Deductive Verification of Concurrent Programs and its Application to Secure Information Flow for Java

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Daniel Grahl geb. Bruns

aus Bergisch Gladbach

Tag der mündlichen Prüfung:	29. Oktober 2015
Erster Gutachter:	Prof. Dr. Bernhard Beckert Karlsruher Institut für Technologie
Zweite Gutachterin:	Prof. Dr. Marieke Huisman Universiteit Twente

This document has been typeset camera-ready by the author in 11 pt Latin Modern using pdfLaTeX with microtyping extensions. RIP Hermann Zapf.
Please print on recycled paper if necessary.

Preface

I am deeply grateful that my supervisor Prof. Dr. Bernhard Beckert and the second reviewer of this dissertation, Prof. Dr. Marieke Huisman, have encouraged and supported me to finally produce this piece of work. Recent case have shown that it is not obvious that that both reviewers put so much effort into critically reading a comprehensive dissertation work.

Based on their comments, this final version contains some changes in comparison to the originally submitted version. Overall, I optimized the layout such that the dissertation could fit into less pages. Lemma 5.4 contained two errors (missing negation), that canceled each other, but introduced incorrectness into its proof. Chapter 8 was thoroughly restructured. Most of the original material was taken from *The KeY Book* [Ahrendt et al., 2016], that was written back to back. *The KeY Book* does not cover concurrency; hence, the overlap was too large. In particular, the section on model fields has been dropped (formerly Sect. 8.4.3) and Sect. 8.2.3 on framing has been shortened. The new chapter structure is meant to put more emphasis to the central themes of this dissertation: concurrency and information flow. Furthermore, I have extended the discussion of related work on concurrent separation logic (Sect. 10.3.2) and the section on future work (Sect. 11.3), which now contains two more items that I mentioned in my defense.

For parts of this dissertation, preliminary results have been published before using peer-reviewed outlets or as technical reports. Preliminary versions of Chap. 4 have been published as [Beckert and Bruns, 2012b, 2013]. Chapters 5 and 6 contain previous results from [Bruns, 2014b, 2015a]; Sect. 6.5 is additionally based on [Beckert, Bruns, Klebanov, Scheben, Schmitt, and Ulbrich, 2013a,b, 2014]. Chapter 8 is based in part on [Huisman, Ahrendt, Bruns, and Hentschel, 2014] and on [Grahl, Bubel, Mostowski, Ulbrich, and Weiß, 2016] (which itself is based on the thesis by Weiß [2011]) as well as short fragments from [Bruns, Mostowski, and Ulbrich, 2015b] and the yet unpublished [Grahl and Ulbrich, 2016]. Chapter 9 contains results from [Bruns, 2014a]. The production of all the text fragments that have been reused has mainly been an effort of myself. I would like to thank all coauthors, who made it possible for these papers to appear in the first place and who provided their consent for the contents to be reused in this dissertation.

I would like to thank all collegues and former collegues at Karlsruhe Institute of Technology (KIT), in the KeY project, and in the RS^3 program for the scientific coöperation, their moral support, or lunch time entertainment. I am particularily indebted to Assoc. Prof. Dr. Wolfgang Ahrendt for the fruitful discussions on concurrent program semantics, to Dr. Thorsten Bormer for his practical hints on the actual graduation process, to Simon Greiner for his extensive comments on earlier drafts, to Michael Kirsten for his contribution to the implementation described in Chap. 7, and to Prof. Dr. Ralf H. Reussner and Prof. Dr. Gregor Snelting, who happily agreed to

act as additional examiners in my defense. Additional proof-reading was thankfully provided by Dr. Erik Burger, Tina Grahl, Katharina Sanzillo, and Alexander Weigl. I would like to thank Audrey Bohlinger for her support in all formalities. I further appreciate the effort that all the anonymous reviewers have put into improving the aforementioned papers.

I acknowledge financial support by the Deutsche Forschungsgemeinschaft (DFG, German National Science Foundation) under project "Program-level Specification and Deductive Verification of Security Properties (DeduSec)"[1] within priority program 1496 "Reliable Secure Software Systems (RS^3)," by the Bundesministerium für Bildung und Forschung (BMBF, Federal Ministry of Education and Research) under project "Formal Information-Flow Analysis of Component-Based Systems (FIfAKS)"[2] as part of Software Campus, as well as by the Klaus Tschira Stiftung.

Karlsruhe, November 2015 *Daniel Grahl*

[1] `http://key-project.org/DeduSec/`
[2] `http://formal.iti.kit.edu/projects/FIfAKS/`

Abstract

The present dissertation has been produced in the context of the DFG priority programme "Reliably Secure Software Systems" (RS^3). The goal is to develop a rigorous analysis technique for proving functional and relational correctness of shared-memory multi-threaded object-oriented software systems—such as programs written in the Java language—based on formal program semantics and deductive verification. This work contains the first approach towards a semantically precise information flow anlaysis for concurrent programs.

Verification of concurrent programs still poses one of the major challenges in computer science. Several techniques to tackle this problem have been proposed. However, they often are not amenable to apply to components of open systems. Our approach is based on an adaptation of the modular rely/guarantee methodology in dynamic logic. Rely/guarantee uses functional specification to symbolically describe the observable behavior of concurrently running threads: while each thread *guarantees* adherence to a specified property at any point in time, all other threads can *rely* on this property being established. This allows to regard threads largely in isolation—only w.r.t. an symbolic thread environment that is constrained by these specifications.

In order to keep specifications modular and concise, we additionally need to ensure that 'nothing else changes.' In sequential programs, this *frame problem* has already been studied extensively. Our approach complements functional rely/guarantee specification with respective frame clauses. Our approach is completely modular in the sense that we prove correctness of threads w.r.t. any thread state, not assuming an initial state in which the environment is empty. Our framework of proof obligations does not include postconditions (i.e., functional correctness properties) directly. This decision is rooted in our effort to separate rely/guarantee, i.e., proof of well-behaved thread interactions, from other program properties. This permits to apply our approach to properties *beyond* postconditions, such as information flow security.

The feature dimensions of high-level programing languages are largely orthogonal. Many problems regarding object-orientation or the general richness in features of Java have been sufficiently solved before. This allows us in this thesis to focus on the issue of concurrency. We will explain our approach using a simple, but concurrent programing language. Besides the usual constructs for sequential programs, it caters for dynamic thread creation. In order to define it as a *conservative* extension of a previously defined sequential language, we make interleaving points explicit in the code, effectively disentangling 'own' and environment changes. We define interleaving semantics of concurrent programs parametric w.r.t. an underspecified deterministic scheduling function. Deterministic program semantics yields tractable definitions, while underspecification allows to abstract from concrete behavior.

To formally reason about programs of this language, we introduce a novel multi-modal logic, *Concurrent Dynamic Trace Logic* (CDTL). It combines the strengths of dynamic logic with those of linear temporal logic and allows to express temporal properties about symbolic program traces. We first develop a sound and relatively complete sequent calculus for the logic subset that uses the sequential part of the language, based on symbolic execution and temporal unwinding. In a second step, we extend this to a calculus for the complete logic by adding symbolic execution rules for concurrent interleavings and dynamic thread creation, based on the rely/guarantee methodology. Again, this calculus is proven sound and complete; it is suitable for automated deduction.

Besides *functional* properties, that assure safety of the system, we also investigate software security in the form of confidentiality—a *relational* property, i.e., a property relating multiple possible executions. The goal is to ensure that no data must flow from confidential sources to public sinks. *Language based information flow security* covers the scenario where information is handled by software which operational behavior is known to potential attackers. It is assumed that an attacker is aware of all vulnerabilities and how to exploit them. The analysis therefore focuses on the program source alone. We use the semantically defined security notion of *noninterference*—including its extensions to concurrent programs, such as LSOD and SSOD, as well as semantical declassification.

Unlike traditional static or dynamic software assurance techniques, analyses based on formal logics which precisely capture program semantics readily possess the aptitude to faithfully express semantically defined relational properties without false positives. Based on previous results, we develop a novel information flow security property appropriate for object-oriented and multi-threaded programs. This includes the absence of certain timing leaks, that are not considered in LSOD/SSOD.

Due to the expressivity of our logic CDTL, we can precisely formalize a wide range of security properties. Having a sound and complete calculus for CDTL enables us to precisely classify programs as either secure or insecure.

While the aforementioned formal definitions are given for a toy language, the ultimate goal is to reason about programs written in the Java language. The particular challenges lie in its object-orientation and support for multi-threading. We discuss how our verification approach extends naturally to multi-threaded Java and present an implementation of the rely/guarantee approach in KeY. KeY is a mature software verification system for (sequential) Java, based on dynamic logic theorem proving and contracts in the Design by Contract philosophy. It is being co-developed by the author and has been applied to several realistic verification targets. Our implementation benefits from the generality of the KeY approach and the adaptability of the KeY platform.

Specification is an essential part of modular software verification, both sequential and parallel. Given appropriate specification, 'auto-active' verification works without any interaction between user and prover. Instead, specification provides a convenient high-level interface to proofs. State of the art specification languages—such as the established Java Modeling Language (JML)—are rich in constructs for specifying functional behavior of sequential programs. We discuss its effectiveness in the specification of modularity properties and we propose natural extensions to JML regarding both confidentiality properties and multi-threaded programs.

Rely/guarantee based approaches often suffer from a considerable specification overhead; we lift this burden by integrating the specifications of rely/guarantee into JML. In this way, we develop a specification language that includes features for both sequential (method contracts) and parallel modularity (rely/guarantee).

We will demonstrate the effectiveness of the techniques presented in this dissertation in a case study. The target is a simple, but distributed, electronic voting system implemented in Java, that has been developed together with the group of R. Küsters at the University of Trier. Voting demands high assurances regarding information security, e.g., individual votes are confidential. At the same time, it requires some information, i.e., the election result, to be publicly available. This makes voting a prime target for the techniques developed in this work. We present a functional verification and an information flow analysis of a sequential implementation of the system.

Zusammenfassung

Die vorliegende Dissertation ist im Rahmen des DFG-Schwerpunktprogrammes ›Zuverlässig sichere Softwaresysteme‹ entstanden. Das Ziel ist die Entwicklung einer auf formaler Programmsemantik und deduktiver Verifikation basierenden durchgreifenden Technik zum Nachweis funktionaler und relationaler Korrektheit mehrfädriger und objektorientierter Softwaresysteme – wie etwa Programme der Sprache Java. Diese Arbeit beinhaltet den ersten Ansatz einer semantisch präzisen Informationsflussanalyse für nebenläufige Programme.

Die Verifikation nebenläufiger Programme stellt noch immer eine der großen Herausforderungen der Informatik dar. Einige Techniken zur Lösung dieses Problems sind bereits bekannt. Allerdings sind diese oft nicht geeignet, Komponenten offener Systeme zu behandeln. Der hier vorgestellte Ansatz basiert auf einer Adaption der modularen *Annahme/Zusicherungs-Methode* (engl. *rely/guarantee*, RG) in dynamischer Logik. Die RG-Methode benutzt funktionale Spezifikation, um das beobachtbare Verhalten parallel laufender Ausführungsfäden symbolisch zu beschreiben: Solange jeder Faden die Einhaltung einer gewissen Eigenschaft zu jedem Zeitpunkt zusichert, können alle anderen Fäden diese Eigenschaft annehmen. Dies erlaubt, Fäden weitgehend isoliert – lediglich bezüglich einer symbolischen Umgebung, die durch diese Spezifikation eingeschränkt ist – zu betrachten.

Um Spezifikationen modular und kurz zu halten, muss zusätzlich sichergestellt werden, dass keine weiteren Änderungen erfolgen. Für sequentielle Programme wurde dieses *Rahmenproblem* (engl. *frame problem*) bereits eingehend studiert. Der vorliegende Ansatz komplementiert die funktionale RG-Spezifikation mit entsprechenden Rahmenbedingungen. Er ist vollends modular: Die Korrektheit wird von Fäden bezüglich jeglichen Ausführungszustandes nachgewiesen. Das Beweisverpflichtungsrahmenwerk umfasst unmittelbar keine Nachbedingungen (d.h. funktionale Korrektheitseigenschaften).

Diese Entscheidung gründet auf dem Wunsch, den Nachweis wohlverträglicher Interaktion zwischen Fäden von sonstigen Programmeigenschaften zu separieren. Dies erlaubt, den Ansatz auf darüber hinaus gehende Eigenschaften, wie etwa Informationsflusssicherheit, zu erweitern.

Die Charakteristika von Programmierhochsprachen liegen weitestgehend orthogonal zueinander. Viele Fragen hinsichtlich Objektorientierung wurden bereits hinreichend beantwortet. Daher konzentriert sich diese Dissertation auf das Thema der Nebenläufigkeit. Zur Erläuterung des Ansatz wird eine simple, jedoch nebenläufige, Programmiersprache herangezogen. Neben den für sequentielle Programme üblichen Konstrukten umfasst sie auch das dynamische Abzweigen neuer Fäden. Um sie als eine *konservative* Erweiterung einer zuvor eingeführten sequentiellen Sprache zu definieren, werden Spreizungen des Programmflusses (engl. *interleaving*) explizit gemacht. Dadurch werden ›eigene‹ Schreibzugriffe von denen der Umgebung getrennt betrachtet. Die Spreizungssemantik nebenläufiger Programme ist parametrisch bezüglich einer unterspezifizierten, deterministischen Ablaufplanungsfunktion definiert.

Um logische Schlüsse über Programme dieser Sprache zu ermöglichen, führen wir eine neue Multimodallogik, *Concurrent Dynamic Trace Logic* (CDTL), ein. Sie vereint die Stärken dynamischer und linearer Temporallogik miteinander und vermag es daher, temporale Eigenschaften symbolischer Programmabläufe auszudrücken. Wir entwickeln zunächst – basierend auf symbolischer Ausführung und dem Abwickeln temporaler Operatoren – einen korrekten und relativ vollständigen Sequenzenkalkül für die Untermenge der Logik, welche den sequentiellen Teil der Sprache betrachtet. In einem zweiten Schritt erweitern wir diesen zu einem Kalkül für die komplette Logik durch Hinzufügen von Schlussregeln für Spreizungen und Abzweigungen basierend auf der RG-Methode. Für diesen Kalkül zeigen wir wiederum Korrektheit und Vollständigkeit; er ist geeignet zum automatisierten Schließen.

Neben funktionalen Eigenschaften – die die Funktionssicherheit (engl. *safety*) sicherstellen – untersuchen wir auch die Angriffssicherheit (engl. *security*) als eine relationale Eigenschaft, d.h. eine Eigenschaft, die mehrere mögliche Programmläufe in Relation setzt. Das Ziel ist, sicher zu stellen, dass keine Daten von vertraulichen Quellen zu öffentlichen Senken fließen. Sprachbasierte Informationsflusssicherheit betrachtet das Szenario, in welchem Information von Softwaresystemen, deren operationales Verhalten potentiellen Angreifern bekannt ist, behandelt wird. Es ist zu erwarten, dass Angreifer alle Schwachstellen und ihre Nutzung kennen. Die Analyse muss daher auf das Programm als solches konzentriert sein. Wir benutzen den semantisch definierten Begriff der *Nichtinterferenz* (engl. *noninterference*) – einschließlich seiner Erweiterungen bezüglich nebenläufigen Programmen, sowie semantische Deklassifikation.

Anders als traditionelle statische oder dynamische Methoden, sind auf formalen Logiken beruhende Analysen geeignet, semantisch definierte relationale Eigenschaften getreu auszudrücken. Ausgehend von vorhergehenden

Resultaten entwickeln wir eine neuartige Informationsflusssicherheitseigenschaft, die für objektorientierte und mehrfädrige Programme geeignet ist. Diese umfasst die Abwesenheit gewisser Terminierungslecks.

Dank der Ausdrucksmächtigkeit unserer Logik CDTL kann eine Bandbreite an Sicherheitseigenschaften präzise formalisiert werden. Der korrekte und vollständige Kalkül für CDTL ermöglicht sodann die zuverlässige Klassifikation von Programmen in ›sicher‹ und ›unsicher‹.

Während die o.g. formalen Definitionen sich nur auf eine Spielsprache beziehen, bleibt das übergeordnete Ziel, Schlüsse über Java-Programme zu ziehen. Die besonderen Herausforderungen liegen in der Objektorientierung und der Unterstützung für Mehrfädrigkeit. Wir erörtern, wie sich unser Ansatz natürlich auf nebenläufiges Java erweitern lässt, und präsentieren eine Implementierung von RG in KeY. KeY ist ein etabliertes Softwareverifikationssystem für (sequentielles) Java und basiert auf einem Theorembeweiser für dynamische Logik sowie auf Methodenverträgen. Es wird u.a. vom Autor entwickelt und wurde bereits in einschlägigen Fallstudien eingesetzt. Die Implementierung profitiert von der Generalität des KeY-Ansatzes und der Modifizierbarkeit der KeY-Plattform.

Spezifikationen sind essentieller Bestandteil modularer Softwareverifikation, im sequentiellen wie im parallelen Bereich. Ist eine passende Spezifikation gebeben – können ›autoaktive‹ Werkzeuge ohne weitere Interaktion ein Ergebnis liefern. Stattdessen bietet Spezifikation eine komfortable Schnittstelle zum Beweis auf höherer Ebene. Spezifikationssprachen – wie die etablierte Java Modeling Language (JML) – bieten vielfältige Konstrukte zur Spezifikation funktionalen Verhaltens sequentieller Programme. Deren Effektivität in Bezug auf Modularitätseigenschaften wird erörtert. Natürliche Erweiterungen für Vertraulichkeitseigenschaften und nebenläufige Programme werden vorgestellt.

Ansätze, die auf RG basieren, werden oft durch einen erheblichen Spezifikationsaufwand geschwächt; dieses Problem kann durch eine Integration in JML behoben werden. Auf diesem Weg entwickeln wir eine Spezifikationssprache, die Elemente sowohl für sequentielle Modularität (Kontrakte) als auch für parallele Modularität (RG) bereit hält.

Die Effektivität der in dieser Dissertation vorgestellten Techniken wird an einer Fallstudie demonstriert. Das Ziel ist ein einfaches, aber verteiltes, in Java implementiertes, elektronisches Wahlsystem, das zusammen mit der Gruppe von Ralf Küsters von der Universität Trier entwickelt wurde. Wahlen stellen hohe Anforderungen an die Informationssicherheit, z.B. die vertrauliche Verarbeitung der abgegebenen Stimmen. Gleichzeitig müssen bestimmte Informationen, z.B. das Wahlergebnis, öffentlich gemacht werden. Diese Herausforderung macht Wahlsysteme zu erstklassigen Zielen der hier entwickelten Techniken. Wir führen den Nachweis funktionaler Korrektheit und Angriffssicherheit einer sequentiellen Implementierung des Systems vor.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

At the time when the "right to privacy" was first proclaimed by Warren and Brandeis in 1890, there was little confidential information that was collected systematically. Today, in the 21st century, software systems are behind almost any process of everyday life—business or private. Not only do they manifest in 'traditional' personal computers, but also in mobile phones, electricity meters, health insurance cards, etc. Many software services are not provided by a local machine, but online in 'the cloud.' Typically, these services are accessed in a web browser that runs scripts or even Java programs. The Common Vulnerabilities and Exposures (CVE) website lists over 300 known vulnerabilities in the Java implementations that were discovered alone within the past 3 years [CVE].

With the ever growing amount of sensitive data which software systems handle, and the connectedness of the world, we are in need of precise and enforcible security mechanisms. The recent discoveries of security threats induced by badly designed software, such as the Heartbleed bug [Carvalho et al., 2014], have demonstrated the demand for a rigorous security assessment in software development. This does not only concern domains with traditionally high assurance demands, such as banking or aeronautics, but also private communication. Modern software is highly adaptable; bugs are too: a simple programing error could be replicated on a billion devices.

At the same time, software systems have become more and more complex. Established concepts for modularization or information hiding, such as object-orientation, facilitate the designers'/implementors' work; but raise the complexity of analyzing such a system. It has already been several years since *concurrency* had become a major paradigm in computer development. Although the software development generally lags behind the hardware, all modern systems are concurrent in some way or another. It is evident that concurrent systems are much harder to comprehend than sequential ones. Many software faults are linked to an inferior understanding of concurrent

program semantics. It has thus become vital for analyses of safety or security to take a leap forward from sequential to concurrent programs.

The "goto fail" bug in Apple's iOS operating system was an infamous programing error [Bland, 2014], that made the system vulnerable to attacks. Through an inadvertently introduced superfluous `goto` statement, the final step in the Transport Layer Security (TLS) protocol was never executed. This made possible person-in-the-middle attacks in seemingly secure TLS connections. We observe that 'small' programing errors, like Heartbleed or "goto fail," have an enormous impact on system security. Even though this kind of programing error would have been very much avoidable through traditional quality assurance techniques—like thorough code review—it remained undiscovered for over one year and left several millions of devices vulnerable during that period.

With today's ubiquituous usage of software systems, it has even become a question of *ethics* to produce software quality [Class, 2008]. For instance, electronic voting can only be established in a democratic society if even lay citizens can *trust* the system. Proving safety and security in a rigorous way is a necessary step towards this (cf. [Ewert et al., 2003; Deutscher Bundestag]).

1.1 Sequential and Concurrent Programs

Concurrent computer systems have already existed for a long time. But since the end of Moore's law has been reached (i.e., the speed of single processors does not evolve significantly anymore) in the past decade, concurrency has become widespread—even in end-user systems. This development further stresses the demand for high-precision analysis. While state of the art techniques for formal specification and verification of *sequential* programs have developed and matured in the past years, similar concepts that would be appropriate for concurrent programs are still in their infancy.

Sequential programs run on single processors. In contrast, concurrent programs can typically be modeled as a collection of processes—that may each be described as sequential programs—and some (implicit or explicit) communication channels. The central advantage of concurrent programs is that they can be actually executed in parallel on physically separate processors. This concurrency model is usually found in distributed systems, where each processor maintains its own memory and cannot interfere directly with the other processes. Instead, they communicate (synchronously or asynchronously) through message passing using explicit channels.

Time share parallelism, on the other hand, runs on (one or more) shared processors,[1] with the next to-be-dispatched process to be determined by a scheduler. Processes may be *preëmpted* in order for other processes to be

[1] The exact number of physical processors is not relevant to the design and analysis of multi-threaded programs.

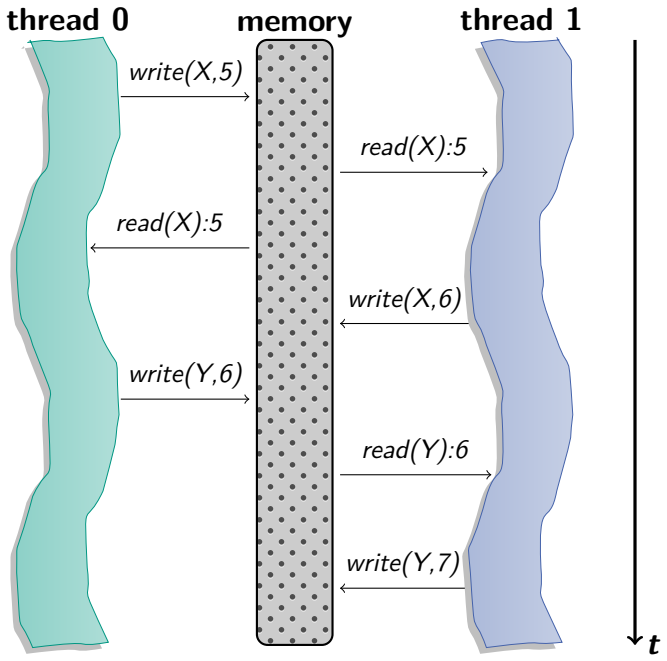


Figure 1.1: Example of two threads naïvely accessing shared memory (center), thus interfering with each other. Thread 0 executes a program `X = 5; Y = X;` and thread 1 executes a program `X++; Y++;` The vertical axis represents time. With the interleaved read and write operations displayed here, neither of both achieves the intention expressed in their respective sequential program. Note that this is only one of many possible interleavings.

executed in between. In most systems this interleaving can occur at any point in time during the execution. In some systems the program states in which interleaving is possible are restricted. An example is the coöperative scheduling paradigm using explicit release points [Dovland et al., 2005].

Typically, time share systems also share memory. While modern desktop computers have multiple processor cores, these are not (purely) distributed, but form a time share system that particularly shares the main memory.[2] Processes that share processors and memory are also called threads, designating this kind of concurrency as *multi-threading*. This thesis is dedicated to the multi-threading paradigm as it is used in the Java language [Arnold and Gosling, 1998; Gosling et al., 2014], amongst others (cf. [Philippsen, 2000]).

Interference

Processes in shared memory systems tend to interfere with each other: one process writes a location that another one is about to read. This means that

[2]Each processor may have private memory in caches, etc., though.

the functional behavior of one process may be influenced by another one (and the scheduler itself). As a consequence, these kind of processes cannot be assessed independently. See Fig. 1.1 on the preceding page for an intuitive visual example of two threads naïvely accessing the same memory locations at the same time. While some interactions are certainly benign—otherwise there would be little benefit in concurrency—their scope cannot be restricted in general. Thus, the goal is to harness concurrent modifications.

Proving Correctness of Concurrent Programs

Computing all possible interleavings for concurrent systems is far from being feasible. This is known as the *global method* [de Roever et al., 2001]. Its complexity is clearly exponential in the number of concurrent processes (with the number of local decision points in each process being the base of that power). The technique by Owicki and Gries [1976] was the first to consider interleavings symbolically. Its complexity is only linear in the number of processes. It is based on traditional local correctness proofs in Hoare [1969] logic plus additional noninterference proofs. However, it requires a high annotation overhead. And, most importantly, we need to verify all remote threads to establish correctness of 'our' thread. In the case that an additional thread is created, all proof results need to be reestablished. Thus the Owicki and Gries technique is not compositional in the sense that, from the correctness of individual modules, overall correctness of the combined system can be derived.

The compositional *rely/guarantee* technique [Jones, 1983; Xu et al., 1997] completely relies on functional specification. This allows to regard the thread under investigation ('our' thread) largely in isolation. The approach considers any environment that adheres to the given specifications: Each thread is assigned two-state invariants *rely* (describing environment behavior that we can rely on) and *guar* (the guarantees we provide to the environment). While it remains to be proven that the rely conditions are actually guaranteed, assessing the correctness of a thread does not require insight into the internals of other threads.

