 25.10 (1992), pp. 40–51. DOI: 10.1109/2.161279.

[Mos85] B. Moszkowski. "A Temporal Logic for Multilevel Reasoning about Hardware". In: *Computer* 18.2 (1985), pp. 10–19. DOI: 10.1109/MC.1985.1662795.

[Mül18] Dirk Müllmann. "Auswirkungen der Industrie 4.0 auf den Schutz von Betriebs- und Geschäftsgeheimnissen". In: *Wettbewerb in Recht und Praxis (WRP) 2018* 64.10 (2018), pp. 1177–1184.

[Mur15] Toby C. Murray. "Short Paper: On High-Assurance Information-Flow-Secure Programming Languages". In: *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2015, Prague, Czech Republic, July 4-10, 2015*. Ed. by Michael Clarkson and Limin Jia. ACM, 2015, pp. 43–48. DOI: 10.1145/2786558.2786561.

[Ova+16] Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Osman Ünver. "An overview of model checking practices on verification of PLC software". In: *Softw. Syst. Model.* 15.4 (2016), pp. 937–960. DOI: 10.1007/s10270-014-0448-7.

[Pak+16] Antti Pakonen, Cheng Pang, Igor Buzhinsky, and Valeriy Vyatkin. "User-friendly formal specification languages – conclusions drawn from industrial experience on model checking". In:

IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016). Vol. 2016-Novem. Berlin, Germany, 2016. DOI: 10.1109/ETFA.2016.7733717.

[PMI94] D. Lorge Parnas, J. Madey, and M. Iglewski. "Precise documentation of well-structured programs". In: *IEEE Transactions on Software Engineering* 20.12 (1994), pp. 948–976. DOI: 10.1109/32.368133.

[Pfr+16] Steffen Pfrang, Jörg Kippe, David Meier, and Christian Haas. "Design and Architecture of an Industrial IT Security Lab". In: *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer International Publishing, 2016, pp. 114–123. DOI: 10.1007/978-3-319-49580-4_11.

[PNW11] Lee Pike, Sebastian Niller, and Nis Wegmann. "Runtime Verification for Ultra-Critical Systems". In: *Runtime Verification (RV)*. Ed. by Sarfraz Khurshid and Koushik Sen. Vol. 7186. LNCS. Springer, 2011, pp. 310–324. DOI: 10.1007/978-3-642-29860-8_23.

[PLC16] PLCopen Promotional Committee Training. *Coding Guidelines*. Tech. rep. Version 1.0. PCLOpen, 2016.

[PLC18] PLCopen Promotional Committee Training. *PLCopen Software Construction Guidelines: Structuring with SFC: do's and don'ts*. Tech. rep. Version 0.99 (release for comments). PCLOpen, 2018.

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *Foundations of Computer Science (FOCS)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[Pre+18] Sorina-Nicoleta Predut, Florentin Ipate, Marian Gheorghe, and Felician Campean. "Formal Modelling of Cruise Control System Using Event-B and Rodin Platform". In: *High Performance Computing and Communications (HPCC)*. IEEE, 2018, pp. 1541–1546. DOI: 10.1109/HPCC/SmartCity/DSS.2018.00253.

[Rob65] John Alan Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *J. ACM* 12.1 (1965), pp. 23–41. DOI: 10.1145/321250.321253.

[Rös+14] S. Rösch, D. Tikhonov, D. Schütz, and B. Vogel-Heuser. "Model-Based Testing of PLC Software: Test of Plants' Reliability by Using Fault Injection on Component Level". In: *IFAC World Congress* (2014), pp. 3509–3515.

[RV17] Susanne Rösch and Birgit Vogel-Heuser. "A Light-Weight Fault Injection Approach to Test Automated Production System PLC Software in Industrial Practice". English. In: *Control Engineering Practice* 58.Complete (2017), pp. 12–23. DOI: 10.1016/j.conengprac.2016.09.012.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Global Value Numbers and Redundant Computations". In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. Ed. by Jeanne Ferrante and P. Mager. ACM Press, 1988, pp. 12–27. DOI: 10.1145/73560.73562.

[SM03] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". In: *IEEE J. Sel. Areas Commun.* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121.

[San09] Davide Sangiorgi. "On the origins of bisimulation and coinduction". In: *ACM Trans. Program. Lang. Syst.* 31.4 (2009), 15:1–15:41. DOI: 10.1145/1516507.1516510.

[SS14] Christoph Scheben and Peter H. Schmitt. "Efficient Self-composition for Weakest Precondition Calculi". In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 579–594. DOI: 10.1007/978-3-319-06410-9_39.