Modularity is another important meta-level property of analysis techniques. Modularity means that analysis of a module can be based on the module itself in isolation—without a concrete representation of its environment. This allows to adapt modules to other environments without losing previously established guarantees. The classical rely/guarantee approach is not modular since it considers programs that are closed under parallel composition. For thread-based systems, it is more appropriate to apply a verification technique that considers *open* programs.

1.2 Information Flow Security

A particular concern of software quality assurance is secure information flow. In general, information can flow both ways between a system to be protected and external agents. Information flow security can thus be divided into *integrity* and *confidentiality*. Integrity is concerned with preventing unauthorized agents to manipulate the system. Conversely, confidentiality is concerned with restricting unqualified access to confidential information. More abstractly, the goal is to ensure that no data must flow from confidential (i.e., high security level) sources to public (i.e., low security level) sinks. Analyses associated with language-based information flow security [Sabelfeld and Myers, 2003a] covers the scenario where information is handled by software whose operational behavior is known to potential attackers. It is assumed that an attacker knows all vulnerabilities in programs and how to exploit them. However, attackers cannot break or circumvent the basic mechanisms provided by the language. The analysis therefore focuses on the program source (the 'language') alone.

A well-known confidentiality property is *noninterference* [Cohen, 1977], which is appropriate for sequential programs. Noninterference can express precisely the information that may legally flow. It provides security against any attacker that provides inputs, listens to primary output channels, and possesses unlimited deductive powers. As noninterference is an expressive, semantically defined property, traditional analyses based on type systems [Volpano and Smith, 1997] do not provide complete reasoning.

Formal Analysis of Secure Information Flow

Recently, theorem proving approaches to language based information flow analysis have gained prominence. These are based on a semantical notion of information flow and therefore bear the advantage of semantical precision over established static techniques like type checking. Some program logics such as dynamic logic [Harel, 1979; Beckert, 2001] are readily able to express relational properties, i.e., properties that relate multiple program executions, like noninterference. And at the same time, formal verification of functional properties about software has made great progress in the past years. In particular, the KeY prover [Beckert et al., 2007a], co-developed by the author, for first-order dynamic logic is able to formally verify information flow properties about *sequential* Java programs. One central aim of this dissertation is to make first steps towards lifting these techniques to reasoning about *concurrent* programs.

In concurrent programs, scheduling may depend on unknown parts of the system state. We assume an attacker model where the attacker is in control of threads, but not the scheduler. This means that an attacker cannot

distinguish why/in which state its threads are scheduled or not—even in case the scheduler computes schedules using confidential information. As Cohen's original definition of noninterference only applies to deterministic sequential and terminating programs, several extensions have been proposed. Low-security observational determinism (LSOD) [McLean, 1992] is a well-known extension for concurrent programs, where schedulers are considered nondeterministic. It requires that each public output is computed in an *observably* deterministic way, thus independent of scheduler indeterminism. However, Giffhorn and Snelting [2015] note that "several attempts to devise program analysis algorithms for LSOD turned out to be unsound, unprecise [sic!], or very restrictive." Furthermore, LSOD both leaves the scheduler out of the picture and thus rejects programs that would be secure under specific (deterministic) schedulers, and at the same time, it does not report timing leaks that are induced by the relative order of memory updates.

Declassification

Confidential information may be declassified, i.e., intentionally released. Typically, this only refers to *parts* of the confidential information. For instance, a password checker is expected to release the information whether the entered string is actually the secret password or not. While this obviously reveals partial information about the secret, this very construction enables security on a higher level (viz. access control). Many approaches to secure information flow ignore the challenge to precisely state *what* information is released [Zdancewic, 2004], but only *where* in the program. The information to be released may depend on secrets in a nontrivial way: e.g., in an election, the public result is the sum of votes on secret ballots; precise information flow analysis thus needs support for reasoning about sum comprehensions. Since theorem proving approaches are founded semantically, precise subject declassification (i.e., which information is released) already comes for free.

We additionally consider *timing* of declassification. Just like subject declassification can be expressed as a relational property between states, temporal declassification can be expressed as a relation between traces. Controlling the temporal dimension of declassification is essential in state based software systems. Consider, for instance, an electronic voting system, that has different declassification policies before and after the election has been closed: only afterwards the result (i.e., the sum of votes) may be published.

1.3 Formal Verification for Safety and Security

Even though a disaster like "goto fail," that was mentioned earlier, would have been avoidable easily (e.g., with code reviews or style checking), traditional assurance techniques for safety and security do only provide weak assurances. Many of the most infamous bugs were overlooked even by experts, demanding an even more thorough and rigorous analysis.

Why Target Software?

Software is an integral component of any cyberphysical system. It is reasonable to target certain modules separately from other software or hardware components. Portability of software renders development more cost-effective, but can introduce bugs more easily. An infamous example from the real world was the maiden flight of the Ariane 5 rocket, where faulty conversion of 64-bit floating point data to 16-bit integers finally caused the spacecraft to self-destruct, just seconds after lift-off [Dowson, 1997; Nuseibeh, 1997]. Undoubtedly, dealing with finite numerical data types is a very common source of programing errors. It is thus desirable to detect such errors early in the development process and not after the system (i.e., a spacecraft in this case) has been finally assembled. The costs for a late fix may be higher by several orders of magnitude; a complete system failure may even cost billions of Euros. In addition, the affected component actually *had been* tested when deployed on the predecessor Ariane 4, but these test depend on particular physical constraints. This example shows that a robust software analysis must be based on software in isolation from physical environments.

To security, formal proofs of software correctness do matter—at the very least as much as they do to safety. In many safety-critical environments, we can rely on developers and users being professional by making the 'right' decisions and that products follow established engineering best practices. It is thus tolerable to restrict a formal safety analysis to 'sane' cases, in order to keep it simple. However, for security, we are forced to assume the most capable attackers to do their worst. They may exploit even the tiniest vulnerability. An effective analysis must cater even for those issues that are hard to find. "Heartbleed, like most security vulnerabilities, only manifests with incorrect or unexpected input." [Wheeler, 2014]

It is known that several catastrophic safety violations were caused by buggy software, but the effects of security vulnerabilities is largely unknown. A particular danger lies with *zero-day exploits* [Bilge and Dumitras, 2012]. These are made possible by vulnerabilities that are freshly discovered, but not yet known to the public. Zero-day exploits grant attackers a head start before countermeasures can be mounted.

Why Use Verification?

Formal verification based on semantics and logics is a heavy-weight approach to software analysis, that may be resource-consuming. However, verification is indispensable for high assurance. It is well-known that dynamic software analyses—such as testing, debugging, or assertion checking—can only be used to find existing bugs, but not show their absence. As mentioned above, traditional static analysis techniques like code review can provide a baseline confidence in correctness.[3] Yet, manual inspection only scratches the surface. More involved bugs may hide deep within the dark corners of programing language semantics. For instance, Wheeler [2014] argues that the Heartbleed bug could not be found by neither dynamic or static analyses that are state of the art in software development, but "virtually any [formal] specification would have turned up Heartbleed."

To assure software correctness with utmost confidence, semantically-founded techniques are required. The fundamental idea of programs having to meet assertions has been put forward by Turing [1949]; Floyd [1967]; and Hoare [1969]. They base formal software analysis on well-studied logical concepts. However, it is infeasible to perform this *assertional reasoning* by pen and paper. Firstly, the proof sizes are usually too large to be assessable by humans. Secondly, humans tend to err: many proofs that had been accepted by the community turned out to be invalid, especially for concurrent programs [de Roever et al., 2001, Sect. 1.4].

Verification Systems

By today, machine-supported formal verification has entered the mainstream; cf. [Beckert et al., 2006; Hinchey et al., 2008; Rushby, 2007; Filliâtre, 2011; Beckert and Hähnle, 2014]. There exists a plethora of implementations of verification techniques that target real-world programing languages such as C, C++, C#, Eiffel, or Java. These differ in the generality of their approach, or the dedicated support for particular classes of verification problems. Roughly, the resulting tools can be divided in two groups: tools based on verification condition generation (VCG) produce verification conditions as monolithic formulae of pure first order logic (FOL) (possibly with additional FOL-definable theories). The resulting formulae are to be checked with general purpose theorem provers, typically satisfiability modulo theories (SMT) solvers [Barrett and Tinelli, 2014]. While the theorem proving component is a manifestation of generality and trustworthiness, the weak spot is the VCG component, that translates programs together with high-level annotations to logic with only little transparency. This approach may be inefficient since

[3] As noted by de Roever et al. [2001], all these techniques may be established for sequential program development, but are a far cry from being applied to concurrent programs.

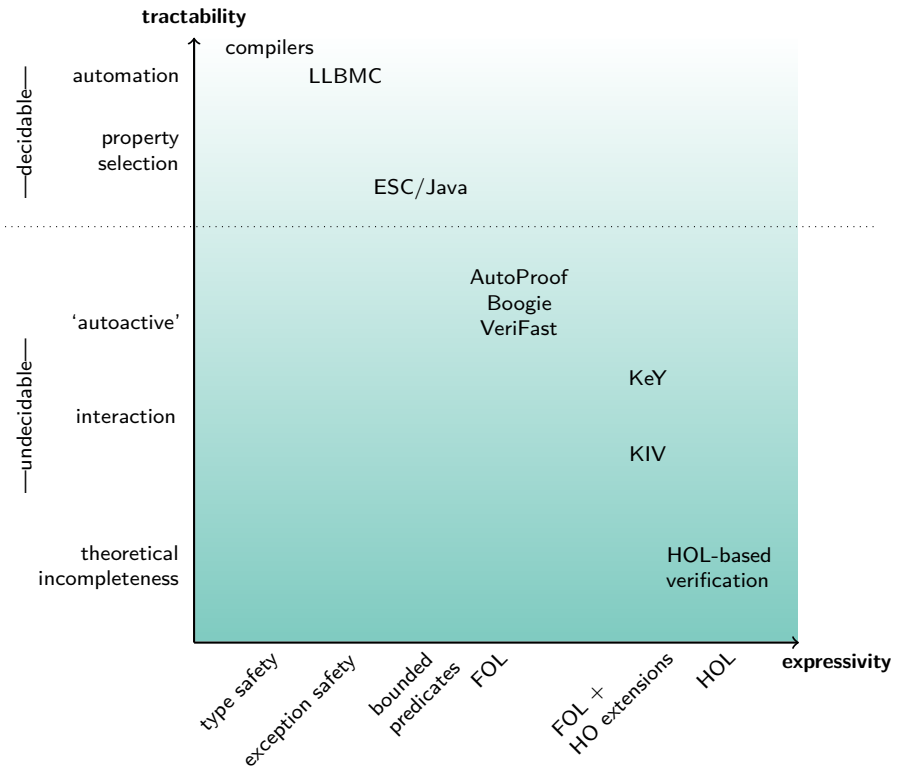


Figure 1.2: The continuum of static analysis approaches

elementary features of the target programing languages, such as typing or visibility, must be encoded in formulae. Pnueli [1977] calls these approaches *endogenous*.

Another class of verification systems, that includes KeY [Ahrendt et al., 2014] and KIV [Reif, 1995], uses specialized theories for reasoning dedicatedly about program correctness. In these systems—which are *exogenous* according to Pnueli [1977]—program semantics are transparently enshrined in the proof rules.

Historically, verification systems build on formal logics were supposed to be used interactively, with more or less automation support. Recently, tools build on the so-called 'auto-active' verification paradigm (a term coined by Leino [2009]) have enjoyed considerable success (see [Bormer, 2014, Chap. 5]). Verification systems of this kind include AutoProof [Tschannen et al., 2015], Dafny [Leino, 2010], KeY,[4] Leon [Kuncak, 2015], VCC [Cohen et al., 2009], VerCors [Amighi et al., 2014], Verifast [Jacobs et al., 2011], and Why3 [Filliâtre and Paskevich, 2013]. Auto-active verifiers are still based on general purpose theorem provers, thus being sound—in contrast

[4]KeY developed from an interactive prover and still allows low-level user interactions.

to many light-weight static checkers. User interaction with the verifier is restricted to providing auxiliary annotations. This direction of research seems very promising. In particular, user feedback directly refers to the program under test, not a barely legible logic representation. This is meant to lead to a flat learning curve, thus being usable to a wider audience. For problem sizes of the real world, proof situations are just too complex to be understandable to interactive users.

The central question is: what guarantees do these tools actually provide? The kind of problems that we are concerned with are undecidable, in general, cf. [Gödel, 1931; Davis, 2004]. This includes both properties of FOL, as well as properties about programs (that include the halting problem, among others). This implies that there cannot be an automated analysis technique that is sound and complete. 'Auto-active' verification includes interaction on a higher level. The dividing lines are the classes of properties for which the tool provides (theoretically) complete reasoning. Traditionally interactive systems gradually move towards high-level interaction; KeY is a good example of this. There is typically a trade-off between automation—and thus practicability—and expressivity of approaches, as displayed in Fig. 1.2 on the previous page. Finding a good balance is the major challenge in designing an effective verification approach.

In both autoactive and real interactive verification, the essential question is that of *practical completeness*. Provided a sufficiently expressible specification language,[5] theoretical (relative) completeness is obviously given. Practical completeness is about with what level of support (interaction or annotation) the automated verifier can succeed. Auto-active tools usually require a large amount of auxiliary annotations. Finding an appropriate specification has become the bottleneck of verification, cf. Zeller [2011]. State of the art specification languages like the Java Modeling Language (JML) [Leavens et al., 2006b] are intuitively understandable and provide the means to effectively specify program behavior. While JML is still a moving target, it does not provide features for concurrent programs yet.

1.4 Contributions and Structure of This Thesis

The present dissertation has been produced in the scope of the "Program-level Specification and Deductive Verification of Security Properties (DeduSec)" project[6] within DFG priority programme 1496 "Reliable Secure Software Systems (RS^3)."[7] It contributes to all three guiding themes of RS^3: information flow policies, information flow control, and security in the large.

[5] Most allow quantification over the integers or calling pure methods in specifications.

[6] `http://www.key-project.org/DeduSec/`

[7] `http://www.spp-rs3.de/`

Overall, this work describes—to the best of the author's knowledge—the first approach towards a **semantically precise** (i.e., without false-positives) **analysis of information flow in concurrent Java**. To this end, we present a precise formalization of information flow properties. When combined with a sound and complete verification technique, we obtain a precise information flow analysis. We present an implementation of the rely/guarantee approach to analyze concurrent programs in a deductive verification framework. It requires little specification overhead and is well suited for automation. We argue that it can be adapted easily to reason about concurrent Java programs. Our approach to concurrency verification primarily targets the application to secure information flow, but can be applied to verify safety properties as well. This work is supposed to be a starting point for adapting the KeY verification system to target concurrent Java programs.

The most discerning contributions that are provided to the scientific community in this dissertation are:

- Deterministic denotational semantics for a concurrent language with multi-threading, that uses underspecification instead of the more common nondeterministic semantics.
- Dynamic logic calculus, that is proven sound and complete
- Extension of the rely/guarantee approach with frame conditions and to open programs
- Adaptation of the rely/guarantee approach in a deductive calculus for dynamic logic
- Generic definition of noninterference, that can be instantiated with a large class of indistinguishability relations (e.g., to include declassification or for object-oriented systems)
- Precise formalization of trace-based noninterference for concurrent programs in dynamic logic
- Extension of JML regarding concurrency and translation to dynamic logic proof obligations
- Implementation in a prototype version of the KeY verification system
- Verification of an electronic voting system

For most of the work, preliminary material has been published (or is at least due to be published), as noted below. In cases, in which results or text fragments from these works are used in this thesis, this is clearly marked. In any case, the author has held a significant stake in their production.

Structure

This dissertation is divided into 11 chapters. It encompasses two interwoven major parts, dedicated to concurrent programs and information flow security, respectively. Readers who are only interested in functional verification of concurrent programs may skip Chaps. 2 and 6; while in Chap. 8, Sect. 8.3 is the interesting section. Readers who are only interested in information flow analysis may read Chap. 3 only briefly and skip Chaps. 5 and 7; the interesting section in Chap. 8 is Sect. 8.4. Chapter 10 on related work is also divided into related work on concurrency on one side and on information flow on the other side.

Chapter 2 covers the basic notions of software security. We introduce the concept confidentiality and classification systems (i.e., security lattices) in Sect. 2.2. Security policies, particularly including noninterference, are covered in Sect. 2.3. We provide an overview over the state of the art in information flow analysis in Sect. 2.4, with foci on both approaches based on theorem proving and approaches targeting concurrent programs.

Chapter 3 is devoted to the concurrent language *deterministic While-Release-Fork* (dWRF), that we will be using as the target of our investigations throughout most parts of this dissertation. dWRF is 'Java-like' in the sense that it distinguishes between global and local memory and caters for dynamic thread creation. Other features of Java, such as objects, are mostly orthogonal to concurrency and thus are left out for simplicity. Languages such as Dafny [Leino, 2010] or Chalice [Leino et al., 2009] are well known examples of this approach to study a particular programing language feature in a dedicated, but simplified language. The basic design decisions are discussed in Sect. 3.1.

We introduce the syntax of dWRF in Sect. 3.2. After defining a semantical modeling of heap memory in Sect. 3.3, we define the semantics of dWRF in Sect. 3.4: first for sequential programs locally (Sect. 3.4) and, on top of that, for concurrent programs (Sect. 3.5.2), which are defined as finite sets of sequential programs. Many approaches to analysis of concurrent languages view semantics as entirely indeterministic. This makes definitions and theoretical results based on this semantics less tractable. We define the semantics of concurrent programs parametric w.r.t. a *deterministic*, but underspecified scheduler. This semantics is agnostic concerning analysis techniques. **Previous work:** [Beckert and Bruns, 2012b, 2013; Bruns, 2015a]

Chapter 4 introduces Concurrent Dynamic Trace Logic (CDTL), a novel dynamic logic (DL) to reason deductively about multi-threaded programs of dWRF. Its semantics is based on symbolic traces—as opposed to single states—in order to reason about temporal properties of programs. CDTL

combines concepts from both dynamic logic and temporal logic. It is a conservative extension of the previously introduced Dynamic Trace Logic (DTL). Section 4.1 reviews the preliminaries. CDTL is introduced by syntax and semantics in Sects. 4.2 and 4.3, respectively. In Sect. 4.4, we present a sequent calculus for the fragment of the logic that is concerned with the sequential fragment of dWRF. This calculus is proven sound and complete (Sect. 4.5). Concurrent programs are to be covered in more detail in the following chapter. Section 4.6 contains a discussion on the covered logic fragment, including a full proof example. **Previous work:** [Beckert and Bruns, 2012b, 2013]

Chapter 5 forms the centerpiece in the 'concurrency part' of this dissertation. In that chapter, we discuss an extension of the above DL calculus to multi-threaded programs. We follow a symbolic approach to represent interleavings based on the rely/guarantee methodology [Jones, 1983]. We introduce an adaptation to CDTL in our framework in Sect. 5.2, that extends the classical rely/guarantee approach. In addition to functional specification, our approach uses frame conditions to describe non-behavior. The approach is entirely thread-modular.

We present the remaining calculus rules in Sect. 5.4: for interleavings in Sect. 5.4.1 and for thread creation in Sect. 5.4.2. We develop a final soundness theorem (Thm. 5.22) in a bottom-up fashion throughout Sect. 5.3. Section 5.5 contains larger examples of proof obligations arising from rely/guarantee specifications. In Sect. 5.6, we provide an outlook on future work to incorporate synchronization primitives into the language. While synchronization is essential to developing meaningful concurrent programs, it further increases the complexity of definitions and the verification approach. **Previous work:** [Bruns, 2015a]

Chapter 6 is the other central chapter of this dissertation, covering information flow in multi-threaded programs. In the introduction (Sect. 6.1), we present several program examples that are intuitively secure or insecure. In Sect. 6.2, we review the state of the art in security properties for sequential programs such as *noninterference*. We develop a flexible meta-level framework in which different notions of noninterference, including preconditions and semantical 'what' declassification, can be expressed. We show how these properties can be formalized in DL faithfully.

In Sect. 6.3, we leverage these properties to multi-threaded programs, additionally taking the scheduler into consideration. We show that system-wide security can be reduced to single threads in order to obtain modular security guarantees. In Sect. 6.4, we further extend this framework using a trace-based notion of noninterference. Behind this redefinition lies a stronger attacker model, that is more appropriate for concurrent programs as it

permits an attacker to observe intermediate events. In Sect. 6.4.2 we show how it can be formalized in the logic of Chap. 4. Furthermore, we compare it to the well-known notions of LSOD [Zdancewic and Myers, 2003].

Section 6.5 introduces a notion of *object-sensitive* noninterference suitable for Java programs, while only sketching the underlying formal definitions. We argue how this can be synthesized with the above to yield an effective analysis technique for information flow in concurrent *and* object-oriented software systems (Sect. 6.6). **Previous work:** [Beckert et al., 2013a,b, 2014; Bruns, 2014b, 2015b]

Chapter 7 reports on an implementation of the calculus and the specification in the KeY verification system. While in the previous chapters, we considered only a simple target programing language, in Chaps. 7f., we discuss how we can use these results to reason about multi-threaded Java programs. We briefly introduce KeY in Sect. 7.1. In Sect. 7.2, we discuss several issues related to multi-threaded Java that have not been considered before in the context of KeY. Section 7.3 provides an account on the implementation, in particular the instantiation of the rules in the *taclet* framework of KeY and a formalization of trace properties in standard dynamic logic.

Chapter 8 reviews the established techniques to modular specification and verification of Java programs. While these concepts have been developed to reason about sequential programs, modularity and encapsulation also play an important role in the analysis of concurrent systems. In particular, our approach to frame the effect of concurrent interleavings in Chap. 5 is heavily inspired by framing for procedure calls in sequential programs. The chapter centers around the widely-known Java Modeling Language (JML). Basic specification features of JML are introduced in Sect. 8.1. In the subsequent section, we discuss the fundamental concepts of (sequential) modularity (Sect. 8.2.1) and abstraction (Sect. 8.2.2), including the frame problem in sequential programs (Sect. 8.2.3).

The remainder of the chapter is dedicated to extensions that were developed within the DeduSec project. We present an extension to JML to accommodate rely/guarantee specifications in Sect. 8.3, that takes into account the considerations on Java concurrency from Sect. 7.2. Section 8.4 explains the approach by Scheben [2014] to specification of secure information flow, that is being used in the KeY system. We also briefly mention the Requirements for Information Flow Language (RIFL), that has been developed in RS^3. **Previous work:** [Beckert and Bruns, 2012a; Huisman et al., 2014; Grahl et al., 2016]

Chapter 9 reports on a case study in the verification of an implementation of an electronic voting system designed by Küsters et al. [2011]. We describe the system in Sect. 9.2. Section 9.3 describes a functional verification of a sequential implementation with KeY. This implementation is amenable to parallelization. The companion work on information flow analysis of the system was presented by Scheben [2014]. We also explain steps for a hybrid approach that combines the KeY approach with an automated analysis. **Previous work:** [Beckert et al., 2012b; Bruns, 2014a; Bruns et al., 2015a; Grahl and Scheben, 2016; Küsters et al., 2013, 2015]

Chapter 10 contains a discussion of related work. Sections 10.1 and 10.2 cover related work regarding logics and specification in general. Section 10.3 covers work related to our verification approach to concurrent programs. Sections 10.4–10.8 cover work related to our approach to secure information flow.

Chapter 11 concludes this dissertation. We summarize the results in Sect. 11.1 and discuss them in Sect. 11.2. We close with an outlook on future work (Sect. 11.3).

Other publications and unpublished material by the author that do not directly relate to this thesis are [Beckert et al., 2012a; Bruns, 2011; Beckert and Bruns, 2011; Bruns et al., 2011; Bruns, 2012; Bruns et al., 2015b; Ahrendt et al., 2014; Grahl and Ulbrich, 2016].

1.5 General Notational Conventions

Throughout this dissertation, we will use the following notations:

- The set of natural numbers $\mathbb{N}$ always includes 0. $\mathbb{N}_{>0} := \mathbb{N} \setminus \{0\}$.
- The ordering $<$ on natural numbers is extended to $\mathbb{N} \cup \{\infty\}$, where $n < \infty$ for all $n \in \mathbb{N}$.
- For subsets of $\mathbb{N}$, we use the interval notations $[i, j) = \{n \in \mathbb{N} \mid i \leq n < j\}$ (half-open interval) and $(i, j) = \{n \in \mathbb{N} \mid i < n < j\}$ (open interval), both were $j \in \mathbb{N} \cup \{\infty\}$.
- By abuse of notation, we usually write types in logic as the domain they represent in the intuitive semantics, e.g., $\forall i{:}\mathbb{Z}.\, \varphi$ means 'for all integers i.' E.g., the domain $\mathcal{D}_{\mathbb{Z}}$ of integer objects ('type $\mathbb{Z}$') is the set of mathematical integers $\mathbb{Z}$.
- For sets $S \subseteq S_0$, we denote the *complement* of S in S_0 by $S^{\complement}$, i.e., $S^{\complement}$ is the unique set satisfying $S \cup S^{\complement} = S_0$ and $S \cap S^{\complement} = \emptyset$. We omit reference to the superset S_0 when it is clear from the context.
- For a set S, we denote the *power set* by 2^S. The set of finite, nonempty subsets of S is denoted by 2^S_{fin}.

- We frequently abuse set notation and write $S \cup e$ instead of $S \cup \{e\}$ and $S \setminus e$ instead of $S \setminus \{e\}$.
- The standard equality symbol $=$ always means equality on the *meta-level* to distinguish it from the equality predicate in logic, written as $\doteq$. For instance, for terms denoted symbolically by t and t', we express syntactical equality trough $t = t'$, while $t \doteq t'$ is a formula (that is true if and only if the semantical value of t is the same as for t'). The equivalent holds for the semantical set operations $\in$, $\cap$, $\cup$, etc. and their logical counterparts $\dot{\in}$, $\dot{\cap}$, $\dot{\cup}$, etc.
- The formula $\neg t \doteq t'$ is frequently written $t \not\doteq t'$ for readability.
- Whenever an equation is meant to define an item, we use the symbol $:=$ to emphasize this.
- A relation R over sets $A_1, \ldots, A_n$ is identified with the set of ordered tuples $\{(a_1, \ldots, a_n) \mid R(a_1, \ldots, a_n)\} \subseteq A_1 \times \cdots \times A_n$. A function or partial function (i.e., functional relation) $f : A \to B$ is sometimes identified with the set of ordered pairs $\{(a, b) \in A \times B \mid f(a) = b\}$. We write partial functions as sets with elements of the shape $a \mapsto b$.
- A (possibly partial) function $f : A \to B$ can be *updated* to the function $f' := f\{a \mapsto b\}$ with $a \in A$, $b \in B$ that is defined as

$$f'(x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases} .$$

- Sequences, i.e., ordered multi-sets, are written with angle delimiters $\langle \cdot \rangle$. We use comprehension notation akin to set comprehension: $\langle x \in S \mid \varphi \rangle$ denotes the multi-set $\{x \in S \mid \varphi\}$ with the same ordering as S.
- Modalities of dynamic logic (see Sect. 4.1) are displayed with bold delimiters $[\pi]$ and $\langle \pi \rangle$ to distinguish them from other notations (such as sequences).
- Program syntax is set in `typewriter font`. In particular, in formulae, we use the typewriter font to distinguish program variables from logical objects, which are set in standard math font.

2 Software Security

In this chapter, we review the basic concepts of software security and how to enforce it. After an introduction in Sect. 2.1, we introduce the notion of security classification (and declassification) in Sect. 2.2. In Sect. 2.3, we present an introduction to language-based information security. Later, in Chap. 6, we will build on the notions that are introduced here. Section 2.4 reviews the state of the art in information flow analysis and control. software and hardware security certification.

2.1 Information Security

This section introduces the general concepts in computer security. For a standard textbook on the topic, see for instance [Anderson, 2008]. While safety is a family of properties stated about a system in isolation, security is a family of properties about the interaction of a system with external *agents*. Since this may be any agent, we rather use the terms *attacker* (or *adversary*) to denote an agent that tries to exploit the system and to do the most possible harm against the system. These interactions are known as *attacks*. Systems may have *vulnerabilities* that allow attacks. Attackers may be smart—any vulnerability must be expected to lead to a successful attack.

In principle, 'systems' in this context can be physical, software, hardware, cyber-physical, etc. Depending on the system, attackers can be human, computer programs, humans using computers, etc. Security properties can be broken down into the following four categories:

- *Confidentiality*: secret information must not be disclosed to unauthorized agents.
- *Integrity*: unauthorized agents must not interfere with the system.
- *Authenticity*: the system must not be forged.

- *Availability* (or, *robustness*): the system must be working as specified at any time.

In this dissertation, we consider software systems on the code level and confidentiality properties thereof, i.e., we investigate *language-based* security.[1] Attackers in this scenario may know the code, run it (on input of their choice), and observe its effect. In particular, attackers are expected to have knowledge of all security measures.[2] The design of security provisions always needs to be designed under this regard. Thus security can be seen as a game between security engineers, developing measurements against particular attacks, and attackers, mounting attacks against particular measurements. Integrity can be seen as the dual property to confidentiality, with communication channels and security levels inverted [Biba, 1977]. The results that we describe therefore can be applied to integrity as well.

In general, we expect an attacker's deductive powers not to be computationally bounded; they can apply any function on public information to obtain all corollaries. For instance, if the attacker knows that the value of secret a is 5 and that the sum $a + b$ is 12, then they also know that b is 7. Information may be partial: the knowledge that a is a positive value or that a is not 7 is less than the complete knowledge of its value.

On the other hand, *cryptographic* security works under the assumption that attackers can only compute functions of polynomial complexity. This is called a Dolev/Yao attacker [Dolev and Yao, 1983]; see also Sect. 9.2.1. If decryption requires solving exponential complexity problems, such as prime number tests, then this encryption is considered secure.[3] To symbolically assess security under cryptographic guarantees with unbounded adversary approaches, cryptography is considered a secure black box component [Küsters et al., 2012].

[1] Of course, language-based security can only form one piece of the puzzle of entire security of the system. For instance, we cannot consider attackers that have access to hardware. Even on the software side, language-based security leaves gaps since it only considers program *source code*. The goal that also compiled programs are at least as secure as the source code it was produced from can be achieved with so-called *fully abstract* compilers [Abadi, 1998].

[2] However, they may not change the code nor have control of the underlying hardware or middleware (i.e., they are *passive* attackers).

[3] As an aside, secrecy against this class of attackers can only be given under the assumption that the $\mathcal{P} \neq \mathcal{NP}$ conjecture [Cook, 1971; Levin, 1973] is valid and usable quantum computers [Deutsch, 1985] do not yet exist.

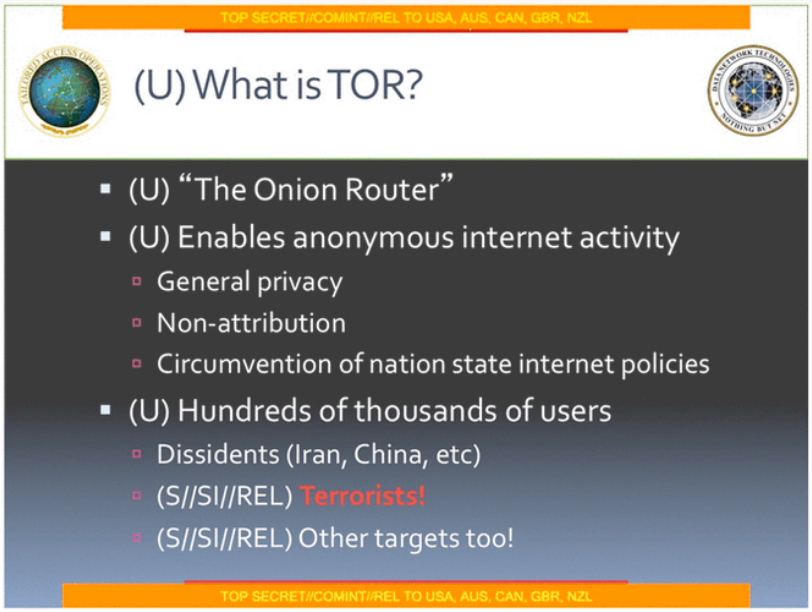


Figure 2.1: This slide is part of a presentation by the National Security Agency (NSA), leaked to the public by Edward Snowden and first published by The Washington Post. It contains information with different security classifications. Most lines, including the head line, are marked U, i.e., unclassified. The information that the system may be used by terrorists is marked 'secret' S. The combination S//SI further restricts access to personnel in the 'special intelligence' compartment. The additional mark REL means that secret information may be shared to (appropriate personnel in) other countries (in this case Australia, Canada, the UK, and New Zealand as stated in the footer).

2.2 Classification

When considering confidentiality, all information is assumed to be assigned a security level, i.e., it is *classified*. Information on the lowest security level is called *unclassified*. In general, some information is 'more confidential' than other, meaning it is assigned a higher classification. We formally describe that by the binary relation $\prec$. When an information is assigned a classification C, only parties with a *clearance* of C' with $C \preceq C'$ may access this information.

Classification systems can grow into complex hierarchies. A classical system, that had been used by public administrations long before the age of computers, uses the 5 levels 'unclassified' $\prec$ 'confidential' $\prec$ 'secret' $\prec$ 'restricted' $\prec$ 'top secret' (sometimes with only 4 levels, lacking 'restricted'). This particular hierarchy is linear, but in general, a hierarchy can be any finite lattice.[4] For this reason, the term *security lattice* [Denning, 1976] is often used in place of 'classification system.'

[4] A *lattice* $(L, \prec)$ is a partially ordered set in which any two elements have both a unique least upper and greatest lower bound (i.e., supremum and infimum). Bounded

For instance, the classification system used by the United States government [US Government] uses the 4 basic levels as described above, but adds further restrictions or overrides. The classification S//NOFORN (secret, no foreign nationals (even if they have 'secret' clearance)) is strictly above S (secret) in the hierarchy, but incomparable to TS (top secret). On the other hand, TS//REL TO AUS means that the information is 'top secret' in principle, but may be shared to (appropriate government personnel in) Australia—it is strictly below TS, but incomparable to S. An example of an originally classified document from the real world can be seen in Fig. 2.1 on the preceding page.[5]

An excerpt from this security lattice can be seen in Fig. 2.2. It contains distinct top $\top$ and $\bot$ bottom elements. To prove that the confidentiality property holds for a given lattice, we need to show that no information with a classification C' is leaked to a party with clearance $C \prec C'$. Due to the nature of the $\prec$ relation of being transitive, it suffices to prove this property for every pair of *immediately* related levels; cf. [Scheben, 2014, Lemma 7], for instance. Thus, the general problem can always be reduced to a number of problems each regarding a two-element lattice. In the remainder of this dissertation, we will use this result and restrict our considerations to two-element lattices without loss of generality. The elements will conventionally be called 'high' and 'low.'

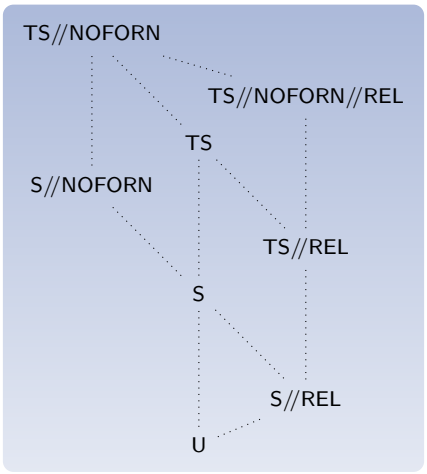


Figure 2.2: In this excerpt from the US Government classification system, the base security levels TS, S, and U are refined with NOFORN or REL annotations, which respectively raise or decrease the baseline security level.

lattices have unique least and greatest elements $\bot$ and $\top$. Note that we do not require lattices to be complete.

[5] Another example, immediately related to this thesis, is the paper by McLean [1992] introducing the notion of low-security observational determinism (LSOD) (see Sect. 2.3 below). While the rest of the paper is public, the central theorem is classified NOINT ('no international use').

Declassification is the deliberate release of otherwise classified information (also known as *downgrading*). It is usually not included in the declaration of a security classification, but is defined as an override to it. For instance, the document in Fig. 2.1 is classified as 'top secret' in general, but the individual text fragments displayed there are classified as 'secure' or are even unclassified. Using the taxonomy of Sabelfeld and Sands [2009], it can include the dimensions 1. *what* information is released (subjective dimension) 2. *by whom to whom* 3. *where* in the system (spatial dimension) and 4. *when* (temporal dimension). Declassification is often essential to the functionality of the system. A typical example is access control through password checking: a user is prompted for an input string and will be granted access to the restricted subsystem if and only if the input equals their password. The password is considered secret, yet the system does release some information about it, namely whether it is equal to the entered string.

In our setting, where we use a semantical notion of information flow, we are mostly concerned with *what* information is released. Other items, like who releases to whom, can be considered if they are appropriately represented in the semantical model. The spatial dimension can be further divided into *level locality* and *code locality*. The code-local dimension is important to many syntax-based analysis techniques (see below), but cannot be represented in a semantical model.[6] In concurrent programs, we also consider the temporal dimension since several internal program states may be observable. In general, the temporal dimension can also refer to complexity classes or even physical time, but we only consider the relative order of publicly observable events (cf. [Sabelfeld and Sands, 2009, Sect. 2.4]).

2.3 Secure Information Flow

In *language-based* information flow security [Sabelfeld and Myers, 2003a], we regard software systems at the source code level. In this model, attackers can use the present code (i.e., run it or read from memory) as the programing language permits it. Attackers can not change the code or have access other than through defined language constructs.[7] In particular, this notion of software security is hardware independent. Attacks are carried out on idealized machines.

Sources and Sinks

In shared-memory systems, on the language level, information sources (or 'inputs') and sinks ('outputs') usually are just memory locations. For a

[6] The *dynamic* classification approach by Scheben and Schmitt [2012a] achieves a similar goal, though.

[7] For instance, in high level languages like Java, an attacker can not observe the raw memory location of a variable.

modular analysis of programs, method calls have to be considered sources or sinks, too. Whether a location is regarded as a source or as a sink depends on the mode of access, i.e., read or write. In contrast to this traditional view of memory locations as sources and sinks, Scheben and Schmitt [2012a] introduce a more flexible notion of *observation expressions*; see also Sect. 8.4. Arbitrary complex expressions of first order logic (FOL) can act as sources or sinks. This includes, in particular, method parameters and return values; thus allowing the approach to be method-modular. This is a generalization of the aforementioned definition and unifies it with the—otherwise separately considered—concept of declassification. The intuition behind this is that a passive attacker does not 'sit in the machine,' but observes its behavior by posting queries to it.

2.3.1 Channels

Sources and sinks are connected through *channels*. Channels are commonly categorized in *direct* and *covert* channels [Lampson, 1973]. A direct channel is primarily intended for information passing, while information flow through covert channels is unintentional in general. On the language level, a direct channel can manifest as an assignment or the return of a method call, etc. This results in an explicit information flow. Listing 2.3 on the facing page shows a minimal example of a direct information flow in a Java-like language—even though only parts of the original confidential information flow to the public sink.

Covert channels include the following kinds according to Sabelfeld and Myers [2003a]:

- "*Implicit flows* signal information through the control structure of a program.
- *Termination channels* signal information through the termination or nontermination of a computation.
- *Timing channels* signal information through the time at which an action occurs [...]
- *Probabilistic channels* signal information by changing the probability distribution of observable data. [...]
- *Resource exhaustion channels* signal information by the possible exhaustion of a finite, shared resource, such as memory or disk space.
- *Power channels* embed information in the power consumed by the computer [...]"

Attacks on covert channels are particularly common if direct channels are (practically) secure. For instance, sophisticated attacks on cryptographic protocols do not use brute force, but often timing or power channels (cf. [Kocher, 1996]), because the resource consumption is observable in practice, e.g., multiplication takes more processor cycles than addition.

We do not consider resource or power channels in this work since they can only be regarded in combined cyberphysical systems. Instead, in language-based security analysis, we assume idealized hardware (both functionally and security-wise). For instance, as explained in Chap. 3, our program model assumes an infinite memory. Likewise, probabilistic leaks do not appear in our model, where program execution—even for concurrent executions—is defined as being strictly deterministic.[8]

The remaining covert channels, implicit flows and termination—and also timing to some extend—exist in our computational model. Since we follow a semantical approach to analysis—i.e., we actually consider the flow of *information* itself, not just the potential existence of a channel—such channels are naturally considered. A minimal example for an implicit flow is shown in Listing 2.4: there is no direct assignment to a low sink, yet the value of `low` depends on `high`. Listing 2.5 shows a termination channel: there is no flow to a low sink at all, but termination of the program depends on a high source. It is sometimes argued—cf. Zdancewic and Myers [2003], for instance—that termination channels were negligible as they would 'only' leak 1 bit of information. Askarov et al. [2008] disagree and showed that this leakage can be larger in general.

```
low = high % 5;
```
Listing 2.3: Direct flow

```
if (high) low = 23;
```
Listing 2.4: Implicit flow

```
while (high) {}
```
Listing 2.5: Termination channel

```
while (low < high)
    { low = low+1; }
```
Listing 2.6: Timing channel

Zdancewic and Myers [2003] further distinguish between 'internal' and 'external' observations of a system. External observations can be made by observers outside the system, such as measuring responses against wall-clock time. Internal observations are those by other components within the same system, that may make use of mechanisms internal to the system. An example would be the order of memory accesses. Listing 2.6 shows a leak through an internal timing channel: the number of assignments to `low` depends on a secret.

Quantitative information flow security [McIver and Morgan, 2003; Mu, 2008] is a generalization of the above concept of security. Instead of considering whether a channel is secure or not, it is the *quantity* of information that flows through. There are multiple concrete definitions, each based

[8] As we explain below, our approach would be able to detect probabilistic leaks if they were modeled.

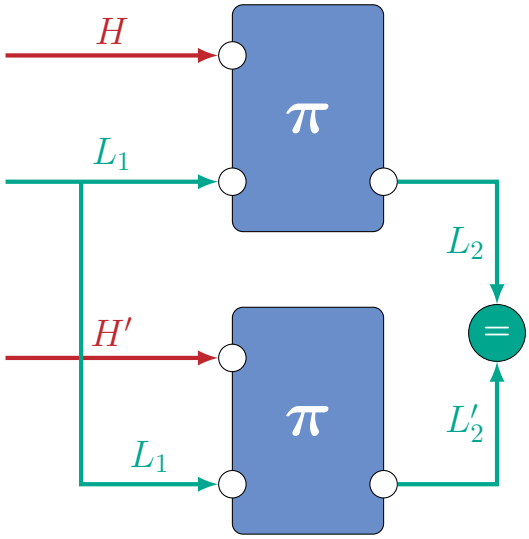


Figure 2.7: Two instances of a program π are run with the same low input, but possibly different high inputs. If the respective low outputs are again the same, then π is noninterferent.

on information theory, that use different entropy metrics. A quantitative notion of security obsoletes declassification by setting a maximum threshold of information to be allowed to leak. This is particularly important for cryptography, which is difficult to assess with qualitative analyses.

2.3.2 Security policies

Policies are concrete definitions of a system being secure against some class of attackers. They are sometimes stated in an informal manner, or only implicitly through an associated analysis technique. But at least with the development of semantically precise analysis techniques (see Sect. 2.4.3 below), there is a need for formal definitions. *Noninterference* is a semantically defined policy, that was first mentioned by Cohen in 1977 and later popularized by Goguen and Meseguer [1982]. Intuitively, it means that "the value of public outputs does not depend on the value of secret inputs" [Barthe et al., 2004].

To test whether a given program π is noninterferent, it is executed twice: both executions start with the same low input L_1, but possibly different high inputs H and H'. Regarding two respective outputs L_2 and L_2', they do not depend on the respective high input if and only if they are equivalent again. A graphical rendition of this principle is displayed in Fig. 2.7.

Cohen's original definition only targets terminating, sequential, and deterministic systems. Sabelfeld and Myers [2003a] define several extensions of noninterference for different classes of nondeterministic systems. The *generalized noninterference (GNI)* policy by McCullough [1988] defines a

nondeterministic system as secure if there exists at least one choice that is secure. This definition is very liberal. As Zdancewic and Myers [2003] point out, under this definition, many concurrent programs are classified as 'possibly secure' that are intuitively considered insecure—for instance the following program with three threads:

```
L = true;    ||    L = false;    ||    L = H;
```

For every value of the secret location `H` and a scheduler that selects the leaking thread last, there is a scheduler such that it selects one of the other threads that does reveal the same value. This class of attacker is not able to distinguish between the secret value and the same value of a constant. Security under this definition means that it is *possible* that the attacker does not learn the secret.

Low-security observational determinism (LSOD) conversely defines a nondeterministic system as secure if *all* choices are secure [McLean, 1992]. These definitions of noninterference for nondeterministic systems are largely applied to concurrent systems, as scheduling is often modeled as a kind of nondeterminism. Since multi-threaded systems are expected to output information throughout their execution, LSOD is defined on an execution *trace*, rather than a single final system state. As proven by Huisman, Worah, and Sunesen [2006], LSOD implies the absence of probabilistic leaks, even though probabilities are not modeled explicitly.

As for basic noninterference, there are many variations of the definition of LSOD [Roscoe, 1995; Roscoe et al., 1996; Zdancewic and Myers, 2003; Huisman et al., 2006; Terauchi, 2008]. While both Zdancewic and Myers and Huisman et al. consider traces of values of individual variables, in Terauchi's work, traces consist of low state partitions. On the other hand, Huisman et al. always consider a complete trace (up to stuttering), while both Zdancewic and Myers and Terauchi relax trace equivalence to prefixes, which permits flows that occur later in the program execution.

LSOD considers scheduling to be completely indeterministic. This raises the issue that both interference from high variables and benign indeterminism on low variables needs to be considered, which is overly strict. To overcome this, Huisman and Ngô [2012] introduce scheduler-specific observational determinism (SSOD). It extends the weaker property that is proven by the type system of Zdancewic and Myers [2003]—named SSOD-1 here—with a second property (SSOD-2) that requires that there *exists* a scheduler for which traces are stutter-equivalent. Giffhorn and Snelting [2015] introduce *relaxed* low-security observational determinism (RLSOD) with the same goal in mind. Giffhorn and Snelting use a different notion of 'trace' than the aforementioned authors: a trace is a sequence of observable program events (e.g., read, write, fork, etc.)

Amtoft, Bandhakavi, and Banerjee [2006] first proposed an extension of noninterference that is appropriate for object-oriented systems. In Sect. 6.5, we present a formalization of object-sensitive noninterference in dynamic logic, as introduced by Beckert, Bruns, Klebanov, Scheben, Schmitt, and Ulbrich [2014].

GNI, LSOD, and SSOD are kinds of *possibilistic* security properties [McLean, 1996]. They classify concurrent programs into 'secure' and 'possibly insecure.' Attacks that are additionally possible through possible interleaving are called *refinement attacks* [Zdancewic and Myers, 2003]. *Probabilistic noninterference* [Sabelfeld and Sands, 2000; Smith, 2003; Snelting, 2015] is a generalization of possibilistic noninterference, considering probabilities of schedules. Programs are considered secure if the probability of any possible program trace being followed is the same for any public input. As mentioned by Snelting, attacks on probabilistic channels can also be (internal) timing attacks. This allows computational models without probabilities, but a notion of time, to describe a some of these attacks. LSOD considers the order of individual variables and strictly implies probabilistic noninterference.

For an overview over security policies other than those based on noninterference, cf. the survey paper by Zakinthinos and Lee [1997]. It includes *noninference* [O'Halloran, 1990] (which is weaker than noninterference) and *separability* [McLean, 1994] (which is stronger). Zdancewic and Myers [2001] introduce the concept of *robustness*. This is similar to noninterference, but allows flows to low sinks as long as the adversary cannot control what information flows. Thus, robustness is a combination of a confidentiality property and an integrity property, both of which can be represented through noninterference. Popescu, Hölzl, and Nipkow [2012] present additional policies *self-isomorphism* and *discreeteness* to discuss secure information flow in concurrent programs. We do not further discuss those policies here as they are not as frequently used in the literature as noninterference.

2.4 Information Flow Analysis and Control

Above, we have introduced notions of security itself. In this section, we review the state of the art in techniques to analyze and enforce security. These techniques can be categorized into *dynamic* and *static* analyses. A particular subclass of static analyses includes approaches based on formal semantics, such as our own. Those are discussed in Sect. 2.4.3.

Important characterizations of analyses are soundness and completeness. An analysis is *sound* if it detects every forbidden flow that would be possible. A sound analysis classifies every insecure program as such. A missed leak is called a *false negative* error. An analysis is *complete* (or *precise*) if every secure program can be classified as secure. The failure of recognizing a secure

program (by reporting a spurious leak or no result at all) is called a *false positive* error.

2.4.1 Dynamic Analysis and Control

The earliest approaches to secure information flow analysis and control were based on runtime checking. In the context of information flow *control*, i.e., enforcing secure information flow, we usually speak of runtime *monitoring*. A well known example of a monitoring approach is the one by Bell and LaPadula [1973]. These analysis techniques check at runtime whether there is an assignment from a high variable to a low variable. But they fail to detect implicit flows, although they use security *labels* attached to each statement to distinguish high contexts such as the conditional statement depending on a high variable as in Listing 2.4. They will report the case where `high` is true as being insecure, but fail to detect the implicit flow in the opposite case. These dynamic analyses are always unsound: they may miss a possible security violation that does not appear in the particular run.

Implicit flows can only be detected entirely by comparing all possible execution paths, which is not possible at runtime. Dynamic analyses can be made sound by considering the remaining program after the high conditional as a high context, too. This effect is known as *label creep* and yields a very restrictive, highly incomplete analysis. For instance, later erasure of low variables will not be recognized as lowering the security context. In an extreme case, any program would be considered as insecure. Declassification can only be considered so far that from a given point in the execution on (i.e., 'where' declassification), *any* information will be permitted to flow. These points are marked by *escape hatches* [Sabelfeld and Myers, 2003b]. Despite these major weaknesses, dynamic analysis has seen a revival as noted by Sabelfeld and Russo [2009].

2.4.2 Static Analyses

Static analyses allow to consider multiple execution paths and therefore do not suffer from the problems of dynamic analyses. Analyses based on type systems [Denning and Denning, 1977; Volpano et al., 1996; Volpano and Smith, 1997] have been the predominant information flow analysis technique until recently. In type-based approaches, program elements do not only have the usual types (such as `int`), but also a *security type*. If a program is type-safe (which can be checked at compile time) then it respects the given confidentiality property.

Type system approaches are usually sound; i.e., they are guaranteed to find all security violations. But they are incomplete w.r.t. sophisticated policies such as noninterference. This means that they tend to report false positives by rejecting (semantically) secure programs. Some type systems are highly incomplete: For instance, the type system by Smith and Volpano [1998] completely rejects loops with 'high' guards.

There exists a plethora of type systems for different programing languages using different extensions to narrow the incompleteness gap; e.g., *decentralized labels* [Myers and Liskov, 1998], *floating types* [Hunt and Sands, 2006], or *gradual types* [Fennell and Thiemann, 2015]. In particular, floating types allow for *flow-sensitive* analyses, which has become the standard for type systems. First ideas to extend type checking to multi-threaded languages have been published by McCullough [1987]; Smith and Volpano [1998]; and Boudol and Castellani [2002]. A widely used implementation of type-based analysis for sequential Java programs is the Java with Information Flow (JIF) compiler [Myers, 1999], that however requires a specialized dialect of Java. Other type systems for sequential Java include [Strecker, 2003; Banerjee and Naumann, 2005; Barthe et al., 2013c].

A related technique is explicit *dependency tracking* of variables. Pan [2005]; Bubel et al. [2009]; van Delft [2011] present approaches—built on symbolic execution—in which for each variable a set of variables it is stored on which its value depends at most.

Program dependence graphs (PDGs) provide an alternative static technique [Hammer, 2009; Hammer and Snelting, 2009]. In this approach, secure information flow is reduced to a property of graph-theoretical reachability. It is based on the fundamental insight that many implicit flows are linked to program control dependencies. In a PDG, nodes represent sources and sinks while directed edges represent dependencies between them, like an assignment or a branch condition. The PDG approach is implemented for Java bytecode in the JOANA tool [Graf et al., 2013]. Snelting, Giffhorn, Graf, Hammer, Hecker, Mohr, and Wasserrab [2014] extended this technique to concurrent Java, checking for RLSOD. Although they provide a more precise treatment of implicit flows, program dependence analyses are still incomplete in general. Several approach to further narrow the completeness gap exist, for instance by Gocht [2014]. For a comparison between the type system and PDG approaches, see [Mantel and Sudbrock, 2012].