[SJW98] Rainer Schlör, Bernhard Josko, and Dieter Werth. "Using a visual formalism for design verification in industrial environments". In: *Services and Visualization Towards User-Friendly Design*. Ed. by Tiziana Margaria, Bernhard Steffen, Roland Rückert, and Joachim Posegga. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 208–221.

[Sim+15] Milivoj Simeonovski, Fabian Bendun, Muhammad Rizwan Asghar, Michael Backes, Ninja Marnau, and Peter Druschel. "Oblivion: Mitigating Privacy Leaks by Controlling the Discoverability of Online Information". In: *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*. Ed. by Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis. Vol. 9092. Lecture Notes in Computer Science. Springer, 2015, pp. 431–453. DOI: 10.1007/978-3-319-28166-7_21.

[Smi09] Geoffrey Smith. "On the Foundations of Quantitative Information Flow". In: *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Luca de Alfaro. Vol. 5504. Lecture Notes in Computer Science. Springer, 2009, pp. 288–302. DOI: 10.1007/978-3-642-00596-1_21.

[SB11] Fabio Somenzi and Aaron R Bradley. "IC3: Where Monolithic and Incremental Meet". In: *FMCAD*. 2011, pp. 3–8.

[Spi+17] Markus Spindler, Thomas Aicher, Birgit Vogel-Heuser, and Johannes Fottner. "Erstellung von Steuerungssoftware für automatisierte Materialflusssysteme per Drag & Drop". In: *Logistics Journal : Proceedings* 2017.10 (2017). DOI: 10.2195/lj_Proc_spindler_de_201710_01.

[IEC61508] International Standard. *EN 61508-3:2010: Functional safety of electrical/ electronic/programmable electronic safety-related systems*. 2010.

[TC211] TC2 Task Force Motion Control. *Technical Specification: Part 1 -Function blocks for motion control*. eng. Tech. rep. Version 2.0. 2011. 141 pp.

[Tec06] Technical Committee 5 – Safety Software. *Technical Specification – Part 1: Concepts and Function Blocks*. Tech. rep. Version 1.0. PLCOpen, 2006. URL: http://www.plcopen.org/pages/tc5_safety/downloads/plcopen_tc5_safety_v1_0.pdf.

[Thr10] Kleanthis Thramboulidis. "The 3+1 SysML View-Model in Model Integrated Mechatronics". In: *Journal of Software Engineering and Applications* 03.02 (2010), pp. 109–118. DOI: 10.4236/jsea.2010.32014.

[TGK17] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. "Modular Demand-Driven Analysis of Semantic Difference for Program Versions". In: *SAS 2017*. Ed. by Francesco Ranzato. Vol. 10422. LNCS. Springer, 2017, pp. 405–427. DOI: 10.1007/978-3-319-66706-5_20.

[Ule+16a] Sebastian Ulewicz, Mattias Ulbrich, Alexander Weigl, Michael Kirsten, Franziska Wiebe, Bernhard Beckert, and Birgit Vogel-Heuser. "A Verification-Supported Evolution Approach to Assist Software Application Engineers in Industrial Factory Automation". In: *IEEE International Symposium on Assembly and Manufacturing (ISAM 2016)*. IEEE, 2016, pp. 19–25. DOI: 10.1109/ISAM.2016.7750714.

[Ule+16b] Sebastian Ulewicz, Mattias Ulbrich, Alexander Weigl, Michael Kirsten, Franziska Wiebe, Bernhard Beckert, and Birgit Vogel-Heuser. "A Verification-Supported Evolution Approach to Assist Software Application Engineers in Industrial Factory Automation". In: *IEEE International Symposium on Assembly and Manufacturing (ISAM)*. Fort Worth, USA, 2016, pp. 19–25.

[VBS18] Birgit Vogel-Heuser, Safa Bougouffa, and Michael Sollfrank. *Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the extended Pick and Place Unit.* Tech. rep. Institute of Automation and Information Systems, Technische Universität München, 2018.

[Vog+15] Birgit Vogel-Heuser, Alexander Fay, Ina Schaefer, and Matthias Tichy. "Evolution of software in automated production systems: Challenges and research directions". In: *Journal of Systems and Software* 110 (2015), pp. 54–84. DOI: 10.1016/j.jss.2015.08.026.

[Vog+14] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. *Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit.* Tech. rep. Institute of Automation and Information Systems, Technische

Universität München, 2014. URL: https://mediatum.ub.tum.de/node?id=1208973.

[Vog+16] Birgit Vogel-Heuser, Susanne Rösch, Juliane Fischer, Thomas Simon, Sebastian Ulewicz, and Jens Folmer. "Fault Handling in PLC-Based Industry 4.0 Automated Production Systems as a Basis for Restart and Self-Configuration and Its Evaluation". In: *Journal of Software Engineering and Applications* 9.1 (2016), pp. 1–43. DOI: 10.4236/jsea.2016.91001.