2.4.3 Semantical Approaches

A faithful formal semantics for a programing language allows analyses that are both sound and complete, combining the strengths of both dynamic and static worlds. In the safety community, this semantical principle has already been established for some time. Joshi and Leino [2000] and Amtoft and Banerjee [2004] were the first to formalize information flow properties in a program logic. They use dedicated relational extensions of Hoare logic [Benton, 2004].

The *self-composition* approach by Barthe, D'Argenio, and Rezk [2004] works with an off-the-shelf program logic but uses program transformations to express relational properties. Thus a security problem can be reduced to an equivalent safety problem [Terauchi and Aiken, 2005]. Self-composition is appealing, due to being easy to implement. For instance, the Fährmann system [Beuter et al., 2013] is an undergraduate student project that uses self-composition and a weakest precondition calculus to prove the noninterference property in a simple imperative language. Imprecise analysis techniques can only overapproximate semantical information flow properties like noninterference. *Product programs* [Barthe et al., 2011] are similar to self-composed programs, but again use a dedicated program logic.

Darvas, Hähnle, and Sands [2005] and Scheben and Schmitt [2012a] present formalizations of noninterference in dynamic logic. These require neither a dedicated logic nor program instrumentation. In contrast to Hoare logics, sentences of dynamic logic are freely composable, thus allowing to express relational properties naturally. If specification languages are expressive enough—such as the first order dynamic logic used in the latter named approaches—we can also express full semantical declassification (i.e., subjective declassification) [Banerjee et al., 2008b].

Taking these advantages into account, semantical information flow analysis is a much stronger technique than dynamic or traditional static approaches. Many of the research questions that are still open or currently discussed in the context of type-based or PDG-based approaches are already solved by construction. On the downside, security properties like noninterference are theoretically undecidable—at least, they are expensive to prove in practice. In general, they are not provable in a fully automated fashion.

2.4.4 Comparison of Dynamic and Static Analysis

There is a kind of dualism between dynamic and static analyses (cf. [Russo and Sabelfeld, 2010]). This is not particular to security analysis, but also well known in the safety world; cf. [Ernst, 2003]. On the one hand, dynamic analyses are semantically precise since they execute the actual code. Static analyses operate on a *model* of the system. Type systems and PDGs (as well as so-called extended static checking (ESC) [Detlefs et al., 1998] for

safety) use an abstracted state model and are therefore incomplete, but very efficient.

Logic-based approaches are usually precise because they are built on a semantical model that captures all possible program states symbolically. Given this advantage, logic-based approaches appear in both worlds. The particular challenge is to lift these techniques that were primarily designed for functional analysis to the relational setting. Chudnov et al. [2014] regard runtime monitoring as a kind of abstract interpretation for a relational logic. On the other hand, dynamic analyses can effectively never be sound.[9] But this possibly requires an infinite number of test cases. Static analyses are usually conservative in order to be sound.[10] Even logic-based techniques, that are theoretically precise, can lack *practical* completeness. As they are concerned with problems that are undecidable in general, they have to rely on 'good' specifications provided by a user.

Considering these techniques as complementary, it is possible for them to work together in a hybrid approach like the one proposed by Küsters, Truderung, Beckert, Bruns, Graf, and Scheben [2013]; Küsters, Truderung, Beckert, Bruns, Kirsten, and Mohr [2015] and discussed in Sect. 9.3.3 of this dissertation.

[9] In theory, they *could*—by exhaustively covering every possible execution path.

[10] A notable exception is the ESC/Java tool [Flanagan et al., 2002], that implements the above named extended static checking technique.

3 Concurrent Programs

In this chapter, we introduce our target concurrent language. Besides the usual constructs for sequential execution, it features a `fork` statement to spawn a fresh thread. We will develop denotational semantics for sequential programs and subsequently for concurrent programs. Concurrent programs consist of a set of threads, where each thread executes a sequential program that is interleaved by the environment. Concurrent changes to the shared memory are modeled explicitly in the program code through explicit release points and an explicit—yet underspecified—scheduling function. The statement `release` syntactically represents a release point in the code.

The semantics defined in this chapter is agnostic concerning analysis techniques. This will serve as the foundation for the dynamic logic to be introduced in Chap. 4 as well as for Chap. 5 on modular reasoning about concurrent programs using rely/guarantee. We start by reviewing the established concepts of concurrency and by explaining the foundations of our approach.

3.1 Approach Overview

All kinds of programs can be described by their observable behavior. For sequential programs it suffices to describe the relation between initial and terminal system states (or between a multitude of possible states). Even for sequential programs, the exact definitions of observable behavior widely diverge, e.g., regarding termination, exceptions, heap structures, or side effects, etc. Modules of sequential programs are (public) procedures. The techniques for modular reasoning about sequential programs are design by contract, behavioral subtyping, etc. (cf. [Grahl et al., 2016]).

Concurrent programs may allow even more observations. “The key to formulating compositional proof methods for concurrent processes is the

realisation that one has to specify not only their initial-final state behaviour, but also their interaction at intermediate points." [de Roever et al., 2001].

Devising a *minimal* programing language is the key to verifiable programs, as frequently advocated by Hoare [1981]: "You include only those features which you know to be needed for *every* single application of the language [...]. Then extensions can be specially designed where necessary [...]." In this thesis, we consider a simple concurrent imperative language, that we call *deterministic While-Release-Fork* (dWRF). It extends the sequential language presented by Beckert and Bruns [2013] with interleavings and dynamic thread creation. It is 'Java-like' in the sense that it uses both local and global variables (aka. fields) and that an arbitrary number of sequential program fragments[1] can be executed concurrently. dWRF distinguishes between local variables with atomic assignments and global variables with assignments inducing (local) state transitions. The rationale behind this is that, in a concurrent setting, only global memory can be observed by the environment. Expressions do not have side effects. New threads can be spawned in a simple `fork` statement, that includes the program of the thread to create, but does not have parameters. Synchronization is not considered at the moment and will be left to future work. We introduce the syntax of dWRF in Sect. 3.2.

Other Java features such as objects, arrays, types, or exceptions are not of relevance to our discourse. These features are largely orthogonal to each other (cf., e.g., [Stärk et al., 2001]) and could be added without invalidating the central results.[2] All such features can be added in principle, but we keep the programing language simple for the presentation in this chapter. In Sect. 7.3, we discuss how to extend dWRF to full Java, which will lead to the development of an extension of the KeY verification system [Ahrendt et al., 2014] to concurrent Java.

We assume that write actions are immediately visible to the environment, i.e., we assume sequential consistency of the memory.[3] On the other hand, concurrent changes induced by the environment only appear in the semantics when they actually may have an effect, namely upon read actions or termination.

[1] Throughout this thesis, we will use the term 'program' for sequential program fragments (or, 'blocks' in Java). This is the usual notion of programs in the context of dynamic logics. We will sometimes use 'system of (sequential) programs' to denote entities that are considered 'programs' in other contexts. Since we do not introduce method calls in our simple language, this distinction is not essential.

[2] As Jones [1996] argues, object-orientation is even helpful to constrain concurrent interleavings.

[3] Unfortunately, the Java memory model (JMM) does not guarantee this property; see Sect. 7.2 for details.

3.1.1 Explicit Thread Release

The common approach to define an interleaving semantics of concurrent programs is to alter the semantics of read actions to encompass additional havocking of values. In order to extend the language defined by Beckert and Bruns [2013] in a *conservative* manner, we refrain from this idea. Instead, we introduce explicit *release points* [Dovland et al., 2005], a concept borrowed from the Abstract Behavioral Specification (ABS) language [Johnsen et al., 2010; Hähnle et al., 2011]. Release points denote in the code that a thread voluntarily releases control and the scheduler may select another thread. We represent this through explicit `release` statements, whose semantics is defined through the local semantics of the environment threads. All other program statements are not affected by the environment. Even though in real concurrent programs interleavings may occur at any point in time, this setup is sufficient to model such systems, while it greatly reduces the number of program states in which we must expect interleavings.

For most of this chapter, we just assume that `release` statements may or may not appear in the code. In Sect. 3.6, we discuss how this can be used to model actual concurrency. A purely sequential program can be explicitly instrumented with `release` statements at the relevant program points to model interleaving behavior. Conversely, atomic blocks (if present in the target language) can be represented through the absence of `release`.

Determinism

While unconventional, we define the semantics of concurrent programs in a deterministic way. Many models of concurrent program executions regard programs as indeterministic. For instance, Zdancewic and Myers [2003] postulate that reasoning about confidentiality in "concurrent languages is problematic because these languages are naturally nondeterministic; the order of execution of concurrent threads is not specified by the language semantics." We deliberately do not follow this paradigm. There is both a practical and a theoretical rationale behind this decision. Firstly, nondeterminism is just *a model* for unknown behavior; it is not 'natural.' In the physical world, there is no such thing as nondeterminism. Secondly, many definitions are much easier to give in terms of deterministic programs. E.g., the effect of a program should be the same when run twice from the same initial state under the same scheduler. This allows us to talk about exactly one computation trace, which greatly improves tractability of reasoning—in particular for relational properties. Zdancewic and Myers are right about the execution order not being *completely* determined by program semantics. But instead of nondeterminism, we prefer to view this as a kind of parametricism. We model unknown behavior on the language through *underspecification* [Gries and Schneider, 1995]; cf. Sect. 3.1.2 below. As explained by Hähnle [2005],

the underspecification approach is the most viable option to model undefined or partially defined behavior.

The semantics of dWRF is meant to extend the semantics of the sequential language by Beckert and Bruns [2013] in a conservative way. However, in contrast to Beckert and Bruns [2013], we model global memory using an explicit (ghost) program variable `heap`, as explained in Sect. 3.3. The semantics of `heap` is a mapping from global variable names to values. This modeling caters both for abstract anonymization (i.e., havocking) on (possibly underspecified) parts of the heap and for a convenient comparisons of the entire memory, that we need for the techniques presented in Chaps. 5f. Program semantics with explicit heap representations have been used in [Weiß, 2011], for instance. We extend Weiß's approach with a second variable `heap'` to denote the heap in the previous state, that we use to represent two-state invariants in the rely/guarantee approach in Chap. 5.

3.1.2 Scheduler Assumptions

Our approach is widely scheduler agnostic. Validity of program properties will be defined in Sect. 3.4 w.r.t. (almost) *any* scheduler; we only make the following six fundamental assumptions. A formalization of these properties will appear in Defs. 3.12 and 3.16 on pages 45ff. A general framework to formalize scheduler policies is not part of this work.

1. The number of active threads is always finite.

2. The scheduler selects an active thread in any state in which at least one thread is active, i.e., a thread that is not yet terminated.[4] Without loss of generality, we assume that there is always an active thread. This assumption could be realized with a synthetic 'idle' thread that infinitely loops with ineffective global write actions.

3. The schedule only depends on the heap state. This implies that a scheduler is deterministic. This should not impose a loss of generality since a nondeterministic scheduler can be simulated through a set of deterministic schedulers. There already exist formalizations of concurrent program semantics using deterministic schedulers in the literature, e.g., by Beckert and Klebanov [2013]. In fact, indeterminism does not offer more expressiveness as we only consider properties that are valid w.r.t. *any* deterministic scheduler.

Typically, there will be a special thread that manages the schedule, remembers its history, book-keeps the active threads, and performs the necessary computations. We assume that this particular thread does not

[4]Thread suspension is not considered in this work.

get stuck in an infinite loop. Even though all other threads may not be advancing to a different global state, we can assume that this thread will.

4. No state is reached twice. We assume that a management thread as described in item 3 is present, without modeling it explicitly; it shall at least keep a global program counter. Given this assumption, a deterministic scheduler is not less capable than a nondeterministic one.

5. The scheduling does not change the global heap state. Under 'scheduling' we understand solely the process of selecting an active thread. It does not have side effects. In combination with item 3, this means that the scheduler can be modeled as a mathematical function on the global state.

6. The scheduler is fair. By 'fair' we understand the property that every thread will be chosen sufficiently often to terminate—or infinitely often—cf. [Francez, 1986]. Given the finiteness assumption from above, an equivalent phrasing is that any thread is selected at least once within a finite time frame. Similar assumptions are, for instance, made by Stølen [1991]. This does not seem to bar us from modeling real world schedulers. As mentioned by Beckert and Klebanov [2013], Java schedulers are "statistically fair," which means effectively fair in almost any practical situation. From a theoretical point, the fairness assumption makes validity definitions simpler and more consistent. Taking the possibility into account that an interleaving may not return, would effectively introduce a kind of indeterminism.[5]

3.2 Target Programing Language

In this section, we introduce our target programing language *deterministic While-Release-Fork* (dWRF).[6] The sequential language constructs are assignments, conditional branching, and conditional loop statements. Additional constructs for concurrency are forking and release statements. Programs are sequences of statements. The (mathematical) integers and boolean are the only data type for program variables. Expressions can be of types integer or boolean; they do not have side effects. Integer operators are unary minus, addition, multiplication, division and modulo. The program language does not contain features such as functions and arrays; and there are no

[5]Dropping fairness would require to relax the semantics of properties on programs, leading only to partial correctness. An approach would be to introduce a special program 'state' that is unreachable and in which any formula is vacuously true. Such a definition would be very disturbing to our logic as there cannot be a regular program state with this properties. It would have to be treated explicitly in every definition. The logic of Beckert and Bruns [2013] is particularly well-behaved because of the (one) modality being dual to itself (i.e., it is invariant under negation). Such a property would be lost.

[6]It is pronounced [dwɔːɹf].

object-oriented features. The only special feature is the distinction between local variables (written in lowercase letters) and global variables (written in uppercase). We assume that local variable names are unique; in particular, there are no name clashes between threads.

Program expressions are typed. We use pairwise disjoint types $\mathbb{Z}$ (integers) and $\mathbb{B}$ (boolean). We assume disjoint infinite sets *LVar* of local program variables and *GVar* of global program variables to be given. These sets are universal; programs cannot declare variables.

Definition 3.1 (Program expressions)**.** Program expressions of type $\mathbb{Z}$ are constructed as usual over integer literals, local and global variables, and the operators `+`, `-`, `*`, `/`, and `%`. Program expressions of type $\mathbb{B}$ are constructed using the relations `==`, `>`, and `<` on integer expressions, the boolean literals `true` and `false`, and the logical operators `&&`, `||`, and `!`. A program expression is *simple* if it does not contain global variables.

As will be explained in Sect. 3.4, we consider assignments to global variables to be the only program statements that (locally) lead to a new observable state. To ensure that there cannot be a program that gets stuck in an infinite loop without ever progressing to a new observable state, we demand that every loop contains an assignment to a global variable. This technical restriction can easily be fulfilled by adding ineffective assignments to an unused global variable; we use the special statement `skip` as syntactic sugar for this. Expressions on the right hand side of global assignments and conditions for `if` or `while` statements must be simple. The right hand side of local assignments may refer to at most one global variable.[7]

We extend the core language introduced by Beckert and Bruns [2013] with two additional statements `release` and `fork` that represent explicit thread release and thread creation, respectively. By instrumenting a sequential program with the `release` statement, we simulate interleavings in a concurrent program, as explained in Sect. 3.6. A `fork` itself does not introduce interleavings, it merely updates the thread pool.

Definition 3.2 (dWRF syntax)**.** A *statement* is one of the following:

- ***local assignment***: `v =` x`;` where `v` is a local variable and x is an expression of the same type not containing reference to more than one global variable
- ***global assignment***: `F =` x`;` where `F` is a global variable and x is a simple expression of the same type
- ***conditional***: `if (`b`) {`π_0`} else {`π_1`}` where b is a simple boolean expression and π_0, π_1 are programs

[7] A similar requirement is imposed in the system by Manna and Pnueli [1995].

- ***loop***: `while (`b`) {`π`}` where b is a simple boolean expression and π is a program containing at least one global assignment
- ***thread release***: `release;`
- ***thread creation***: `fork {`π`};`

Definition 3.3. A *sequential program*, or just 'program' for short, is a finite sequence of statements. The set of sequential programs is denoted by Prg. Programs not containing `release` are called *noninterleaved*. A *concurrent program* Π is a finite set of sequential programs, $\Pi \in 2^{Prg}_{\text{fin}}$.

Programs are static entities, that must not be confused with threads, that only exist at runtime. Sequential programs serve as templates for threads and, conversely, threads have associated sequential programs. Different threads may be associated to sequential programs that are equivalent up to variable renaming.

Table 3.1 summarizes the syntax of sequential programs. The language of Beckert and Bruns [2013] is not strictly included in this, as it permits nonsimple expressions to appear as guards or the right hand side of global assignments. Nevertheless, any Beckert and Bruns [2013] program can be transformed into an equivalent program in the intersection of the languages by adding local assignments; see Lemma 3.9.

Table 3.1: Syntax of sequential dWRF programs. Local and global program variables are represented by rules `v` and `G`, respectively.

z	::=	z`+`z | z`-`z | z`*`z | z`/`z | z`%`z | `v` | `G` | `0` | `1` | ...
b	::=	`true` | `false` | b `&&` b | b `||` b | `!`b | z `==` z | z `>` z | z `<` z | `v` | `G`
x	::=	z | b
π	::=	`G =` x | `v =` x | π`;`π | `if (`b`) {`π`} else {`π`}` | `while (`b`) {`π`}` | `release` | `fork {`π`}` | `skip`

Example 3.4. The following line shows a small (noninterleaved) program, that reads two integers from global variables `A` and `B` and writes the minimum to a third global variable `C`.

```
x = A; y = B; if (x < y) { C = x; } else { C = y; }
```

The following, shorter, line is not a valid dWRF program since the statements in the conditional and on the right hand side of the global assignments to `C` are not simple. Yet, both are equivalent in the language presented by Beckert and Bruns [2013].

```
if (A < B) { C = A; } else { C = y; }
```

3.3 Representing Memory and Threads

We now lay the foundations for defining a semantics for dWRF. There are essentially two possibilities of representing computer memory in semantics and logic:[8] 1. to represent each memory location by a function symbol[9] or 2. to use a dedicated theory of storage and update a special variable representing the current memory state [McCarthy, 1962].

While Hoare logic and classical dynamic logic [Harel, 1979] pursue the former approach and use function symbols for each memory location,[10] the concept of having just one mathematical object to represent the whole memory of a computer system has been proven to be more convenient in may regards. It does allow to specify dynamically allocated memory—in particular recursively defined data structures—in a modular way; and it allows to specify information flow properties conveniently [Scheben and Schmitt, 2012a]. Instead of enumerating all the locations that are unchanged, we can just quantify over them. In Chap. 5, we show that with an explicit heap we can express two-state invariants conveniently. Such explicit heap modeling appears in [Poetzsch-Heffter and Müller, 1999; Stenzel, 2005; Barnett et al., 2005; Smans et al., 2008; Leino, 2010; Leino and Rümmer, 2010; Schmitt et al., 2011]. Specifications based on explicit heap notions are particularly valuable when reasoning about object-based structures, as the above reference suggest, but can also be applied to a setting with only global variables and no notion of objects.

To represent the heap on the semantical level, we introduce special program variables `heap` and `heap'`, that represent the current heap and the previous heap, respectively. These variables must not appear in programs.[11] Their semantics is a partial function from global variables to values. Upon every state change, induced by a write action, the values of `heap` and `heap'` are updated.

Reasoning about heaps is provided through the explicit heap theory of Schmitt et al. [2011]; Weiß [2011], that is already implemented in the KeY verification system from version 2.0 onwards. It compromises of the three data types Field (representing the syntactical entity of the same name in program code, denoted by $\mathbb{F}$), LocSet (representing finite sets of locations, denoted by $\mathbb{L}$), and Heap (representing a mapping from the set of all locations

[8] The reader may excuse that this section anticipates logic to some extend, that is meant to appear in Chap. 4. On the other hand, the logic representation is strongly related to the modeling issues that are discussed here.

[9] Although it may sound confusing, program variables are considered (nonrigid) constants, i.e., 0-ary functions, in this context.

[10] This approach was also taken in earlier versions of the KeY system, see [Beckert, 2001; Beckert et al., 2007b].

[11] They can be described as ghost variables, thus.

to values, denoted by $\mathbb{H}$). For most of this dissertation, we identify the terms 'location' and 'field' with each other, since we do not have the notion of objects.[12]

The field data type contains only a finite number of constants. The signature of the location set data type is the same as for standard (finite) sets; it includes the constructors empty set $\dot{\varnothing}$ and singleton $\{\ell\}$ (with a location ℓ), the binary set operators $\dot{\cup}$ (union), $\dot{\cap}$ (intersection), and $\setminus$ (set minus); as well as the unary set operator $\cdot^{\complement}$ representing the complement in the set of all locations.[13] The predicate $\dot{\in}$ indicates whether a field is in a location set. For convenience, we write $\{x_0, x_1, \ldots, x_n\}$ as shorthand for $\{x_0\} \dot{\cup} \{x_1\} \dot{\cup} \ldots \dot{\cup} \{x_n\}$.

The heap data type is a coälgebraic data type[14] with a single observer (or, 'destructor') *select* and two elementary mutator functions *store* and *anon* whose semantics is given in terms of *select*s on them. This modeling is based on the theory of arrays by McCarthy [1962], that is widely used to represent memory in logic or functional programing.

(i) $\mathit{select}(h, \ell)$ of type $\top$, where h is term of type $\mathbb{H}$ and ℓ ('location') of type $\mathbb{F}$, representing value retrieval from a location;

(ii) $\mathit{store}(h, \ell, v)$ of type $\mathbb{H}$, where v is a term of any value type (e.g., integer) and h, ℓ as above, representing a state change; and

(iii) $\mathit{anon}(h, L)$ of type $\mathbb{H}$, represents a heap that is havocked on all location in the location set L, but agrees on h otherwise.

We do not give formal semantics for the LocSet and Heap theories here as they are part of common folklore and should be intuitively clear. The nevertheless interested reader is pointed to [Weiß, 2011, Chap. 5]. By abuse of notation, we write logic symbols and their semantical counterpart functions alike. See also Sect. 8.2.2 on the role of the location set data type in Java Modeling Language (JML) specifications.

Threads are also represented by semantical objects. The state of currently active threads is recorded in a special variable `threads`. Like `heap`, it must not appear in programs. It is updated whenever a fresh thread is forked. It is of type $\mathbb{T}$, which elements are to be understood as finite, nonempty sets of threads. We assume set theoretical operators present, equivalent to the ones for $\mathbb{L}$ introduced above, that we denote with the same symbols. Like

[12] Weiß [2011] defines a location as a pair of a receiver object and a field (which is just an identifier). We will adopt the latter definition temporarily for Sect. 6.5, where we deal with object-sensitive information flow.

[13] The set of all locations is a welldefined finite set in this setting since there are only finitely many field constants. In general, the set of locations may not be finite.

[14] See the introduction in Sect. 8.2.2.

for the above theories, we refrain from overloading this section with formal semantics.

3.4 Trace Semantics for Sequential Programs

In this section, we develop semantics for noninterleaved sequential dWRF programs without dynamic thread creation. Instead of defining semantics as a relation between initial and final states of an abstract execution of the program (like by Beckert [2001], for instance), we use complete traces of intermediate program states. This will be extended to noninterleaved programs containing `fork` in Sect. 3.5.1 and to interleaved sequential programs in Sect. 3.5.2.

Expressions and formulae are evaluated over traces of states (that give meaning to program variables) and variable assignments (that give meaning to logical variables). The *domain*, denoted by $\mathcal{D}$, contains all semantical values to which an expression can evaluate. It does not depend on the program state (i.e., a *constant domain*). The domain can be partitioned into $\mathcal{D}_T$ for a type T.[15] All theories have the usual semantics. In particular the domain of integer expressions is $\mathbb{Z}$ and the domain of location set expressions consists of sets of locations: $\mathcal{D}_{\mathbb{L}} \subseteq 2^{\mathcal{D}_{\mathbb{F}}}$.

In addition to the sets $LVar$ and $GVar$, we introduce the disjoint set of 'special variables' $SVar := \{\texttt{heap}, \texttt{heap'}, \texttt{estp}, \texttt{threads}\}$ that do not appear in programs, but only in semantics. We refer to non-special variables as 'proper' variables. The variables `heap`, `heap'`, and `threads` have been introduced in the previous section, representing the current and previous global memory as well as the current thread pool. The fourth special variable `estp` of type boolean is used to distinguish between steps of the thread under investigation and environment steps (in interleaved programs). It is explicitly set on every step: to `true` for an environment step and to `false` for a local step.

Definition 3.5. A *program state*—or simply *state* for short—is a function $s : LVar \cup SVar \to \mathcal{D}$ assigning values to program variables. It assigns integer or boolean values to all proper local variables of the appropriate type (i.e., $s|_{LVar} : LVar \to \mathcal{D}_{\mathbb{Z}} \cup \mathcal{D}_{\mathbb{B}}$), a heap function to the special variables `heap` and `heap'`(i.e., $s|_{\{\texttt{heap},\texttt{heap'}\}} : SVar \to \mathcal{D}_{\mathbb{H}}$), a boolean value to the special variable `estp` (i.e., $s|_{\{\texttt{estp}\}} : SVar \to \mathcal{D}_{\mathbb{B}}$), and a set of threads to the special variable `threads` (i.e., $s|_{\{\texttt{threads}\}} : SVar \to \mathcal{D}_{\mathbb{T}}$).

Instead of the usual mathematical notation $s(x)$ for function application, we will frequently use the notation x^s, that is common in logic texts. We use the notation $s\{x \mapsto d\}$ to denote the state that is identical to s except that the variable x is assigned the value $d \in \mathcal{D}$, formally $s\{x \mapsto d\} = \{x \mapsto$

[15] Remember that we do not have subtypes in dWRF.

$d\} \cup \{y \mapsto s(y) \mid y \in LVar \cup SVar \setminus x\}$. Likewise, we write $\tau\{x \mapsto d\}$ (where τ is a trace, see below) with the obvious semantics. For global program variables, the special variable `heap` is updated to a new function using the (higher order) function *store*, see Sect. 3.3.

Definition 3.6 (Traces)**.** A *computation trace*, or just *trace* for short, τ is a non-empty, finite or infinite sequence of (not necessarily different) states. The set of traces is denoted by $\mathcal{S}^*$.

We use the following notations related to traces:

- $|\tau| \in \mathbb{N} \cup \{\infty\}$ is the *length* of a trace τ. If $\tau = \langle s_0, \dots s_k \rangle$, then $|\tau| = k + 1$.
- $\tau_1 \cdot \tau_2$ is the *concatenation* of traces:
  - If $|\tau_1| = \infty$, then $\tau_1 \cdot \tau_2 = \tau_1$.
  - If $\tau_1 = \langle s_0, \dots, s_k \rangle$ (finite) and $\tau_2 = \langle t_0, \dots \rangle$ (possibly infinite), then $\tau_1 \cdot \tau_2 = \langle s_0, \dots, s_k, t_0, \dots \rangle$.
- $\tau[i, j)$ for $i, j \in \mathbb{N} \cup \{\infty\}$ is the *subtrace* beginning in the i-th state (inclusive) and ending before the j-th state:
  - If $i \geq |\tau|$ or $i \geq j$, then $\tau[i, j) = \tau$
  - If $i < |\tau| < j$, then $\tau[i, j) = \tau[i, |\tau|)$
  - If $\tau = \langle s_0, \dots, s_i, s_{i+1}, \dots, s_{j-1}, s_j, \dots \rangle$, then $\tau[i, j) = \langle s_i, s_{i+1}, \dots, s_{j-1} \rangle$ for $j < \infty$ and $\tau[i, \infty) = \langle s_i, s_{i+1}, \dots \rangle$.
- $\tau[i]$ for $i \in \mathbb{N}$ is the state at position i in τ (with $\tau[i] := \tau[0]$ for $i \geq |\tau|$). For convenience, we identify singleton traces with their sole element.

Computation traces of programs are defined through small step denotational semantics on observable states (cf. [Scott and Strachey, 1971; Reynolds, 1998]). As mentioned above, we consider assignments to global variables to be the only sequential statements that lead to a new observable state. By specifying which variables are local and which are global, the user can thus determine which states are 'interesting' and are to be included in a trace. For the feasibility of proving properties about dWRF programs, it is important that not too many irrelevant intermediate states are included in a trace.

Definition 3.7 (Trace of a noninterleaved program (without fork))**.** Given an initial state s, the trace of a noninterleaved program π, denoted $trc_\Sigma(s, \pi)$, is defined by (the greatest fixpoint of) the following equations:

$$
\begin{aligned}
&trc_\Sigma(s, \epsilon) && := \langle s \rangle \\
&trc_\Sigma(s, \texttt{x = a; } \omega) && := trc_\Sigma(s\{x \mapsto a^s\}, \omega) \\
&trc_\Sigma(s, \texttt{X = a; } \omega) && := \langle s \rangle \cdot trc_\Sigma\left(s \left\{ \begin{array}{l} \texttt{estp} \mapsto \mathit{false}, \\ \texttt{heap'} \mapsto \texttt{heap}^s, \\ \texttt{heap} \mapsto \texttt{heap}^s\{\texttt{X} \mapsto a^s\} \end{array} \right\}, \omega \right) \\
&trc_\Sigma\left(s, \begin{array}{l} \texttt{if (a) \{}\pi_1\texttt{\}} \\ \texttt{else \{}\pi_2\texttt{\}} \ \omega \end{array} \right) && := \begin{cases} trc_\Sigma(s, \pi_1\ \omega) & \text{if } s \vDash a \\ trc_\Sigma(s, \pi_2\ \omega) & \text{if } s \nvDash a \end{cases} \\
&trc_\Sigma(s, \texttt{while (a) \{}\pi\texttt{\}}\ \omega) && := \begin{cases} trc_\Sigma(s, \pi\ \texttt{while (a) \{}\pi\texttt{\}}\ \omega) & \text{if } s \vDash a \\ trc_\Sigma(s, \omega) & \text{if } s \nvDash a \end{cases}
\end{aligned}
$$

where ϵ is the empty program and ω is a program.

The scheduling function Σ (see Sect. 3.5.2 below) does not have an effect on this definition. We will omit Σ whenever it is not relevant to the context. The definition of a program trace for noninterleaved programs will be completed with Def. 3.10 below. See also Example 3.8.

Remark. Typically, program semantics are defined inductively in terms of (sets of) reachable terminal states (i.e., big step semantics), cf. Beckert et al. [2007b, Sect. 3.3]. Opposed to this, our definition is *coïnductive*. It is based on traces of all reachable states (i.e., small step semantics). Traces may be of infinite length. To cater for this, the semantics is defined through the *greatest* fixpoint of trc_Σ instead of the least fixpoint. The coïnductive notion is motivated by our goal to express trace properties about nonterminating programs.

Example 3.8. We look again on the program from Example 3.4. It reads integers from two global variables and writes the minimum to another global variable.

```
x = A; y = B; if (x < y) { C = x; } else { C = y; }
```

Let s be a state with $\texttt{heap}^s = \{\texttt{A} \mapsto 5, \texttt{B} \mapsto 7, \texttt{C} \mapsto -1\}$. Figure 3.2 on the next page shows the concrete intermediate states that the execution passes through when started in s. Not all these states are included in the trace of the program, but only the two states before the global assignment (3) and the terminal state (5). All other states are equivalent to one of them regarding the value of `heap`.

The following lemma states that the syntactical restrictions regarding global variables imposed on programs (see Def. 3.2) do not lessen expressivity. Further, it allows us to view dWRF as a *conservative* extension [Shoenfield, 1967; Kaufmann and Moore, 1999] to sequential programs.

Lemma 3.9. *Let π be a noninterleaved 'program' without* `fork` *and with the restrictions on global variables waved, i.e., a program of Beckert and Bruns [2013]. 1. There is a proper dWRF program π' that is semantically equivalent to π, i.e., $trc(s, \pi) = trc(s, \pi')$ for all $s \in \mathcal{S}$, where trc is the*

```
                heap ↦ {A ↦ 5, B ↦ 7, C ↦ −1}                          (0)
x = A;
                heap ↦ {A ↦ 5, B ↦ 7, C ↦ −1}, x ↦ 5                   (1)
y = B;
                heap ↦ {A ↦ 5, B ↦ 7, C ↦ −1}, x ↦ 5, y ↦ 7            (2)
if (x < y) {
                heap ↦ {A ↦ 5, B ↦ 7, C ↦ −1}, x ↦ 5, y ↦ 7            (3)
  C = x;
                heap ↦ {A ↦ 5, B ↦ 7, C ↦ 5}, x ↦ 5, y ↦ 7, estp ↦ false (4)
} else {
  C = y; }
                heap ↦ {A ↦ 5, B ↦ 7, C ↦ 5}, x ↦ 5, y ↦ 7, estp ↦ false (5)
```

Figure 3.2: The intermediate states of a program execution are shown on the right. The states shown in red are included in the program trace.

function introduced in [Beckert and Bruns, 2013, Def. 8]. 2. The definitions of traces trc and trc_Σ restricted to local and global variables are equivalent.

Proof. Ad 1: We show the lemma by structural induction over π. The base case is the empty program. For the step case, assume that for any proper subprogram π_i of π, there is an equivalent dWRF program π_i'. We have to distinguish between the different kinds of statements:

- $\pi = \texttt{v = } x\texttt{;}\ \pi_2$: Let x be representable as a function $f_x(\texttt{G}_1, \ldots, \texttt{G}_n)$ where $\texttt{G}_j$ are the global variables in x for some $n \in \mathbb{N}$. Let $\tilde{\pi}_x := \texttt{v}_1\texttt{ = G}_1\texttt{; } \ldots \texttt{ v}_n\texttt{ = G}_n\texttt{;}$ where the $\texttt{v}_j \in \mathit{LVar}$ are fresh. Then we define $\pi' := \tilde{\pi}_x\ \texttt{v = } f_x(\texttt{v}_1, \ldots, \texttt{v}_n)\texttt{;}\ \pi_2'$. Let $\tilde{s} := s\{\texttt{v}_j \mapsto \texttt{G}_j^s \mid 0 < j \leq n\}$. Since the $\texttt{v}_j$ are fresh, for all expressions y that do not contain any $\texttt{v}_j$, it is $y^s = y^{\tilde{s}}$. It is obvious to see that it follows from the definition of *trc* for local assignments that the traces of π and π' are the same.
- $\pi = \texttt{F = } x\texttt{;}\ \pi_2$: as above.
- $\pi = \texttt{if (}b\texttt{) \{}\pi_0\texttt{\} else \{}\pi_1\texttt{\}}\ \pi_2$: Let b be representable as a function $f_b(\texttt{G}_1, \ldots, \texttt{G}_n)$. Then $\pi' := \tilde{\pi}_b\ \texttt{if(}f_b(\texttt{v}_1, \ldots, \texttt{v}_n)\texttt{)\{}\pi_0'\texttt{\}else\{}\pi_1'\texttt{\}}\pi_2'$, where $\tilde{\pi}_b$ uses the 'tilde' notation from above. Again, the trace equality is obvious.
- $\pi = \texttt{while (}b\texttt{) \{}\pi_0\texttt{\}}\ \pi_2$: Let everything be as above. Then $\pi' := \tilde{\pi}_b\ \texttt{while(}f_b(\texttt{v}_1, \ldots, \texttt{v}_n)\texttt{)\{}\pi_0'\ \tilde{\pi}_b\texttt{\}}\pi_2'$. We prove the trace equality; it is $trc(s, \pi') = trc(\tilde{s}, \texttt{while(}f_b(\ldots)\texttt{)}\ \ldots)$ and $b^s = (f_b(\vec{\texttt{v}}))^{\tilde{s}}$. If $s \vDash b$, then $trc(s, \pi') = trc(\tilde{s}, \pi_2')$ and we are done. If $s \nvDash b$, then $trc_\Sigma(s, \pi') = trc(\tilde{s}, \pi_0'\tilde{\pi}_b\ \texttt{while}\ \ldots)$. Assume $trc(\tilde{s}, \pi_0') = trc(s, \pi_0')$ is finite with final state $\bar{s}$. Then $trc(s, \pi') = trc(\bar{s}, \tilde{\pi}_b\ \texttt{while}\ \ldots) = trc(\tilde{\bar{s}}, \texttt{while}\ \ldots)$ and the fixpoint theorem closes the proof.

Ad 2: The only item to differ between [Beckert and Bruns, 2013, Def. 8] and Def. 3.7 is the global assignment. The 'old' *trc* can be emulated by trc_Σ through restriction to `heap` and mapping to global variables. ◁

3.5 Semantics of Concurrent Programs

Above in Def. 3.7, we have given a semantics for the purely sequential part of dWRF. In this section, we define semantics for interleaved programs and in turn for concurrent dWRF programs. Remember that interleavings are made explicit in the program code. We first extend our definition of program traces for noninterleaved programs, from Def. 3.7 above, to programs that contain `fork` statements, but not `release`, in Sect. 3.5.1. Then, in Sect. 3.5.2, we develop a semantical model of interleavings, which gives rise to a semantics of `release`.

At runtime, a concurrent program Π is identified with a set $\mathcal{T}$ of *threads* that can be created and a (fair) *scheduling function* Σ. Every thread $t \in \mathcal{T}$ has an associated sequential program π_t, which syntactically is one of the members of the concurrent program Π, modulo renaming of local variables. Without loss of generality, we assume that local variables are unique to one thread. This means, in particular, that no two threads have the same program. It is reasonable to define that $\mathcal{T}$ contains an infinite number of isomorphic copies of each sequential programs as a reservoir. We can think of the syntactical appearance of sequential programs as templates for these copies. Keeping a reservoir from which fresh objects can be selected, instead of actually 'creating' new ones through an extension of the state, is a common modeling technique in the context of dynamic logic (cf. [Beckert et al., 2007b, Sect. 3.6.6]). It permits us to work with the constant domain assumption.[16] We refer to the pair $(\mathcal{T}, \Sigma)$ as a *concurrent system*.

3.5.1 Dynamic Thread Creation

Besides the memory state as introduced above, concurrent programs also have a *thread state*. We will refer to the set $T \subseteq \mathcal{T}$ of currently alive threads as the *thread pool*. Through dynamic thread creation, the thread pool may change throughout program execution. In any reachable program state, a thread pool is finite and nonempty. Syntactically, we represent the thread pool by the special variable $\texttt{threads} \in SVar$.

Definition 3.10 (Computation trace of a noninterleaved program)**.**
Let everything be as in Def. 3.7. Additionally, we define the following:

$$trc_\Sigma(s, \texttt{fork \{}\pi\texttt{\}; } \omega) := trc_\Sigma(s\{\texttt{threads} \mapsto \texttt{threads}^s \cup t\}, \omega)$$

[16] For an alternative model of object creation with a nonconstant domain, see [Ahrendt et al., 2009].

where $t \in \mathcal{T}$ is a fresh thread with program π (modulo renaming of local variables).

Example 3.11. Consider the following simple, forking program:

```
A = 4; fork { A = 7; }
```

The trace consists of two states: an initial state s and a final state s' with $\texttt{heap} \mapsto \texttt{heap}^s\{\texttt{A} \mapsto 4\}$ and $\texttt{threads} \mapsto \texttt{threads}^s \cup \{t\}$ where t is a fresh thread with the program `A = 7;` The effect of the forked thread t is not included in this trace as it only contains the local changes.

We will complete the definition of program traces for interleaved programs in Def. 3.17 below, where we add the definition of trc_Σ applied to a `release` statement, after having developed a semantics for interleavings.

3.5.2 Interleaved Programs

Following our assumptions in Sect. 3.1.2, the scheduler Σ is a mathematical function—deterministic and without side effects. Depending on the state, it chooses a thread from the thread pool. Without loss of generality, we assume that any run of a concurrent program never reaches the exact same state s twice,[17] except for the special case that all threads in $\texttt{threads}^s$ have terminated. The axiom of choice [Zermelo, 1904] guarantees that schedulers do actually exist.

Definition 3.12. A *scheduler* is a function $\Sigma : \mathcal{S} \to \mathcal{T}$ such that $\Sigma(s) \in \texttt{threads}^s$ for any state s.

To define the big step semantics of concurrent programs, we need to define a total state transition function σ, that also takes into account dynamic thread creation. The thread-local transition function σ_t is equivalent to the computation trace of the noninterleaved program π_t. We can safely assume exactly one, definite trace here since a sequential program is deterministic.

Definition 3.13. Let t be a thread with noninterleaved sequential program π_t. Let the trace for π_t be given as $\langle s_0, s_1, \dots \rangle$. The *thread-local state transition* function $\sigma_t : \mathcal{S} \to \mathcal{S}$ maps any nonterminal state s_i to its successor s_{i+1} and a terminal state to itself in a single step.

The definition of σ is welldefined since we assume that no state is visited twice. We now take the environment into consideration; we assume that it is also deterministic (i.e., it contains other deterministic sequential programs that are executed according to a deterministic scheduler).

Definition 3.14.

[17] Otherwise, we would have to model a program counter. So far, we assume that it is encoded in the state.

1. For a concurrent system $(\mathcal{T}, \Sigma)$ the *system state transition* function $\sigma_\Sigma : \mathcal{S} \to \mathcal{S}$ denotes a single step of the concurrent system, with $\sigma_\Sigma(s) := \sigma_t(s)$ where $t = \Sigma(s)$.

2. The iterated extension of the transition function σ_Σ with $n \in \mathbb{N}$ is defined inductively as $\sigma_\Sigma^0(s) := s$ and $\sigma_\Sigma^{n+1}(s) := \sigma_\Sigma\left(\sigma_\Sigma^n(s)\right)$.

3. The set $\Omega_t \subseteq \mathcal{S}$ contains the terminal states for thread t, i.e., Ω_t is the set of fixpoints of σ_t.[18] Ω_T denotes the set of terminal states for all threads t in a thread pool T, i.e., $\Omega_T = \bigcap_{t \in T} \Omega_t$.

The following definitions are only welldefined for fair schedulers, following our assumptions in Sect. 3.1.2. We define fairness as the property that for every point in time, every alive and not yet terminated thread is scheduled within finite time. For nonterminating threads this means being scheduled infinitely often. In case all threads have terminated, any thread may be chosen ad infinitum.

Definition 3.15 (Fairness)**.** A scheduler Σ is *fair* if for every thread pool T and every thread $t \in T$ there is exists an $n \in \mathbb{N}$ such that $\Sigma\left(\sigma_\Sigma^n(s)\right) = t$, for any nonterminal state $s \in \mathcal{S} \setminus \Omega_t$.

Definition 3.16 (Macro step)**.** Let Σ be a fair scheduler. The state transition function $\sigma_\Sigma^* : \mathcal{S} \times \mathcal{T} \to \mathcal{S}$ describes the *macro step* between two states in which either there exists a thread $t \in \texttt{threads}^s$ that is active or all threads have terminated. Formally: $\sigma_\Sigma^*(s, t) := \sigma_\Sigma^n(s)$ where n is the smallest natural number such that $\Sigma\left(\sigma_\Sigma^n(s)\right) = t$, or $\sigma_\Sigma^n(s) \in \Omega_T$ with $T = \texttt{threads}^{\sigma_\Sigma^n(s)}$.

The state transition function $\sigma_\Sigma^*(s, t)$ describes the state change induced by the environment, that occurs in between atomic steps of a thread t under investigation. Within the transition $\sigma_\Sigma^*(t)$, there is no transition of thread t. But in the final state of $\sigma_\Sigma^*(s, t)$, the scheduler selects t again, as displayed in the example in Fig. 3.3 on the facing page. The fairness assumption guarantees that this minimum actually exists. In the special case that all threads have terminated, the scheduler may select any thread. In this case, $\sigma_\Sigma^*(t)$ is the identity function for any thread t. Note that we do not need to specify program pointers/active statements in the other threads; this is already encoded in the program state s.

Following these definitions, the thread pool T has an influence on the computation trace of a program, and thus on the definition of validity. We extend Defs. 3.7 and 3.10 with the `release` statement, defined through the macro step function σ_Σ^*; thereby effectively providing a semantics for interleaved sequential programs. The `release` statement is the second statement (the other being global assignment) that induces a step in the trace,

[18] This is sufficient for termination since we assume no state to be repeated.

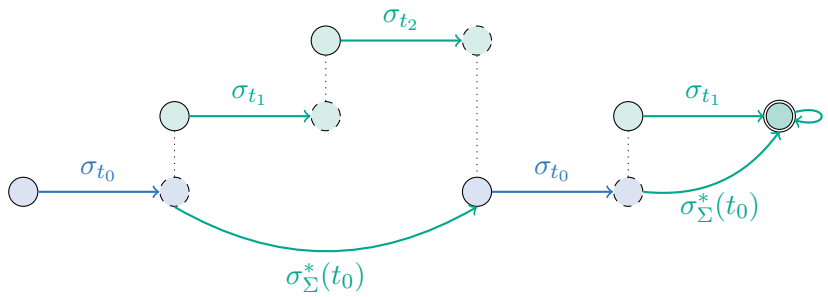


Figure 3.3: A system trace for three threads t_0, t_1, t_2. Two atomic steps σ_{t_1} and σ_{t_2} are combined into a macro step $\sigma^*_\Sigma(t_0)$. Another macro step points to the terminal state represented by the dark node on the right.

following our fundamental idea of having a step when 'something interesting' happens. To further distinguish steps of the thread under investigation (i.e., induced by an assignment statement in its program) and environment steps induced by `release` statements, we encode this in the special variable `estp`. The previous heap state is stored in the special variable `heap'`. This is established by the update to the result of σ^*_Σ.

Definition 3.17 (Computation trace of an interleaved program)**.**
Let everything be as in Defs. 3.7 and 3.10, but let additionally Σ be a fair scheduler. We additionally define

$$trc_\Sigma(s, \texttt{release;}\ \omega) := \langle s \rangle \cdot trc_\Sigma\left(\sigma^*_\Sigma(s, \Sigma(s))\left\{\begin{array}{l}\texttt{heap'} \mapsto \texttt{heap}^s, \\ \texttt{estp} \mapsto \mathit{true}\end{array}\right\}, \omega\right) \quad .$$

Note that the macrostep σ^*_Σ is defined through local transitions of the environment, and thus only affects `heap`, but not `heap'` or `estp`. This definition of a trace is similar to what Xu et al. [1997] call a "computation," that distinguishes between component and environment state transitions, on the one hand. On the other hand, they model concurrency as indeterminism, allowing any enabled transition to be taken, while our definition is based on deterministic program semantics.

Example 3.18. We consider the program from Example 3.11 again, this time with added `release` statements.

```
s0  A = 4;  s1  fork { A = 7; release; } release;  s2
```

The trace of the local thread t now consists of three states—as opposed to only two in Example 3.11. Given an initial state s_0, the intermediate state s_1 is produced by the assignment to `A` and includes the valuations $\texttt{heap} \mapsto \texttt{heap}^{s_0}\{\texttt{A} \mapsto 4\}$ and $\texttt{estp} \mapsto \mathit{false}$ as before. The state reached after the `fork` statement is not on the trace. Finally, there is a terminal state s_2 on the trace that is induced by the final `release` statement. Assuming that

the environment is empty in the beginning, s_2 holds the following mappings: $\texttt{heap} \mapsto \texttt{heap}^{s_0}\{\texttt{A} \mapsto 7\}$, $\texttt{threads} \mapsto \{t, t'\}$ (where t' is the thread that is being forked), and $\texttt{estp} \mapsto \mathit{true}$. In this case, the final `release` of the local thread brings in exactly the heap update of the forked thread t'. The `release` in the program of t' has no effect since there is no further local transition of t.

3.5.3 Properties of Program Traces

The above definition of program traces establishes that the special variable `heap'` refers to the previous state heap and `estp` faithfully indicates that an environment step occurred or not (in a non-trivial trace produced by a valid program):[19]

Lemma 3.19. *Let π be a sequential program with a trace $\tau = trc_\Sigma(s, \pi)$ of length $|\tau| \geq 2$. Then for all $i \in (0, |\tau|)$, the following hold:*

1. $\texttt{heap}'^{\tau[i]} = \texttt{heap}^{\tau[i-1]}$ *and*
2. $\texttt{estp}^{\tau[i]} = \mathit{true}$ *if and only if* $\tau[i] = \sigma^*_\Sigma(\tau[i-1], \Sigma(\tau[i-1]))$.

Proof. Follows from Defs. 3.7, 3.10, and 3.17 by structural induction over programs. ◁

The following definitions and lemmas are required for reasoning about single threads in a concurrent system.

Definition 3.20 (System trace; projection)**.** Let $(\mathcal{T}, \Sigma)$ be a concurrent system. Let $s \in \mathcal{S}$ be a state. 1. The *system trace* $trc_\Sigma(s)$ is the smallest repetition-free prefix of the sequence $\langle \sigma^i_\Sigma(s) \rangle_{i \in \mathbb{N}}$. 2. If $\tau = trc_\Sigma(s)$ is a system trace and $t \in \texttt{threads}^{\sigma^*_\Sigma(s)}$ is a thread, then $\tau{\downarrow_t}$ denotes the projection to t:

$$\left\langle \tau[i] \mid i \in [0, |\tau|) \wedge s' = \sigma^i_\Sigma(s) \wedge T' = \texttt{threads}^{s'} \wedge \left(\Sigma(s') = t \vee s' \in \Omega_{T'}\right) \right\rangle$$

The projection function is not the inverse to the system trace construction. The projection of the system trace $trc_\Sigma(s){\downarrow_t}$ to a thread t is similar to the local interleaved trace of the thread $trc_\Sigma(s, \pi_t)$, as it includes local steps and the effects of interleavings, but not the steps induced by interleavings. Another way to express this is, that the interleaved trace $trc_\Sigma(s, \pi_t)$ is a contraction of the system trace, fusing all (zero or more) environment steps into one `release` step. This leads us to the following lemma:

[19] This statement is not valid for all traces, since there are traces that are not induced by a program.

Lemma 3.21. *Let $(\mathcal{T}, \Sigma)$ be a concurrent system. Let s be a state and $t \in \texttt{threads}^s$. The projection $trc_\Sigma(s)\downarrow_t$ of the system trace to t contains all states s_i of $trc_\Sigma(s, \pi_t)$ for which $\texttt{estp}^{s_i} = true$ or $s_i \in \Omega_t$ holds in the original order.*

Proof. For the first case on s_i: according to Lemma 3.19, the condition $\texttt{estp}^{s_i} = true$ is a precise characterization of the 'own' steps of t. For the second case, $s_i \in \Omega_t$ follows from $s_i \in \Omega_{T'}$ according to Def. 3.14. The preservation of order follows from Def. 3.20. ◁

If a concurrent system terminates, i.e., all threads terminate, including those created during the run, then there is a unique 'idle' state that is the terminal state for all threads. This is enshrined in the following lemma.

Lemma 3.22. *Let $(\mathcal{T}, \Sigma)$ be a concurrent system. Let s be a state. If $\tau = trc_\Sigma(s)$ is finite with length $n+1$, then $\tau[n]$ is the final state in $trc_\Sigma(s, \pi_t)$ for every $t \in \texttt{threads}^{\sigma^n_\Sigma(s)}$.*

Proof. Definition 3.20(1) defines the system trace as finite if and only if there is a terminal state for the system. The definition of transition functions state that a terminal state s' is a fixpoint.[20] From Def. 3.16 it follows that is also a fixpoint for σ^*_Σ: If it were not, then there must be an active thread t' with $\sigma_{t'}(s') = \sigma^{n+1}_\Sigma(s)$. Since s' is already final, it must be $\sigma^{n+1}_\Sigma(s) = s'$. By Def. 3.20(2), s' is included in the projection $\tau\downarrow_t$. By Lemma 3.21, the final state is also included in the local trace $trc_\Sigma(s, \pi_t)$. ◁

3.6 Modeling Concurrent Programs

In this section, we discuss how the dWRF language, that was introduced in this chapter, can model 'real' concurrency as in the Java language. Basically, we follow the usual approach in which the execution of concurrent programs is mimicked by interleaved sequential programs (cf. [de Roever et al., 2001, Chap. 3]). However, we refrain from introducing parallel composition operators in favor of `fork` and `release` constructs. As mentioned above, the idea of explicit thread release points is borrowed from the Creol [Dovland et al., 2005] and ABS [Johnsen et al., 2010] languages. The `release` statements indicates a program point where the environment, i.e., concurrently running threads, may change the global memory, while it must not do elsewhere. This paradigm is helpful to define the language of concurrent programs as a *conservative* extension to sequential programs: the semantics of read actions from the global memory can be retained; all concurrent changes are encapsulated in `release`. Creol and ABS additionally feature a conditional release statement `await` b, that we do not incorporate into dWRF.