[VH01] V. Vyatkin and H. M. Hanisch. "Application of visual specifications for verification of distributed controllers". In: *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace*. Vol. 1. 2001, 646–651 vol.1. DOI: 10.1109/ICSMC.2001.969925.

[Wan+09] Hai Wan, Gang Chen, Xiaoyu Song, and Ming Gu. "Formalization and Verification of PLC Timers in Coq". In: *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2009, Seattle, Washington, USA, July 20-24, 2009. Volume 1*. Ed. by Sheikh Iqbal Ahamed, Elisa Bertino, Carl K. Chang, Vladimir Getov, Lin Liu, Hua Ming, and Rajesh Subramanyan. IEEE Computer Society, 2009, pp. 315–323. DOI: 10.1109/COMPSAC.2009.49.

[Wan+13] Rui Wang, Yong Guan, Liming Luo, Xiaojuan Li, and Jie Zhang. "Component-Based Formal Modeling of PLC Systems". In: *J. Appl. Math.* 2013 (2013), 721624:1–721624:9. DOI: 10.1155/2013/721624.

[Wei19] Alexander Weigl. *Provably Forgetting of Information in Manufacturing Systems: Verification of the KASTEL Industry Demonstrator*. KIT, Fakultät der Informatik, 2019. DOI: 10.5445/IR/1000117803.

[Wei21] Alexander Weigl. *Companion Material for the PhD thesis "Formal Specification and Verification for Automated Production Systems"*. 2021. DOI: 10.5445/IR/1000139656.

[Wei+20] Alexander Weigl, Mattias Ulbrich, Suhyun Cha, Bernhard Beckert, and Birgit Vogel-Heuser. "Relational Test Tables: A Practical Specification Language for Evolution and Security". In: *FormaliSE@ICSE 2020: 8th International Conference on Formal*

Methods in Software Engineering, Seoul, Republic of Korea, July 13, 2020. ACM, 2020, pp. 77–86. DOI: 10.1145/3372020.3391566.

[WUL20] Alexander Weigl, Mattias Ulbrich, and Daniel Lentzsch. "Modular Regression Verification for Reactive Systems". In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12477. Lecture Notes in Computer Science. Springer, 2020, pp. 25–43. DOI: 10.1007/978-3-030-61470-6_3.

[Wei+21] Alexander Weigl, Mattias Ulbrich, Shmuel Tyszberowicz, and Jonas Klamroth. "Runtime Verification of Generalized Test Tables". In: *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings*. Ed. by Aaron Dutle, Mariano Moscato, and Laura Titolo. Lecture Notes in Computer Science. accepted. Springer, 2021.

[Wei+17] Alexander Weigl, Franziska Wiebe, Mattias Ulbrich, Sebastian Ulewicz, Suhyun Cha, Michael Kirsten, Bernhard Beckert, and Birgit Vogel-Heuser. "Generalized test tables: A powerful and intuitive specification language for reactive systems". In: *15th IEEE International Conference on Industrial Informatics, INDIN 2017, Emden, Germany, July 24-26, 2017*. IEEE, 2017, pp. 875–882. DOI: 10.1109/INDIN.2017.8104887.

[Wei15] Alexander Sebastian Weigl. "Regression Verification for Programmable Logic Controller Software". Master's thesis. Karlsruhe Institute of Technology, 2015.

[Wie20] Andreas Wieland. "Horn-basierte Verifikation mit Generalisierten Testtabellen". B.Sc Thesis. 2020.

[Xio+20] Jiawen Xiong, Gang Zhu, Yanhong Huang, and Jianqi Shi. "A User-Friendly Verification Approach for IEC 61131-3 PLC Programs". In: *Electronics* 9.4 (2020), p. 572. DOI: 10.3390/electronics9040572.

[YB18] Mark Yep and Sylvain Bechet. *Esterel Cruise Controller*. Website, https://github.com/ooksei/esterel-cruise-controller/. access: 2019-10-16. 2018.

[Yi+15] Jooyong Yi, Dawei Qi, Shin Hwei Tan, and Abhik Roychoudhury. "Software Change Contracts". In: *ACM Trans. Softw. Eng. Methodol.* 24.3 (2015), 18:1–18:43. DOI: 10.1145/2729973.

[YF03] M Bani Younis and Georg Frey. "Formalization of existing PLC programs: A survey". In: *CESA*. 2003.

[Zee+07] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin C. Rinard. "Runtime Checking for Program Verification". In: *Runtime Verification (RV)*. Ed. by Oleg Sokolsky and Serdar Tasiran. Vol. 4839. LNCS. Springer, 2007, pp. 202–213. DOI: 10.1007/978-3-540-77395-5_17.