[20] Note that $\Sigma(s') = t$ is not necessarily true, however.

Explicit release does not follow the usual concurrency paradigm, where environment changes may occur at any program point (except in atomic blocks).[21] Of course, we do not expect a programmer to type out `release`; we only consider it an instrument to conveniently define semantics. The transformation of a program without explicit release, but in the usual interleaving semantics, to a program with explicit release is straightforward: a `release` is to be inserted at any point where an interleaving may occur. In this way, we construct sequential programs that simulate the *observable* runtime behavior of concurrent programs. Although in a real concurrent program the environment may be active at *any* point in time, with this definition, we restrict it to fewer states. The rationale behind this is to already keep to model simple enough to efficiently reason about. This is justified—on the meta level—by the observation that only some interleavings are actually observable to the thread under investigation.

An interleaving is only observable if has an effect on 1. the control or data flow in the program or 2. an assertion stated about the program. In our language, control statements are atomic and free of side effects. Furthermore, we separate read and write actions from/to the global memory; there is no atomic update of the heap. This coïncides with the interleaving model in Java [Gosling et al., 2014] where compound statements such as `X++` are not atomic, but can be broken up into a read and write action each.[22]

Item 1 in this setup means that concurrent changes only affect *read* actions. Therefore, it suffices to insert one `release` immediately before such statements. Note that write actions and control statements are never directly influenced by environment actions. This is obvious, given our restrictions in Def. 3.2 on the occurrence of global variables. In the actual concurrent behavior that is to be modeled, the order of writes may technically be different, but the observable effect of a write action is always the same. Any concurrent write action that is not modeled by our interleaving semantics is eventually shadowed by a write action of 'our' thread.

To simulate the concurrent behavior, we can instrument noninterleaved sequential programs with `release` statements and amend the invariant rules. Basically, a noninterleaved program is instrumented in a way such that environment actions appear before every heap read and the termination action as follows.

[21] Note that the concept of an atomic block is not present in the Java language.

[22] We consider Java at the *source code level* with the operational semantics provided by Gosling et al. [2014], in comparison to Bytecode. A strict correspondence of dWRF statements to Java Bytecode instructions is coïncidental. The number and order of Bytecode instruction corresponding to source code statement is not fixed either, as compilers enjoy certain degrees of freedom.

Definition 3.23 (Instrumented program)**.** Let $\pi = \langle stm_1, \ldots, stm_n \rangle$ be a noninterleaved program. The corresponding *instrumented program* $\underline{\pi}$ is constructed following: For each statement stm_i with $i \in (0, n]$,

- if stm_i is a local assignment with a nonsimple expression on the right hand side, the statement `release;` is inserted before stm_i,
- if $stm_i =$ `while (`b`) {`π'`}`, it is replaced by `while (`b`) {`$\underline{\pi'}$`}`
- if $stm_i =$ `if (`b`) {`π_1`} else {`π_2`}`, it is replaced by `if (`b`) {`$\underline{\pi_1}$`} else {`$\underline{\pi_2}$`}`,
- and the statement `release;` is inserted after stm_n.

Please note that this instrumentation is purely syntactic and independent of a thread pool. Later, we expect that programs under investigation are already instrumented.

Item 2 of the postulate above is more intricate. First of all, 'assertions' refers to the trace properties that appear after program modalities in formulae (see the following chapter). As terminal execution states are always included in traces, it demands that an interleaving after the (locally) final state of a trace has an effect. A particular class of these properties are postconditions. But secondly, interleavings also affect any property stated on *subtraces*. Fortunately, the calculus we present in the following chapters, does not allow arbitrary trace decompositions, since program rules for dynamic logic do (usually) focus on active statements. This is in contrast to other program logics, in particular Hoare [1969] logics, that have sequential decomposition rules.[23]

The only exception are the invariant rules (see Tab. 4.7 on page 73), that state properties about the subtrace induced by the loop body. While the original invariant rules are technically sound w.r.t. the semantics defined in this chapter, they are inappropriate to model symbolic execution of concurrent programs. The reason is that the loop invariant may depend on shared locations that are modified by the environment. We present slightly modified invariant rules, that only differ in additional instrumentation, in Tab. 4.11 on page 84.

[23] Confer to [Beckert et al., 2007b, p. 115] for a discussion on this.

4 Concurrent Dynamic Trace Logic

Dynamic logic is an established instrument for program verification and for reasoning about the semantics of programs and programing languages. It benefits from a high flexibility—in particular, dynamic logic can readily express relational properties such as noninterference. Most dynamic logics, however, consider only sequential programs. In this chapter, we develop a novel dynamic logic that targets threads of concurrent programs. Although, we define our logic for the simple language dWRF, which was introduced above, we argue that it can be extended to full Java in a natural way.

In contrast to standard dynamic logic, which is entirely state-based, we use a notion of program semantics based on traces of program states. In the previous chapter, we have introduced dWRF, a simple programing language with basic support for multithreading. We have defined a trace-based semantics for dWRF, including an interleaving semantics w.r.t. a deterministic scheduler. In prior work [Beckert and Bruns, 2013], we have defined Dynamic Trace Logic (DTL), that combines the expressiveness of program logics such as first order dynamic logic with that of temporal logic. In this chapter, we define a dynamic logic for dWRF, extending DTL to Concurrent Dynamic Trace Logic (CDTL). We recapitulate the base calculus for DTL from Beckert and Bruns [2013], to be extended to full CDTL in Chap. 5.

The KeY verification system (co-developed by the author; see also Sect. 7.1) is built on a calculus for Java dynamic logic (JavaDL), a dynamic logic for sequential Java [Beckert, 2001; Beckert et al., 2007b]. As these features are mostly orthogonal to those discussed in this chapter, the JavaDL calculus can be used as a basis to extend CDTL to Java. We have implemented the CDTL calculus for Java prototypically in KeY. Additional rules needed to handle full (sequential) Java can be derived from the KeY rules for the $[\cdot]$ modality by analogy. Since a language like Java incorporates a lot of features, in particular object-orientation, and various syntactic sugars, the rule set is quite voluminous (c. 1600 rules) in comparison to simple while

languages. These special cases can, however, be reduced to a smaller set of base cases. For instance, the assignment `x=y++` containing a postincrement operator is transformed into two consecutive assignments `x=y` and `y=y+1` during symbolic execution.[1] This becomes particularly relevant as soon as we take interleavings into consideration. See also Chap. 7 for details on rule implementations in the KeY prover.

Chapter Overview

In this chapter, we conservatively extend the logic of our prior work—that only considers sequential programs—with concurrent dWRF programs as defined in Chap. 3. In this way, we produce CDTL, a dynamic logic with program modalities that still contain sequential programs, but which actually represent threads of concurrent programs. We present a sound and (relatively) complete sequent calculus for proving validity of core DTL formulae. Later, in Chap. 5, we show that this calculus can be extended to a sound calculus for full CDTL, based on the rely/guarantee formalism. In Chap. 7, we discuss how this calculus can be implemented in the KeY system in order to verify programs in the Java language.

Section 4.1 covers the state of the art in the areas of mathematical logic that is being used in this chapter—in particular regarding program logics (that include dynamic logic). This is a prerequisite for defining syntax (Sect. 4.2) and semantics (Sect. 4.3) of CDTL. The remainder of the chapter is dedicated to proving valid propositions of *sequential* DTL as presented previously [Beckert and Bruns, 2013]. We present a calculus in Sect. 4.4 and prove it sound and complete in Sect. 4.5. The concluding discussion in Sect. 4.6 contains an example proof and an outlook to the following chapters.

4.1 Logic Background

In this chapter, we define a variation of first order dynamic logic that can be used to express properties about the concurrent programing language defined in Sect. 3.2 above. We start by reviewing the foundations of mathematical logic.

First Order Logic

First order predicate logic, or just FOL for short, is a very expressive logic that features quantifiers; see, e.g., [Hilbert and Ackermann, 1949] for a historical or [Ebbinghaus et al., 1994; Fitting, 1996] for a modern overview. A famous example is the axiomatization of set theory by Zermelo and Fraenkel [1908], which uses only plain FOL plus a defining 'element-of' operator. Models of

[1] This is what actually happens inside the Java Virtual Machine, cf. [Lindholm et al., 2014, Sect. 3.11].

FOL consist of variable assignments and interpretations of predicates and functions. Church and Turing [1936] have proven that it is undecidable in general whether sentences of FOL are valid.[2] This effectively means that a sound procedure that decides validity automatically and completely cannot exist. There are, however, sound and complete calculi for FOL that are not automatic and require interaction. This property is sometimes called *semi-decidability* to distinguish FOL from higher order logics (HOLs), which is even 'more undecidable' not having a sound and complete calculus. Still, some common properties cannot be expressed in FOL, such as a binary relation being the transitive closure of another. Traditional definitions of predicate logic are built on a homogeneous universe of elements and characterize elements through predicates. For more practical considerations, it is helpful to have *types* in the logic [Schmitt and Ulbrich, 2014], e.g., integers, in particular when reasoning with certain theories.

4.1.1 Modal Logic

Modal logics (cf. [Stirling, 1992] for an overview) introduce the concept of a *state* (or 'world'). Models for modal logics consist of multiple states, that all hold a characteristic interpretation and variable assignment. A state itself is a propositional logic or FOL structure. These states are connected through transitions, forming *Kripke structures* [Kripke, 1963]. There are many different modal logics; but they all have two basic modal operators $\Box$ ('box') and $\Diamond$ ('diamond') in common, that intuitively mean 'for all transitions' or 'for some transition.' Although (propositional or first-order) modal logics can be encoded in first-order logic by axiomatizing the transitions, it is convenient to use these intuitive constructs. One important aspect is that many propositional modal logics are decidable.

Temporal Logic

Temporal logics are a particular class of modal logics, where Kripke structures usually form directed acyclic graphs, representing time. Probably the most well known temporal logic is Linear Temporal Logic (LTL) [Manna and Pnueli, 1995], where the Kripke structure forms a single linear trace of infinite length. Modalities have the meaning 'always in the future' ($\Box$) or 'eventually' ($\Diamond$), but the binary modal operator U ('until'), that is more expressive than $\Box$ and $\Diamond$, is also frequently used. The additional binary operators R ('release') and W ('weak until') can be derived from U. In many definitions, LTL is further extended with a 'next' operator $\bullet$ (sometimes X). These temporal operators are concerned with *future* states. There are extensions of LTL using *past* operators, such as 'once' or 'since.' However, they do not introduce any

[2] Decidable fragments are identified, for instance, by Davis [2004] or van Benthem [2005].

additional expressive power (cf. [Benedetti and Cimatti, 2003]). Lichtenstein et al. [1985] define a variant of LTL with finite structures.

Validity in propositional LTL is decidable (in exponential time).[3] The most common verification technique associated with LTL is model checking [Clarke et al., 1986]. LTL model checkers construct a Büchi automaton [Büchi, 1962] that is equivalent to the formula's model (treating variable assignments as words of an alphabet). Büchi automata accept ω-regular languages.

However, not every ω-regular language has an accepting LTL formula. A well known example of a property that is not expressible in LTL is the 'two-step' property: a formula φ holds in every second state of a temporal temporal structure,[4] cf. [Wolper, 1981, Corollary 4.2]. The Interval Temporal Logic (ITL) by Moszkowski [1985]; Cau, Moszkowski, and Zedan [2002] has this kind of expressiveness. A framework to enrich LTL with arbitrary temporal operators has been presented by Wolper [1981].

The temporal logics Computation Tree Logic (CTL) [Clarke and Emerson, 1981] and CTL* [Emerson and Sistla, 1984] are based on branching structures and feature quantification over paths. The Alternating Time Logic (ATL) [Alur et al., 2002] subsumes CTL* and features explicit *actors* (or *strategies*) that determine certain paths. The modal μ-calculus by Kozen [1983] is even more expressive. It uses the higher-order concept of explicit fixpoint operators. Modal μ-calculus subsumes CTL* [Dam, 1994]. Hodkinson and Reynolds [2007] provide a comprehensive overview over temporal logics.

4.1.2 Dynamic Logic

Dynamic logic (DL) [Fischer and Ladner, 1979; Harel, 1979, 1984] is a family of multi-modal logics where each legal sequential program fragment π of a given language gives rise to modal operators $[\pi]$ ('box') and $\langle\pi\rangle$ ('diamond'). The formula $[\pi]\,\varphi$ expresses 'in any state in which π terminates, φ holds,' while the dual $\langle\pi\rangle\,\varphi$ expresses 'there is a state in which π terminates and φ holds in that one.' In this context, the formula φ is called a postcondition. Modal tautologies like $\neg\,[\pi]\,\varphi \leftrightarrow \langle\pi\rangle\,\neg\varphi$ are still valid. If programs are deterministic—i.e., there is at most one final state—the modality $\langle\cdot\rangle$ is a variant of $[\cdot]$ that demands termination. Programs in languages like Java are deterministic in the sense that, under some assumptions about the environment (e.g., the presence of unlimited memory), the program represents a partial function from one system state to another. Postconditions including termination of programs are known as total correctness properties, as opposed to partial correctness.

Propositional dynamic logic of regular languages (PDL) is decidable (in exponential time) [Fischer and Ladner, 1979], but not PDL with parallel

[3] First order LTL is obviously in the same class of decidability as classical FOL.

[4] The related 'φ holds *exactly* in every second state' property is expressible.

composition [Balbani and Tinchev, 2014]. Since we consider a first-order version of dynamic logic—and programs are structured, i.e., using deterministic looping, instead of regular programs—the logic becomes as undecidable as FOL (plus arithmetic) itself. In particular, the halting problem is a proposition of first-order dynamic logic.

The above mentioned early works on dynamic logic are based on elementary notions of programs, usually involving (non-deterministic) unconditional branching and looping. In contrast, the dynamic logics used in practice in program verification are much more involved as they capture the semantics of real-world languages. JavaDL is an instance of dynamic logic [Beckert, 2000, 2001], that is tailored to a substantial subset of sequential Java.[5] In particular, it involves deterministic program semantics. One drawback, however, is the sheer size of the rule base that is necessary. The implementation of JavaDL in the KeY system involves some 1,600 rules. Furthermore, there is no formal official semantics for Java, that would ultimately justify soundness of these rules.

Dynamic logic formulae are related to the weakest precondition calculus [Dijkstra, 1975] and Hoare logic [Hoare, 1969, 1972]. For a FOL formula φ, the DL formula $\langle\pi\rangle\,\varphi$ represents the weakest precondition of π w.r.t. the postcondition φ. A Hoare triple $\{\psi\}\pi\{\varphi\}$ (under total correctness semantics) is equivalent to the DL formula $\psi \to \langle\pi\rangle\,\varphi$. This formula is valid if and only if ψ is not stronger than the weakest precondition $\langle\pi\rangle\,\varphi$. However, dynamic logic is more expressive than Hoare logic in that programs are part of formulae. This allows universal or existential quantification to range over the state transition induced by the program and to have formulae have multiple modalities. This allows to express relational properties on programs, e.g., noninterference [Scheben and Schmitt, 2012a].

In other regards, however, standard dynamic logic lacks expressiveness: The semantics of a program is a relation between states; formulae can only describe the input/output behavior of programs. Standard dynamic logic cannot be used to reason about program behavior not manifested in the input/output relation. It is inadequate for reasoning about nonterminating programs and for verifying temporal properties.

Updates

State updates are explicit state transition operators (i.e., a third class of modalities) in an extension of dynamic logic [Rümmer, 2006] [Beckert et al., 2007b]. Updates allow symbolic forward execution [King, 1976] starting from an initial state—in contrast to weakest precondition where one starts with a postcondition and works in a backwards manner. In order to capture the state

[5] This original definition by Beckert [2001] captures JavaCard, a minimal dialect of Java that is used on smart cards. The JavaCard language is covered completely by the logic [Mostowski, 2006, 2007b].

transitions in between, we use state updates. One way to intuitively think of updates is to regard them as 'delayed substitutions,' i.e., a substitution takes place once the program has been completely eliminated. For instance, $\{v := 4\}$ and $\{v := v + 1\}$ are (elementary) updates. Applying these updates sequentially (i.e., after each other, from right to left) to the formula $v \doteq 5$ yields $4 + 1 \doteq 5$. In general, the parallel composition operator $||$ allows swapping of variables without the use of intermediate variables. For instance, $\{x := y \,||\, y := x\}x \doteq y + 1$ simplifies to $y \doteq x + 1$. The update simplification calculus by Rümmer [2006] brings dynamic logic formulae into a normal form: program modalities are prefixed with only one non-clashing parallel update.[6] This normal form corresponds to the well-known static single assingment (SSA) form [Cytron et al., 1989], that is used in program transformations.

4.1.3 Dynamic Trace Logic

Dynamic Trace Logic (DTL) [Beckert and Bruns, 2013] is a marriage of dynamic and temporal logic. It allows to express temporal properties about programs. There is only one program modal operator, called *trace modality* $[\![\cdot]\!]$. We distinguish between state formulae and trace formulae. A *state formula* consists of the usual propositional and first order constructs plus subformulae of the form $\mathcal{U}[\![\pi]\!]\varphi$ where $\mathcal{U}$ is a sequence of updates, π is a program, and φ is a *trace formula* (that may contain temporal operators and further subformulae of the same form). Intuitively, $\mathcal{U}[\![\pi]\!]\varphi$ expresses that φ holds when evaluated over all traces τ such that the initial state of τ is (partially) described by $\mathcal{U}$ and the further states of τ are constructed by running the program π. Since we have deterministic programs, traces are determined by their initial states. However, states are symbolic (i.e., possibly underspecified), and thus are traces.

In addition to propositional operators and quantification, trace formulae may contain temporal operators similar those in LTL: unary operators $\Box$ ('always') and $\Diamond$ ('eventually'), and binary operators U ('until'), W ('weak until'), and R ('release') with the obvious semantics. Since traces may be finite or infinite, there are weak ($\bullet$) and strong 'next' ($\circ$) operators, that are dual to each other (cf. [Lichtenstein et al., 1985]). For example, the formula $\bullet \mathit{false}$ (with 'weak next') holds exactly in the final state of a trace and never else. The formula $\Diamond\bullet \mathit{false}$ then expresses termination. 'Strong next' additionally mandates the existence of a next state. The addition of a 'strong next' operator is the only difference to LTL operators. A formula is called *nontemporal* if it neither contains a temporal operator nor a program modality $[\![\pi]\!]$.

Since programs are included in formulae of DTL, we can have both state and trace formulae in a sequent at the same time—and even formulae

[6] The semantics of parallel updates that do clash is defined as 'last wins.'

expressing how different traces of programs relate to each other. This allows to express information flow properties by stating that the traces of some program π that result from different secret inputs are sufficiently similar in order to keep secret information inobservable during program execution.

Standard dynamic logic is covered by DTL because the semantics of the standard $[\cdot]$ and $\langle\cdot\rangle$ modalities can be expressed in DTL: The formula $\bullet\mathit{false}$ holds exactly on a trace with only one (remaining) state, thus characterizing termination. We are therefore able to represent $[\pi]\,\varphi$ ('if π terminates, then φ holds') by $[\![\pi]\!]\Box(\bullet\mathit{false} \rightarrow \varphi)$ and $\langle\pi\rangle\,\varphi$ ('π terminates and then φ holds') by $[\![\pi]\!]\Diamond(\bullet\mathit{false} \wedge \varphi)$. Yet, both can still coëxist and we will use the $[\cdot]$ notation later.

Theories

First order defineable theories can be used to write down concise and intuitively understandable properties. First order dynamic logic with uninterpreted functions is already very expressive. However, it is convenient to add dedicated theories to lift the burden of axiomatizing commonly used functions over and over again. Even though theories usually encode higher order properties, they are axiomatizable in first order logic. In the following, we use theories for finite sequences [Beckert et al., 2013b, Appendix A] and heap memory (cf. Sect. 3.3).

4.2 Syntax of Concurrent Dynamic Trace Logic

In this section, we define the syntax of formulae in our target logic, Concurrent Dynamic Trace Logic (CDTL). It is a typed first order dynamic logic with dedicated theories that extends Dynamic Trace Logic (DTL) [Beckert and Bruns, 2012b, 2013]. Programs of the *deterministic While-Release-Fork* (dWRF) language, which we introduced in Chap. 3, give rise to modalities in CDTL.

Signatures and Expressions

In addition to program variables (cf. Sect. 3.2), there is a separate set V of logical variables. Logical variables are *rigid*, i.e., they cannot be changed by programs and—in contrast to program variables—are assigned the same value in all states of a program trace.[7] Logical variables must not occur in programs. Quantifiers can only range over logical variables and not over program variables.

[7] Rigid variables are essential to the expressiveness of the logic. Without them it would be impossible to compare values in different states. E.g., expressing '`X` has increased by 1' requires to introduce a rigid variable u which in every state evaluates to the prestate value of `X`.

Expressions are typed. We use pairwise disjoint types $\mathbb{Z}$ (integers), $\mathbb{B}$ (boolean), $\mathbb{H}$ (heaps), $\mathbb{L}$ (location sets), $\mathbb{F}$ (fields), $\mathbb{T}$ (thread pools), and $\mathbb{S}$ (sequences); cf. Sect. 3.3. There is a common supertype $\top$. Quantified formulae have the shape $\forall x{:}U.\ \varphi$ where U is one of the above types. If U is the supertype $\top$, it is omitted. For an expression x, its type is denoted by $\mathit{type}(x)$.[8] Both local and global program variables always have types $\mathbb{Z}$ or $\mathbb{B}$.

Functions have signatures $A_1 \times \cdots \times A_n \to B$ where all A_i and B are types. A 0-ary function is called a *constant*. *Predicates* have signatures $A_1 \times \cdots \times A_n$, where $n = 0$ is allowed. Both functions and predicates are rigid. The sets of functions and predicates are denoted by $\mathcal{F}$ and $\mathcal{P}$, respectively. The set $\mathfrak{S} = \mathit{LVar} \cup \mathit{GVar} \cup \mathit{SVar} \cup V \cup \mathcal{F} \cup \mathcal{P}$ is called the *signature* of the logic.

In this chapter, the sets of function and predicate symbols are fixed. They contain the usual integer and boolean operators with their standard semantics and the theories of heaps (see Sect. 3.3) and final sequences.

Final sequences (i.e., tuples of arbitrary size) are represented by the algebraic data type $\mathbb{S}$. The constructors are $\langle\rangle$ (empty), $\langle\cdot\rangle$ (singleton) and $\oplus$ (concatenation). We use the two observer functions $|\cdot|$ (length) and $\cdot[i]$ (random access at position i, where i is an expression of type $\mathbb{Z}$; postfix operator). For longer sequences, we write $\langle x_0, x_1, \ldots, x_n\rangle$ as shorthand for $\langle x_0\rangle \oplus \langle x_1\rangle \oplus \cdots \oplus \langle x_n\rangle$. Cf. Sect. 8.2.2 for a short introduction to algebraic data types.

Definition 4.1 (Logic expressions)**.** Logic expressions of type $\mathbb{Z}$ are constructed as usual over integer literals, program variables, logical variables, and the operators $+$, $-$, $*$, $/$, $\%$. Expressions of type $\mathbb{B}$ are constructed using the relations $\doteq$, $>$, $<$ on integer expressions, the boolean literals *true* and *false*, the special variable `estp`, and the logical operators $\wedge$, $\vee$, $\neg$.

Expressions of type $\mathbb{S}$ are constructed using the operators $\langle\cdot\rangle$, $\oplus$, $|\cdot|$, and $\cdot[\cdot]$. Expressions of types $\mathbb{F}$, $\mathbb{L}$, $\mathbb{H}$, and $\mathbb{T}$ are constructed using the special variables `heap` and `heap'`; and the operators $\dot{\varnothing}$, $\{\cdot\}$, $\dot{\in}$, $\dot{\cap}$, $\dot{\cup}$, $\setminus$, $\cdot^{\complement}$, *select*, *store*, and *anon* as described above in Sect. 3.3. The ternary operator ite ('if-then-else') can be used with any type.

Integer and boolean logic expressions are constructed similar to their program expression counterparts (cf. Def. 3.1). They may additionally contain logical variables and 'special' variables. They must contain not global program variables. Instead, they may refer to the special program variable `heap`. The special variable `threads` does not appear in expressions. For a concise representation, we pairwise identify the literals and operators of program and logic expressions, e.g., the symbols `&&` and $\wedge$ denote the same operator. We display them in program style (using typewriter font) when they appear inside of programs and in math style when appearing outside.

[8] Later, we overload the function *type* to map semantical objects to their type.

Our logic uses updates [Rümmer, 2006] (as introduced above in Sect. 4.1) to keep track of the state of symbolic execution.

Definition 4.2 (State updates)**.** For $i \in [0, n]$, let x_i be a local program variable, and let a_i be an expression. Then, $\{x_0 := a_0 \mid\mid \ldots \mid\mid x_n := a_n\}$ is a *parallel update*. Let $\mathcal{U}_0, \ldots, \mathcal{U}_m$ be parallel updates, then the juxtaposition $\mathcal{U}_0 \ldots \mathcal{U}_m$ is a *sequential update*. *Update* means parallel or sequential update.

Formulae

CDTL formulae have the general appearance $\mathcal{U}[\![\pi]\!]\varphi$ where $\mathcal{U}$ is an update, π is a sequential dWRF program, and φ is a formula (that may or may not contain temporal operators and further sub-formulae of the same form). Intuitively, $\mathcal{U}[\![\pi]\!]\varphi$ expresses that φ holds when evaluated over all traces τ such that the initial state of τ is (partially) described by $\mathcal{U}$ and the further states of τ are constructed by running the program π.

Definition 4.3 (Formula)**.** *State formulae* and *trace formulae* are inductively defined as follows:

0. All boolean expressions (Def. 4.1) are state formulae.
1. All state formulae are also trace formulae.
2. If φ and ψ are (state or trace) formulae, then the following are trace formulae: $\Box\varphi$ ('always'), $\bullet\varphi$ ('weak next'), $\varphi \,\mathsf{U}\, \psi$ ('until').
3. If $\mathcal{U}$ is an update and φ a state formula, then $\mathcal{U}\varphi$ is a state formula.
4. If π is a sequential program (Def. 3.2) and φ a trace formula, then $[\![\pi]\!]\varphi$ is a state formulae.
5. The sets of state and trace formulae are closed under the logical operators $\neg$ (negation), $\wedge$ (conjunction), and $\forall$ (universal quantification).

In addition, we use the following abbreviations as syntactical sugars:

$$
\begin{array}{rcll}
\varphi \vee \psi & := & \neg(\neg\varphi \wedge \neg\psi) & \text{(disjunction)}, \\
\varphi \rightarrow \psi & := & \neg\varphi \vee \psi & \text{(implication)}, \\
\exists x.\varphi & := & \neg\forall x.\neg\varphi & \text{(existential quantification)} \\
\circ\varphi & := & \neg\bullet\neg\varphi & \text{('strong next')}, \\
\Diamond\varphi & := & \neg\Box\neg\varphi & \text{('eventually')}, \\
\varphi \,\mathsf{W}\, \psi & := & \varphi \,\mathsf{U}\, \psi \vee \Box\varphi & \text{('weak until')}, \\
\varphi \,\mathsf{R}\, \psi & := & \neg(\neg\varphi \,\mathsf{U}\, \neg\psi) & \text{('release')}.
\end{array}
$$

A formula is called *non-temporal* if it neither contains a temporal operator nor a program modality $[\![\pi]\!]$. A formula is a *strict* DTL formula if it contains only program modalities with noninterleaved programs.

Table 4.1: Syntax of Concurrent Dynamic Trace Logic. The program syntax (production rule π) can be found in Tab. 3.1 on page 37.

$$
\begin{array}{lcl}
\varphi & ::= & \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \forall x{:}T.\varphi \mid \exists x{:}T.\varphi \mid \mathcal{U}\varphi \mid [\![\pi]\!]\psi \mid b \\
\psi & ::= & \bullet\psi \mid \circ\psi \mid \Box\psi \mid \Diamond\psi \mid \psi \,\mathsf{U}\, \psi \mid \psi \,\mathsf{W}\, \psi \mid \psi \,\mathsf{R}\, \psi \mid \varphi \\
\mathcal{U} & ::= & \{\mathtt{v} := e\} \\
T & ::= & \mathbb{B} \mid \mathbb{Z} \mid \mathbb{H} \mid \mathbb{F} \mid \mathbb{L} \mid \mathbb{T} \mid \mathbb{S} \\
b & ::= & \mathit{true} \mid \mathit{false} \mid \mathtt{estp} \mid\ e \doteq e\ \mid\ z < z\ \mid\ \ldots \\
e & ::= & x \mid \mathit{select}(h, \mathtt{G}) \mid \mathrm{ite}(b, e, e) \mid \bot \mid m(e) \mid S[z] \mid z \mid h \mid L \mid S \\
z & ::= & z + z\ \mid\ z * z\ \mid\ -z \mid |S| \mid 0 \mid 1 \mid 2 \mid\ \ldots \\
h & ::= & \mathtt{heap} \mid \mathtt{heap'} \mid \mathit{store}(h, \mathtt{G}, e) \mid \mathit{anon}(h, L) \\
L & ::= & \dot{\varnothing} \mid \{\mathtt{G}\} \mid L \,\dot{\cup}\, L \mid L \,\dot{\cap}\, L \mid L \setminus L \mid L^{-1} \\
S & ::= & \langle e \rangle \mid S \oplus S
\end{array}
$$

A complete syntactical schema of the logic used in this work can be found in Tab. 4.1, while the program syntax appears in Tab. 3.1 on page 37.

4.3 Semantics of Concurrent DTL

To devise a semantics for CDTL, we extend our prior work on DTL to cater for concurrent dWRF programs. We use modalities of the shape $[\![\pi_t]\!]$ where π_t the program associated with a thread $t \in T$ under investigation, where T is the current thread pool state. The semantics of expressions—including the theories of sequences, location sets, and heaps—is standard. For the semantics of formulae, we incorporate the trace semantics of dWRF programs from Sect. 3.4 to obtain a semantics for program modalities. Below in Lemma 4.8, we will prove that these extensions to DTL are indeed conservative.

We consider assignments to global variables to be the only statements that lead to a new observable state on the trace. All other statements are atomic in this sense. For the feasibility of proving CDTL formulae, it is important that not too many irrelevant intermediate states are included in a trace. For instance, if a formula such as $[\![\pi]\!]\Box\varphi$ is to be proven valid, intermediate states require sub-proofs showing that φ holds in each of them.

Expression Semantics

As usual, the semantics of CDTL expressions is given through a variable assignment and an interpretation.

Definition 4.4 (Variable assignments)**.** A *variable assignment* β is a function assigning integer values to all logical variables, i.e., $\beta : V \to \mathcal{D}$. Similar to the notion for states, we write $\beta\{x \mapsto d\}$ for the updated variable assignment.

Definition 4.5 (Interpretation)**.** An *interpretation* I is a mapping of function symbols $f \in \mathcal{F}$ with signature $A_1 \times \cdots \times A_n \to B$ to a semantical function $I(f) : \mathcal{D}_{A_1} \times \cdots \times \mathcal{D}_{A_n} \to \mathcal{D}_B$ and of predicate symbols $p \in \mathcal{P}$ with signature $A_1 \times \cdots \times A_n$ to a relation $I(p) \subseteq \mathcal{D}_{A_1} \times \cdots \times \mathcal{D}_{A_n}$.

In the following, 'interpretation' refers to the standard interpretation for all aforementioned functions—following the definitions by Weiß [2011, Def. 5.4] (for dynamic logic with an explicit heap) and Beckert et al. [2013b, Appendix A] (for the theory of finite sequences), as well as the obvious interpretation for arithmetic functions.

Semantical functions are always total. Possible gaps, such as division or modulo by zero, are left underspecified [Gries and Schneider, 1995]. For instance, the expression $5/0$ has some fixed determined value, but we cannot draw any further conclusions from this, apart from the fact that the value is an integer (i.e., the domain of the total function represented by the / operator). For instance, $5/0 \doteq 5/0$ is universally valid, but $5/0 \doteq 3/0$ can neither be proved nor disproved. Underspecification is considered the prime choice to model undefinedness in logic [Hähnle, 2005], in particular because it allows to extend pure FOL conservatively.[9] The concurrent dynamic logic of Beckert and Klebanov [2013] also defines semantics through underspecification.

Remark. The tuple $(\mathcal{D}, I, s)$ consisting of the domain, an interpretation, and a state forms a *first-order structure*. In most works on classical first-order logic (and also mostly in general modal logics), the program variable assignment s forms part of the interpretation, with program variables being 0-ary functions. In the dynamic logic literature (cf. [Weiß, 2011]), however, it has recently become customary to separate those in order to distinguish between nonrigid (i.e., functions and predicates) and rigid (i.e., program variables) entities.

Definition 4.6 (Semantics of expressions)**.** Given a state s and a variable assignment β, the value $a^{I,s,\beta}$ of an expression a of type A in a state s is the value $d \in \mathcal{D}_A$ resulting from interpreting program variables x by x^s, logical variables u by u^β, and using the interpretation I for all functions and relations.

Since the interpretation I is assumed to be fixed in a structure, with the standard interpretations for usual function symbols, we usually omit I. Program expressions that do not contain logical variables are independent of β, and we write a^s instead of $a^{I,s,\beta}$. If a is a boolean expression, we write $I, s, \beta \models a$ resp. $s \models a$ to denote that $a^{I,s,\beta}$ resp. a^s is true.

Since the heap and sequence theories are built on (co)algebraic data types, it is trivial to give standard interpretations, yet not very instructive here. For reference, we provide an incomplete list of defining axioms in Tab. 4.2 on the next page. Complete accounts are provided by Weiß [2011] (on heaps,

[9] Other possibilities include 3-valued logics or partial logics [Schmitt, 2011, Sect. 2].

based on McCarthy [1962] theory) or Beckert et al. [2013b, Appendix A] (on sequences), respectively.

Table 4.2: Defining axioms for location set, heap, and sequence theories

$$select(store(h, X, a), Y) \doteq \text{ite}(X \doteq Y, a, select(h, Y)) \tag{4.1}$$

$$select(anon(h, L), X) \doteq \text{ite}(X \mathrel{\dot\in} L, sk, select(h, X)) \quad \text{where } sk \text{ is a fresh symbol} \tag{4.2}$$

$$\neq x \mathrel{\dot\in} \dot\varnothing \tag{4.3}$$

$$x \mathrel{\dot\in} \{x\} \tag{4.4}$$

$$x \mathrel{\dot\in} L_0 \mathbin{\dot\cap} L_1 \leftrightarrow x \mathrel{\dot\in} L_0 \wedge x \mathrel{\dot\in} L_1 \tag{4.5}$$

$$x \mathrel{\dot\in} L_0 \mathbin{\dot\cup} L_1 \leftrightarrow x \mathrel{\dot\in} L_0 \vee x \mathrel{\dot\in} L_1 \tag{4.6}$$

$$x \mathrel{\dot\in} L_0 \setminus L_1 \leftrightarrow x \mathrel{\dot\in} L_0 \wedge x \not\mathrel{\dot\in} L_1 \tag{4.7}$$

$$x \mathrel{\dot\in} L^{\complement} \leftrightarrow x \not\mathrel{\dot\in} L \tag{4.8}$$

$$|\langle\rangle| \doteq 0 \tag{4.9}$$

$$|s_0 \oplus s_1| \doteq |s_0| + |s_1| \tag{4.10}$$

$$0 \le i < |s_0| \rightarrow (s_0 \oplus s_1)[i] \doteq s_0[i] \tag{4.11}$$

$$|s_0| \le i < |s_0| + |s_1| \rightarrow (s_0 \oplus s_1)[i] \doteq s_1[i] \tag{4.12}$$

CDTL Formula Semantics

In Sects. 3.4f., we have defined semantics for dWRF programs based on traces of program states. States (Def. 3.5) are functions mapping program variables to values. Local variables are directly mapped to values of their respective type, the 'special' variables `heap` and `heap'` are mapped to functions themselves, mapping global variables to values. The function trc_Σ (Defs. 3.7, 3.10 and 3.17) assigns a trace to an initial state and a sequential program, w.r.t. a deterministic fair scheduler Σ (cf. Def. 3.15).

Remark. The tuple $(\mathcal{D}, I, \mathcal{S}, \rho)$ with a transition relation $\rho = \{(s, \pi, s') \in \mathcal{S} \times Prg \times \mathcal{S} \mid |trc_\Sigma(s, \pi)| > 1 \wedge trc_\Sigma(s, \pi)[1] = s'\}$ forms a standard Kripke structure [Kripke, 1963]. We do not use this notation here since it only relates initial and final states of an execution, but we are interested in all intermediate states.

We have now everything at hand needed to define the semantics of CDTL formulae in a straightforward way. The valuation of a state formula is given w.r.t. a state s and a variable assignment β; and the valuation of a trace formula is given w.r.t. a trace τ and a variable assignment β. This is expressed by the validity relation, denoted by $\vDash$. For the sake of uniformity, we do not distinguish between state and trace formulae here.

Definition 4.7 (Validity in CDTL)**.** Given a computation trace τ, variable assignment β, and fair scheduler Σ; the *validity* relation $\vDash$ is the smallest relation satisfying the following.

$$
\begin{array}{lll}
\tau,\beta,\Sigma \vDash a & \text{iff} & a^{\tau[0],\beta} = \mathit{true} \\
\tau,\beta,\Sigma \vDash \neg\varphi & \text{iff} & \tau,\beta,\Sigma \nvDash \varphi \\
\tau,\beta,\Sigma \vDash \varphi \wedge \psi & \text{iff} & \tau,\beta,\Sigma \vDash \varphi \text{ and } \tau,\beta,\Sigma \vDash \psi \\
\tau,\beta,\Sigma \vDash \forall u{:}U.\varphi & \text{iff} & \text{for every } d \in \mathcal{D}_U\colon\ \tau,\beta\{u \mapsto d\},\Sigma \vDash \varphi \\
\tau,\beta,\Sigma \vDash \Box\varphi & \text{iff} & \tau[i,\infty),\beta,\Sigma \vDash \varphi \text{ for every } i \in [0,|\tau|) \\
\tau,\beta,\Sigma \vDash \varphi \,\mathsf{U}\, \psi & \text{iff} & \tau[j,i),\beta,\Sigma \vDash \varphi \text{ and } \tau[i,\infty),\beta,\Sigma \vDash \psi \\
 & & \quad \text{for some } i \in [0,|\tau|) \text{ and all } j \in [0,i) \\
\tau,\beta,\Sigma \vDash \bullet\varphi & \text{iff} & \tau[1,\infty),\beta,\Sigma \vDash \varphi \text{ or } |\tau| = 1 \\
\tau,\beta,\Sigma \vDash \{x_1 := a_1 \mid\mid \ldots \mid\mid x_n := a_n\}\varphi & \text{iff} & \tau\{x_1 \mapsto a_1^{\tau[0]}\}\ldots\{x_n \mapsto a_n^{\tau[0]}\},\beta,\Sigma \vDash \varphi \\
\tau,\beta,\Sigma \vDash [\![\pi]\!]\varphi & \text{iff} & \mathit{trc}_\Sigma(\tau[0],\pi),\beta,\Sigma \vDash \varphi
\end{array}
$$

A formula φ is *valid*, written $\vDash \varphi$, if $\tau,\beta,\Sigma \vDash \varphi$ for all τ, β, and Σ.

In this definition, the scheduler Σ forms part of the validity relation. It entails an implicit universal quantification over all (fair) schedulers on the semantical level. A result of that is that our logic still uses only one kind of modality, that speaks about *the* deterministic trace. An alternative definition would be to introduce two modalities, that universally or existentially range over schedulers,[10] respectively.

Remark. Assume that the scheduler is *not* part of the validity relation, but the semantics of $[\![\cdot]\!]$ is defined w.r.t. all schedulers. Let us consider a dual modality $\langle\!\langle\cdot\rangle\!\rangle$ that is defined w.r.t. *some* scheduler. Consider a concurrent program with thread pool $T = \{t_0, t_1\}$ and $\pi_{t_0} =$ `X=0;` and $\pi_{t_1} =$ `X=1;`. Depending on the concrete scheduler, the final value of `X` can be either 0 or 1. Thus the formulae $\langle\!\langle\pi_{t_0}\rangle\!\rangle \circ \mathtt{X} \doteq 0$ and $\langle\!\langle\pi_{t_0}\rangle\!\rangle \circ \mathtt{X} \doteq 1$ are both valid, while neither $[\![\pi_{t_0}]\!] \circ \mathtt{X} \doteq 0$ nor $[\![\pi_{t_0}]\!] \circ \mathtt{X} \doteq 1$ is valid.

On first sight, our approach entails the obvious disadvantage that the semantics of a formula is defined in terms of *concrete* parallel programs. As a result, this definition is not modular. However, as we will see in Sect. 5.4.1, the rely/guarantee approach allows us to reason about interleavings symbolically, i.e., w.r.t. *any* environment.

On the other hand, our single modality is dual to itself and therefore exhibits some good properties. For instance, the formulae $\neg[\![\pi]\!]\varphi \leftrightarrow [\![\pi]\!]\neg\varphi$ or $[\![\pi]\!](\varphi_1 \vee \varphi_2) \leftrightarrow ([\![\pi]\!]\varphi_1 \vee [\![\pi]\!]\varphi_2)$ are tautologies.[11] This allows to give a smaller and more efficient calculus compared to a calculus that would have

[10] or traces, equivalently

[11] Another example of a modality with this property is the update operator of dynamic logic, if viewed as a modality.

to deal with two kinds of modalities,[12] as the results of Jeannin and Platzer [2014] suggest, for instance.

Remark. In this definition, formulae are always of the shape $[\![\pi]\!]\varphi$. This formula is valid if and only if φ is valid on any trace of π under any *deterministic* scheduler Σ. This is equivalent to φ being valid on any trace under any *indeterministic* scheduler. The reason is that any indeterministic scheduler can be simulated by a set of deterministic schedulers.

The logic CDTL presented here is a *semantical conservative extension* [Shoenfield, 1967; Bubel and Schmitt, 2016] of the base DTL logic presented by Beckert and Bruns [2013]. This allows us to adapt the base DTL calculus to a sound calculus for the sequential part of CDTL. It follows from Lemma 3.9 that the replacement of program modalities is welldefined.

Lemma 4.8 (Semantical conservative extension)**.** *Let φ be a valid formula according to [Beckert and Bruns, 2013, Def. 9]. Let φ' be the CDTL formula obtained from replacing all quantifiers by their $\top$-typed equivalent and all program modalities by a CDTL equivalent (i.e., restricting to simple expressions). Then φ' is a valid CDTL formula.*

Proof. The definitions of validity only differ in the additional scheduler parameter Σ, that affects solely the validity of program modalities. Lemma 3.9 shows that legal programs π of [Beckert and Bruns, 2013] have an equivalent trace in both definitions. Thus a modal formula $[\![\pi]\!]\varphi$ is valid in one definition if and only if it is valid in the other one. ◁

4.4 A Sequent Calculus for DTL

In this section, we describe the calculus for DTL with noninterleaved sequential programs as introduced in [Beckert and Bruns, 2013]. All the given rules extend naturally to interleaved programs. In Chap. 5, we extend the calculus to a calculus for CDTL, including rules to reason about environment actions and thread creation.

We present a sequent calculus [Gentzen, 1935; Hähnle, 2001]. We denote the calculus for DTL by $\mathcal{C}_{\text{DTL}}$. It is sound and relatively complete, i.e., complete up to the handling of arithmetic (see Sect. 4.5). The rules are numbered as in [Beckert and Bruns, 2013]. The calculus consists of the following rule classes:

Classical logic rules These rules simplify formulae whose top-level operator is a quantifier or a propositional operator (Sect. 4.4.1).

[12] As already discussed in Sect. 3.1.2, the possibility of an environment macro step of infinite length would require such a change.

Simplification and normalization rules Rules for simplifying formulae of the form $\mathcal{U}[\![\pi]\!]\varphi$, where the top-level operator in φ is not temporal (Sect. 4.4.2).

Rules for temporal operators Rules that apply to formulae $\mathcal{U}[\![\pi]\!]\varphi$ with a top-level temporal operator in φ, and that do not change the program π (Sect. 4.4.3).

Program rules Rules that apply to formulae of the form $\mathcal{U}[\![\pi]\!]\varphi$, and that analyze and/or simplify the program π. Not surprisingly, this class has the most complex rules, including invariant rules for loops (Sect. 4.4.4).

Rules for data structures Since our focus in this chapter is not on how to handle dedicated theories, we use oracle rules for these (Sect. 4.4.5).

Other rules This category includes the closure and the cut rule (Sect. 4.4.6).

Most rules of the calculus are analytic and therefore can be applied automatically. The rules that require user interaction are: (a) the rules for handling while loops (where a loop invariant has to be provided), (b) the cut rule (where the right case distinction has to be used), and (c) the quantifier rules (where the right instantiation has to be found).

Traces are uniquely determined by symbolic program executions of the deterministic programing language. The general idea behind our calculus is to explore a trace until it terminates or it reaches a fixpoint (induced by a non-terminating loop). Thus, proofs usually consist of alternating applications of temporal logic rules (that decompose trace formulae, e.g., $\Box\varphi$ to $\bullet\Box\varphi \wedge \varphi$) and program rules (that let us step forward in the trace). These steps are explicitly given through assignments in the program. Since traces are defined through program semantics, and they are of infinite length if and only if we go through an infinite loop, we either reach the end of the program or a computational fixpoint.

In the rule schemata, Γ, Δ denote arbitrary, possibly empty multi-sets of formulae, φ, ψ denote arbitrary formulae, $\mathcal{U}$ stands for a (possibly empty) update, π, ω for programs, γ is a state formula, `x` and `X` are local and global program variables, n and u are logical variables, a is an expression of type integer, and b is an expression of type boolean.

Definition 4.9 (Sequent)**.** A sequent is a pair of multi-sets of (state) formulae written as $\gamma_1, \ldots, \gamma_m \Longrightarrow \delta_1, \ldots, \delta_n$. The multi-set $\{\gamma_1, \ldots, \gamma_m\}$ of formulae on the left hand side of the sequent arrow $\Longrightarrow$ is called the *antecedent*, the set $\{\delta_1, \ldots, \delta_n\}$ is called the *succedent* of the sequent. We use capital greek letters to denote subsets of formula, e.g., the sequent notion $\Gamma, \varphi \Longrightarrow \psi, \Delta$ means that formulae φ and ψ occur in the antecedent or succedent and the sets of remaining formulae are Γ and Δ, respectively.

A sequent $\Gamma \Longrightarrow \Delta$ is *valid* (in state s and under variable assignment β) if and only if the formulae $\bigwedge_{\gamma\in\Gamma} \gamma \rightarrow \bigvee_{\delta\in\Delta} \delta$ is valid (w.r.t. s, β).

As usual, we write rules with schematic sequents above a horizontal line in a schema (its premisses) and a single schematic sequent below the horizontal line (its conclusion). Note, that in practice the rules are applied from bottom to top. Proof construction starts with the original proof obligation at the bottom. Therefore, if a constraint is attached to a rule that requires a variable to be 'new,' it has to be new w.r.t. the conclusion.

Definition 4.10 (Rule)**.** A *rule* consists of a finite set of schematic sequents $\Gamma_i \Longrightarrow \Delta_i$ called *premisses* and a sequent $\Gamma' \Longrightarrow \Delta'$ called *conclusion*. It is a *closing* rule if it has zero premisses. A rule is commonly written vertically with the premisses above the conclusion:

$$\frac{\Gamma_1 \Longrightarrow \Delta_1 \qquad \dots \qquad \Gamma_n \Longrightarrow \Delta_n}{\Gamma' \Longrightarrow \Delta'}$$

The *calculus* $\mathcal{C}_{\mathrm{DTL}}$ consists of the rules R1 to R35 shown in Tabs. 4.3 to 4.9 on pages 69–76.

Definition 4.11. A rule is *sound* if the conclusion is valid whenever all premisses are valid.

The following lemma helps proving soundness of rules. It states that the environments Γ and Δ do not need to be considered in most cases.

Lemma 4.12 (Omission of environments)**.** *A rule of the shape*

$$\frac{\Gamma, \Phi_1 \vdash \Psi_1, \Delta \qquad \cdots \qquad \Gamma, \Phi_k \vdash \Psi_k, \Delta}{\Gamma, \Phi \vdash \Psi, \Delta} \tag{4.13}$$

is sound if and only if the following rule is sound:

$$\frac{\Phi_1 \vdash \Psi_1 \qquad \cdots \qquad \Phi_k \vdash \Psi_k}{\Phi \vdash \Psi} \tag{4.14}$$

Proof. The one proof direction, from sequent (4.13) to (4.14), is trivial since it is a weakening. For the other direction, assume the sequents $\Gamma, \Phi_i \vdash \Psi_i, \Delta$ valid for all i. This means that $\Gamma \wedge \Phi_i \to \Psi_i \vee \Delta$ is valid. Assume $\Gamma \to \Delta$ invalid. (Otherwise the conclusion would be trivially valid.) This means that $\Phi_i \vdash \Psi_i$ is valid and from (4.14) it follows that $\Phi \vdash \Psi$ is valid. Since Γ is invalid and Δ is valid, the conclusion of (4.13) is a weakening of that sequent. ◁

An instance of a rule is called a rule application. Subsequent (finitely many) rule applications induce a graph, called *proof*, where each rule application is a node. The way we define rules in this dissertation provides that proofs are always finite trees with a determined root. We use the terminology *proof tree* synonymously with 'proof.' Proof *branches* are subtrees. A branch (or tree) is *closed* if all leaves (excluding the root) are instances of closing rules.

Definition 4.13. A sequent is *derivable* (with $\mathcal{C}_{\mathrm{DTL}}$) if it is an instance of the conclusion of a rule schema and all corresponding instances of the premisses of that rule schema are derivable sequents. In particular, all sequents are derivable that are instances of the conclusion of a rule that has no premisses (rules R22, R31, and R34).

4.4.1 Classical Logic

The first-order rules, i.e., rules for quantifiers and propositional operators are shown in Table 4.3. Note that the expressions that are used to instantiate universal quantifiers in rule R5 must be chosen in such a way that the substitution is admissible:

Definition 4.14 (Admissible substitution)**.** A substitution u/a of a logical variable $u \in V$ with an expression a is *admissible* w.r.t. a formula φ if there is no variable v in a such that u is free in φ and, after replacing a for some free occurrence of u in φ, the occurrence of v in a is (i) bound by a quantifier in $\varphi[u/a]$ (in case v is a logical variable) or is (ii) in the scope of a program modality $[\![\pi]\!]$ that contains an assignment to v (in case v is a program variable).

$$\frac{\Gamma \Longrightarrow \varphi, \Delta}{\Gamma, \neg\varphi \Longrightarrow \Delta}\ \mathsf{R1} \qquad \frac{\Gamma, \varphi \Longrightarrow \Delta}{\Gamma \Longrightarrow \neg\varphi, \Delta}\ \mathsf{R2}$$

$$\frac{\Gamma, \varphi, \psi \Longrightarrow \Delta}{\Gamma, \varphi \wedge \psi \Longrightarrow \Delta}\ \mathsf{R3} \qquad \frac{\Gamma \Longrightarrow \varphi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \varphi \wedge \psi, \Delta}\ \mathsf{R4}$$

$$\frac{\Gamma, \varphi[u/a], \forall u.\varphi \Longrightarrow \Delta}{\Gamma, \forall u.\varphi \Longrightarrow \Delta}\ \mathsf{R5} \qquad \frac{\Gamma \Longrightarrow \varphi[u/u'], \Delta}{\Gamma \Longrightarrow \forall u.\varphi, \Delta}\ \mathsf{R6}$$

Table 4.3: Rules for quantifiers and propositional operators. In rule R5, the substitution needs to be admissible; rule R6 introduces a fresh variable u'.

For example, using `X` to instantiate the universal quantifier in the DTL formula $\forall u.(u \doteq 0 \rightarrow [\![\texttt{X = 1;}]\!]\Box u \doteq 0)$ is not admissible. Indeed the result would be incorrect as the original formula is valid while the formula $X \doteq 0 \rightarrow [\![\texttt{X = 1;}]\!]\Box X \doteq 0$ is not even satisfiable.

We do not give rules for update simplification here. The reason is that they constitute an elaborate calculus on their own [Rümmer, 2006]. For the treatment in this dissertation it is sufficient to think of updates as substitutions that can only be applied once a program modality has been removed. In [Beckert and Bruns, 2013], we have given a simplified version of this calculus for the subcategory of sequential updates.

4.4.2 Simplification and Normalization Rules

As said above, our calculus contains simplification rules that apply to formulae of the form $\mathcal{U}[\![\pi]\!]\varphi$, where the top level operator in φ is not temporal. They are shown in Tab. 4.4 on the next page. In particular, they include normalization rules that deal with negated trace formulae through replacement by the respective dual formula.

Rule R12 for negated until avoids introducing the dual R into the sequent. Therefore, no rules for R are required in the calculus. Soundness of R12 follows from the well-known equivalence $\varphi \,\mathsf{R}\, \psi \leftrightarrow \psi \,\mathsf{W}\, (\varphi \wedge \psi)$ in LTL and the definitions of R and W, that applies to finite traces as well (cf. [Bauer, Leucker, and Schallhart, 2010]).

Lemma 4.15 ([Beckert and Bruns, 2012b, Appendix A, Lemma 7])**.** *Rule* R12 *('negation until') is sound.*

Proof. Assume the following sequent valid:

$$\Longrightarrow \mathcal{U}[\![\pi]\!]\Box\neg\psi, \mathcal{U}[\![\pi]\!](\neg\psi \,\mathsf{U}\, (\neg\varphi \wedge \neg\psi))$$

Let $\tau := trc(s^{\mathcal{U}}, \pi)$, where $s^{\mathcal{U}}$ denotes the state derived from s through the effects of $\mathcal{U}$. We make the following case distinction: (i) Assume $\tau \vDash \neg\psi \,\mathsf{U}\, (\neg\varphi \wedge \neg\psi)$. By the definition of U, there is an $j \in \mathbb{N}$ such that $\tau[j, \infty) \vDash \neg\varphi \wedge \neg\psi$ and $\tau[i, j) \vDash \neg\psi$ for every $i < j$. Now assume that $\tau \vDash \varphi \,\mathsf{U}\, \psi$; it requires some $k \in \mathbb{N}$ such that $\tau[k, \infty) \vDash \psi$, but from the above it follows that $k > j$ and $\tau[j, k) \nvDash \varphi$. (ii) Assume $\tau \vDash \Box\neg\psi$; it immediately follows that there is no subtrace τ' of τ such that $\tau' \vDash \psi$. It follows that the sequent $\Longrightarrow \mathcal{U}[\![\pi]\!]\neg(\varphi \,\mathsf{U}\, \psi)$ is valid. ◁

Since (for conciseness of the calculus) we only include program and temporal logic rules for the right hand side of a sequent, we need rule R13 that allows to move a formula with a modality from the left of a sequent to the right.

In case φ is a state formula, rule R16 can be used to remove the program modality (as a state formula is evaluated in the initial state of a trace). Further simplification rules are applied to split formulae such as $[\![\pi]\!](\Box\varphi \wedge \psi)$.

4.4.3 Rules for Temporal Operators

Table 4.5 on the facing page shows the rules that handle temporal operators without changing the program. Rules R19 to R21 'unwind' temporal formulae by splitting them into a 'future' part and a 'present' part. Rules R22 and R23 handle the case of an empty program (i.e., empty remaining trace) for weak and strong next, respectively. Rule R22 also closes a proof branch. Rule R23 is not necessary for a complete calculus, but we include it here in order to have rules for any kind of formula.

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\varphi,\ \mathcal{U}[\![\pi]\!]\psi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!](\varphi \vee \psi), \Delta}\ \mathsf{R9} \qquad \frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\varphi, \Delta \qquad \Gamma \Longrightarrow [\![\pi]\!]\psi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!](\varphi \wedge \psi), \Delta}\ \mathsf{R10}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\neg\varphi, \Delta}{\Gamma \Longrightarrow \neg\mathcal{U}[\![\pi]\!]\varphi, \Delta}\ \mathsf{R11} \qquad \frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\Box\neg\psi,\ \mathcal{U}[\![\pi]\!](\neg\psi\ \mathsf{U}\ (\neg\varphi \wedge \neg\psi)), \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\neg(\varphi\ \mathsf{U}\ \psi), \Delta}\ \mathsf{R12}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\neg\varphi, \Delta}{\Gamma, \mathcal{U}[\![\pi]\!]\varphi \Longrightarrow \Delta}\ \mathsf{R13} \qquad \frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\neg\neg\varphi, \Delta}\ \mathsf{R14}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\circ\neg\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\neg\bullet\varphi, \Delta}\ \mathsf{R15} \qquad \frac{\Gamma \Longrightarrow \mathcal{U}\gamma, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\gamma, \Delta}\ \mathsf{R16}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\varphi[u/u'], \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\forall u.\varphi, \Delta}\ \mathsf{R17} \qquad \frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\varphi[u/a],\ \mathcal{U}[\![\pi]\!]\exists u.\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\exists u.\varphi, \Delta}\ \mathsf{R18}$$

Table 4.4: Simplification and normalization rules. In rule R16, γ is a state formula. Rule R17 introduces a fresh variable u'; in rule R18, the substitution needs to be admissible.

$$\frac{\Gamma \Longrightarrow \mathcal{U}([\![\pi]\!]\circ(\varphi\ \mathsf{U}\ \psi) \wedge [\![\pi]\!]\varphi),\ \mathcal{U}[\![\pi]\!]\psi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\varphi\ \mathsf{U}\ \psi, \Delta}\ \mathsf{R19} \qquad \frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\bullet\Box\varphi, \Delta \qquad \Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\Box\varphi, \Delta}\ \mathsf{R20}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\circ\Diamond\varphi,\ \mathcal{U}[\![\pi]\!]\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\Diamond\varphi, \Delta}\ \mathsf{R21} \qquad \frac{}{\Gamma \Longrightarrow \mathcal{U}[\![]\!]\bullet\varphi, \Delta}\ \mathsf{R22} \qquad \frac{\Gamma \Longrightarrow \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![]\!]\circ\varphi, \Delta}\ \mathsf{R23}$$

Table 4.5: Rules for handling temporal operators

4.4.4 Program Rules

The program rules are shown in Tab. 4.6 on the next page. Assignments to local and global variables are handled by the rules R24 and R25, respectively. The only rules that work on both the program and the temporal logic part of a formula are the rules for global assignments. As a reminder, traces have been defined in Def. 3.7 such that only assignments to the global state induce a transition. As a result, rule R25 is only applicable where the program modality contains an assignment and the trace formula begins with a 'next' operator (either weak or strong), that is 'consumed' by this rule. The actual state change is preserved in the update in front of the program modality. This rule has been changed compared with the original DTL definition in [Beckert and Bruns, 2013]: it now uses heap structures instead of global variables.

An `if` statement is handled by splitting the formula in two parts, each containing the alternative program and the remaining program code as shown in rule R26. Similarly, loops can be handled by unwinding, as shown in rule R27. In the case in which the loop condition holds, the loop body is symbolically executed and then again the whole loop. In the second case where the loop condition does not hold, the loop is simply skipped. However, the number of loop iterations may not be known in advance, or the loop may not even terminate. In those cases, we need invariants.

Invariant rules are an established technique for handling loops in calculi for program logics. The basic idea is to have a state formula γ (the invariant) that holds in all states before and—if it terminates—after an execution of the loop body. If we can show that preservation, it only remains to show that φ holds on the remaining trace. The rules are shown in Table 4.7 on the facing page.

For a trace formula of the shape $\Box\varphi$, the four premisses of R28 intuitively state that (i) γ holds in the beginning; (ii) it is preserved by each loop iteration (i.e., it actually is an invariant), here a possible post-π state is characterized by the temporal formula $\bullet\mathit{false}$; (iii) if the loop terminates, indicated by the negated loop condition b, then $\Box\varphi$ holds on the remaining trace; and (iv) for every loop iteration, φ holds throughout, i.e., for the remaining trace from every state during loop iterations. As an invariant abstracts from concrete loop iterations, the context Γ, Δ must be discarded in all but the first premiss.

Note that—in contrast to invariant rules in state based dynamic logic—it is not sound in premiss (iv), to decompose the program trace and to only regard the subtrace induced by π in isolation, i.e., just proving $[\![\pi]\!]\Box\varphi$ is not sound. This has been pointed out by Wagner [2013]. As an example, consider the formula $[\![\texttt{while (X>0) \{X = X-1;\}}]\!]\Box\bullet\bullet\mathit{false}$, that is not valid, but the formula $[\![\texttt{X = X-1;}]\!]\Box\bullet\bullet\mathit{false}$, containing the loop body, obviously is. This

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{x := a\}[\![\omega]\!]\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{x = a; } \omega]\!]\varphi, \Delta}\ \text{R24} \qquad \frac{\Gamma \Longrightarrow \mathcal{U}\mathcal{V}[\![\omega]\!]\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{X = a; } \omega]\!]^{\bullet}_{\circ}\varphi, \Delta}\ \text{R25}$$

where $\mathcal{V} = \{\texttt{heap} := \mathit{store}(\texttt{heap}, X, a) \mid\mid \texttt{heap'} := \texttt{heap} \mid\mid \texttt{estp} := \mathit{false}\}$

$$\frac{\Gamma,\ \mathcal{U}b \Longrightarrow \mathcal{U}[\![\pi_1\ \omega]\!]\varphi, \Delta \qquad \Gamma,\ \mathcal{U}\neg b \Longrightarrow \mathcal{U}[\![\pi_2\ \omega]\!]\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{if (b) } \pi_1 \texttt{ else } \pi_2\ \omega]\!]\varphi, \Delta}\ \text{R26}$$

$$\frac{\Gamma,\ \mathcal{U}b \Longrightarrow \mathcal{U}[\![\pi \texttt{ while (b) } \pi\ \omega]\!]\varphi, \Delta \qquad \Gamma,\ \mathcal{U}\neg b \Longrightarrow \mathcal{U}[\![\omega]\!]\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{while (b) } \pi\ \omega]\!]\varphi, \Delta}\ \text{R27}$$

Table 4.6: Program rules. The schematic symbol $^{\bullet}_{\circ}$ stands for either $\bullet$ or $\circ$.

$$\frac{\Gamma \Longrightarrow \mathcal{U}\gamma, \Delta \qquad \gamma, b \Longrightarrow [\![\pi]\!]\Box(\bullet \mathit{false} \to \gamma) \qquad \gamma \Longrightarrow b, [\![\omega]\!]\Box\varphi}{\gamma, b \Longrightarrow [\![\pi \mid \texttt{while (b) } \{\pi\}\ \omega]\!]\varphi}\ \text{R28}$$
$$\Gamma \Longrightarrow \mathcal{U}[\![\texttt{while (b) } \{\pi\}\ \omega]\!]\Box\varphi, \Delta$$

$$\frac{\Gamma \Longrightarrow \exists u.(u \geq 0 \wedge \mathcal{U}\mathcal{V}_u\gamma), \Delta \qquad n \geq 0 \Longrightarrow \mathcal{V}_{n+1}(\gamma \to (b \wedge [\![\pi]\!]\Diamond(\bullet \mathit{false} \wedge \mathcal{V}_n\gamma)))}{\Longrightarrow \mathcal{V}_0(\gamma \to [\![\texttt{while (b) } \{\pi\}\ \omega]\!]\Diamond\varphi)}\ \text{R29}$$
$$\Gamma \Longrightarrow \mathcal{U}[\![\texttt{while (b) } \{\pi\}\ \omega]\!]\Diamond\varphi, \Delta$$

$$\frac{\begin{array}{c}\Gamma \Longrightarrow \exists u.(u \geq 0 \wedge \mathcal{U}\mathcal{V}_u\gamma), \Delta \qquad n \geq 0 \Longrightarrow \mathcal{V}_{n+1}(\gamma \to (b \wedge [\![\pi]\!]\Diamond(\bullet \mathit{false} \wedge \mathcal{V}_n\gamma))) \\ \Longrightarrow \mathcal{V}_0(\gamma \to [\![\texttt{while (b) } \{\pi\}\ \omega]\!]\varphi_1 \mathsf{U}\, \varphi_2) \qquad n > 0 \Longrightarrow \mathcal{V}_n(\gamma \to [\![\pi \mid \texttt{while (b) } \{\pi\}\ \omega]\!]\varphi_1)\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{while (b) } \{\pi\}\ \omega]\!]\varphi_1 \mathsf{U}\, \varphi_2, \Delta}\ \text{R30}$$

Table 4.7: Invariant rules

means for a sound rule, that we have to consider the remaining trace as well. However, we are only interested in those traces that begin in the subtrace induced by the loop body π.

For this reason, we introduced another, two-place program modality: $[\![\pi \mid \omega]\!]\varphi$ means that for any state in the subtrace induced by π, trace formula φ holds for the remaining trace including ω. More formally, we define $[\![\pi \mid \omega]\!]\varphi$ as a short-hand for $[\![\texttt{x = 0; } \pi \texttt{ x = 1;} \omega]\!](\varphi \,\mathsf{W}\, x \doteq 1)$ where local program variable x does not occur in π, ω, or φ.[13] Even though the resulting formula is syntactically longer here, it is easier to prove in the sense that there are fewer states in which φ has to hold.

Lemma 4.16 (Soundness of invariant rule for $\Box$)**.** *Rule* R28 *is sound.*

Proof. Assume the premisses to be valid. What is to be shown is that the sequent $\Longrightarrow \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\} } \omega]\!]\Box\varphi$ is valid, i.e., it holds in any state. Let us fix some state s. Let $s^{\mathcal{U}}$ be the state that differs from s through the effects of update $\mathcal{U}$.

1. Assume that the loop executed in state $s^{\mathcal{U}}$ does not terminate. This means the trace of the complete program is equal to an infinite concatenation of the traces yielded by the loop body π. Let the states in which the loop condition is evaluated be denoted by s_i for $i \in \mathbb{N}$, i.e., $s_0 = s^{\mathcal{U}}$ and s_{i+1} is the last state in $trc(s_i, \pi)$ (if such exists). It remains to show that for every state s_i, φ holds on the remaining trace beginning in s_i. This follows from premiss (iv) for every state in which ψ and b hold. Obviously, $s_i \vDash b$ (otherwise the loop would terminate). From the validity of (i) follows that $s_0 \vDash \gamma$ and from (ii) follows that if $s_i \vDash \gamma$ then $s_{i+1} \vDash \gamma$ since the formula $\bullet\mathit{false}$ is true exactly in the final state of a trace. By induction over i, this closes the case where the loop does not terminate.

2. Let us now assume that the loop takes exactly $n \in \mathbb{N}$ iterations. Let $s_0, \ldots, s_n$ be as above. The proof follows an induction over n.

IH If the loop executed in a state s_i with $s_i \vDash \gamma$ takes at most n iterations, then $s_i \vDash \zeta$, where $\zeta := [\![\texttt{while (b) \{}\pi\texttt{\} } \omega]\!]\Box\varphi$.

IA $n = 0$, which means that $s_i \vDash \neg b$ because otherwise there would be another loop iteration. The trace of the complete program therefore is equal to the trace of ω when started in s_i and it remains to show $s_i \vDash [\![\omega]\!]\Box\varphi$, which follows from premiss (iii).

[13] An alternative definition of its semantics would be $\tau \vDash [\![\pi \mid \omega]\!]\varphi$ iff for all $i \in [0, |\rho|)$: $\sigma[i, \infty) \vDash \varphi$ where $\sigma := trc(\tau[0], \pi\,\omega)$ and $\rho := trc(\tau[0], \pi)$.

IS $n > 0$: Assume $s_i \vDash b$ (otherwise the proof would conclude trivally). As we have shown above, for the successor state s_{i+1} of s_i, $s_{i+1} \vDash \gamma$ holds and from the induction hypothesis we get $s_{i+1} \vDash \zeta$. By the definition of successors it follows $s_i \vDash [\![\pi \texttt{ while (b) \{}\pi\texttt{\}} \ \omega]\!]\Box\varphi$. Since the loop condition holds in s_i and we know $s_i \vDash [\![\pi \mid \texttt{while (b) \{}\pi\texttt{\}}\,\omega]\!]\varphi$ from premiss (iv), this is equivalent to $s_i \vDash \zeta$.

From premiss (i) follows that the induction hypothesis holds for the initial state $s^{\mathcal{U}}$ in particular. ◁

In the case of R29 ('diamond') and R30 ('until'), the invariant is accompanied by an update $\mathcal{V}_u$ with an integer expression u, that describes the progress made through each loop iteration. The general shape of $\mathcal{V}_u$ is $\{x_1 := f_1(u)\} \cdots \{x_k := f_k(u)\}$ where $x_1, \ldots, x_k$ are variables appearing in γ and $f_1, \ldots, f_k$ are functions. The intuition behind it is that $\mathcal{V}_0\gamma$ describes either a state in which the loop terminates immediately or a fixpoint of the loop. Such a state must be reached in a finite number of iterations, which is guaranteed since n is decreasing in every iteration. For this reason, premiss (ii) requires executions of the loop body to terminate. In Rule R30, there is a fourth premiss stating that φ_1 holds throughout the loop body for every iteration where $n > 0$.

These invariant rules are only sound for purely sequential programs. In a concurrent setting it may be that the invariant does not hold at the end of the loop body because of concurrent modifications. In Sect. 4.6.3, we will introduce slightly modified invariant rules, involving further instrumentation in the loop body code, that are sound for concurrent programs.

4.4.5 Rules for Data Structures

Our calculus is basically independent of the domain of computation resp. data structures that are used. We therefore abstract from the problem of handling the data structures and just assume that an oracle is available that can decide the validity of nontemporal formulas in the domain of computation Note that the oracle only decides propositions about integers, sequences, heaps, etc., that are pure first order formulas. The oracle is represented by rule R31 in Tab. 4.8 on the next page. Rule R32 is an alternative formalization of the oracle that is often more useful.

Of course, the nontemporal formulae that are valid in arithmetic are not even enumerable (cf. Gödel [1931]). Therefore, in practice, the oracle can only be approximated, and rules R31 and R32 must be replaced by a rule (or set of rules) for computing resp. enumerating a *subset* of all valid nontemporal formulas (in particular, these rules must include equality handling). This is not harmful to 'practical completeness.' Rule sets for arithmetic are available, that—as experience shows—allow to derive all valid nontemporal formulae that occur during the verification of actual programs.

$$\text{if } \bigwedge \Gamma \to \bigvee \Delta \text{ is a valid nontemporal formula: } \frac{}{\Gamma \Longrightarrow \Delta} \ \mathsf{R31}$$

$$\text{if } \bigwedge \Gamma_1 \to \bigwedge \Gamma_1' \text{ is a valid nontemporal formula: } \frac{\Gamma_1', \Gamma_2 \Longrightarrow \Delta}{\Gamma_1, \Gamma_2 \Longrightarrow \Delta} \ \mathsf{R32}$$

$$\frac{\Gamma \Longrightarrow \varphi(0), \Delta \qquad \Gamma, \varphi(u) \Longrightarrow \varphi(u+1), \Delta}{\Gamma \Longrightarrow \forall u.\varphi(u), \Delta} \ \mathsf{R33}$$

Table 4.8: Oracle rules and induction rule for handling arithmetic (n is fresh)

Using powerful satisfiability modulo theories (SMT) solvers or dedicated computer algebra systems, this can be done fully automatically in many cases. A rule set for types are provided by Schmitt and Ulbrich [2014], a rule set for heaps and related theories by Weiß [2011], and a rule set for finite sequences by Bubel and Schmitt [2016].

Typically, an approximation of the computation domain oracle contains a rule for structural induction. In the case of arithmetic, that is rule R33. This rule, however, not only applies to nontemporal formulae but also to DTL formulae containing programs.

4.4.6 Other Rules

The remaining rules, that are shown in Tab. 4.9, are the cut rule R35 (with an arbitrary cut formula φ) and the closure rule R34 that closes a proof branch.

$$\frac{}{\Gamma, \varphi \Longrightarrow \varphi, \Delta} \ \mathsf{R34} \qquad \frac{\Gamma, \varphi \Longrightarrow \Delta \qquad \Gamma \Longrightarrow \varphi, \Delta}{\Gamma \Longrightarrow \Delta} \ \mathsf{R35}$$

Table 4.9: The closure and the cut rule

4.5 Soundness and Completeness

We now show that our calculus is sound and complete. *Soundness* is the property that only valid formulae can be derived. The dual property *completeness* means that every valid formula is derivable. Soundness of the calculus $\mathcal{C}_{\mathrm{DTL}}$ (Corollary 4.18) is based on the following theorem, which states that all rules preserve validity of the derived sequents.

Theorem 4.17. *For all rule schemata of the calculus* $\mathcal{C}_{\mathrm{DTL}}$, R1 *to* R35, *the following holds: If all premisses of a rule schema instance are valid sequents, then its conclusion is a valid sequent.*

Proving Thm. 4.17 is not difficult. The proof is, however, quite large as soundness has to be shown separately for each rule. For most rules, the proofs are given in a technical report [Beckert and Bruns, 2012b, Appendix A], that contains a preliminary version of this chapter. However, the invariant rules for $\Box$ and U (Rules R28 and R30) as presented by Beckert and Bruns [2012b] are erroneous. The revised rules presented in this thesis first appeared in [Beckert and Bruns, 2013]. A revised soundness statement for Rule R28 is given in Lemma 4.16 on page 74. Soundness of Rule R30 can be proven in the same spirit.

Corollary 4.18. *If a sequent $\Gamma \Longrightarrow \Delta$ is derivable with the calculus $\mathcal{C}_{\mathrm{DTL}}$, then it is valid, i.e., $\bigwedge \Gamma \to \bigvee \Delta$ is a valid formula.*

Completeness

The calculus $\mathcal{C}_{\mathrm{DTL}}$ is *relatively* complete [Cook, 1978; Wand, 1978]; that is, it is complete up to the handling of the domain of computation (the data structures). It is complete if an oracle rule for the domain is available—in our case one of the oracle rules R31 and R32. If the domain is extended conservatively with other data types, $\mathcal{C}_{\mathrm{DTL}}$ remains relatively complete; and it is still complete if rules for handling the extended domain of computation are added.

Theorem 4.19. *If a sequent is valid, then it is derivable with $\mathcal{C}_{\mathrm{DTL}}$.*

Corollary 4.20. *If φ is a valid DTL formula, then the sequent $\Longrightarrow \varphi$ is derivable.*

The proof of Thm. 4.19 will not be presented here in detail, as it is quite voluminous. The basic idea of this proof is the same as that used by Harel [1979] to prove relative completeness of his sequent calculus for first order dynamic logic. An extensive proof sketch can be found in the technical report [Beckert and Bruns, 2012b, Appendix B]. The following lemma is central to the completeness proof.

Lemma 4.21. *For every DTL formula φ_{DTL} there is an (arithmetical) nontemporal first-order formula φ_{FOL} that is logically equivalent to φ_{DTL}, i.e., for all traces τ and variable assignments β:*

$$\tau, \beta \vDash \varphi_{DTL} \quad \textit{iff} \quad \tau, \beta \vDash \varphi_{FOL} \ .$$

The above lemma states that DTL is not more expressive than FOL plus arithmetic. This holds as arithmetic—our domain of computation—is expressive enough to encode the behavior of programs. In particular, using gödelization, arithmetic allows to encode program states (i.e., the values of all the variables occurring in a program) and finite (sub-)traces into a single

number. Our TR, Sect. B.1 describes such an encoding, following the ideas by Harel [1979] and Platzer [2004]. Furthermore, these results imply the existence of a strongest invariant for any loop (albeit not in a constructive way).

It is then possible to construct, for every DTL formula ψ, state s, program π, and $n \in \mathbb{N}$, a FOL formula $\varphi_{\psi,s,\pi,n}$ encoding that $trc(s,\pi)[n,\infty) \vDash \psi$. This is shown in [TR, Sect. B.2], effectively proving Lemma 4.21. Temporal operators are thereby substituted by quantification over the integers. Note that Lemma 4.21 states a property of the logic DTL that is independent of any calculus.

Although Lemma 4.21 shows that to every DTL formula there is a logically equivalent FOL formula, we still need to prove that our calculus can actually derive them. The proof essentially amounts to showing that, for every valid formula, there exists a possible rule application that 'brings the proof forward.' This is not obvious since our calculus does not have the *subformula* property, i.e., every formula in a premiss is a subformula of a formula in the conclusion. To formalize proof progress, we first introduce a metric on formulae [TR, Sect. B.3]. We prove that, for every rule, the values for the premisses are strictly smaller than for the conclusion. Finally, we prove that for every DTL formula φ, there is exists a calculus rule with φ in the conclusion, that has the above progression property. It follows Thm. 4.19.

Practical Completeness

Lemma 4.21 implies that a DTL formula could be decided by constructing an equivalent nontemporal formula and then invoking the computation domain oracle—if such an oracle were actually available. But even with a good approximation of an arithmetic oracle, this is not practical (the nontemporal first-order formula would be too complex to prove automatically or interactively). And, indeed, the calculus $\mathcal{C}_{\mathrm{DTL}}$ does not work that way.

The (relative) completeness of $\mathcal{C}_{\mathrm{DTL}}$ requires an expressive computation domain and is lost if a simpler domain and less expressive data structures are used. The reason is that in a simpler domain it may not be possible to express the required invariants for all possible while loops.

4.6 Discussion

In this chapter, we have defined Dynamic Trace Logic (DTL) and its extension to concurrent dWRF programs, CDTL, that stem from a novel combination of dynamic logic and first order temporal logic. In contrast to earlier work by Beckert and Schlager [2001] and Platzer [2007], there is no restriction on the shape of trace formulae. Through this, we have got an expressive logic allowing to describe complex temporal properties of programs. Symbolic execution rules for the `fork` and `release` statements will be presented in the

following chapter. A discussion on how dWRF models actual concurrency can be found in Sect. 3.6.

4.6.1 Example

We present a proof of a nontrivial DTL formula below; other (smaller) proofs can be found in the thesis by Wagner [2013]. Consider the (nonterminating) noninterleaved program

$$\texttt{while (true) \{}\pi\texttt{\}} \quad \text{with} \quad \pi := \texttt{a = X; X = a - a/2;}$$

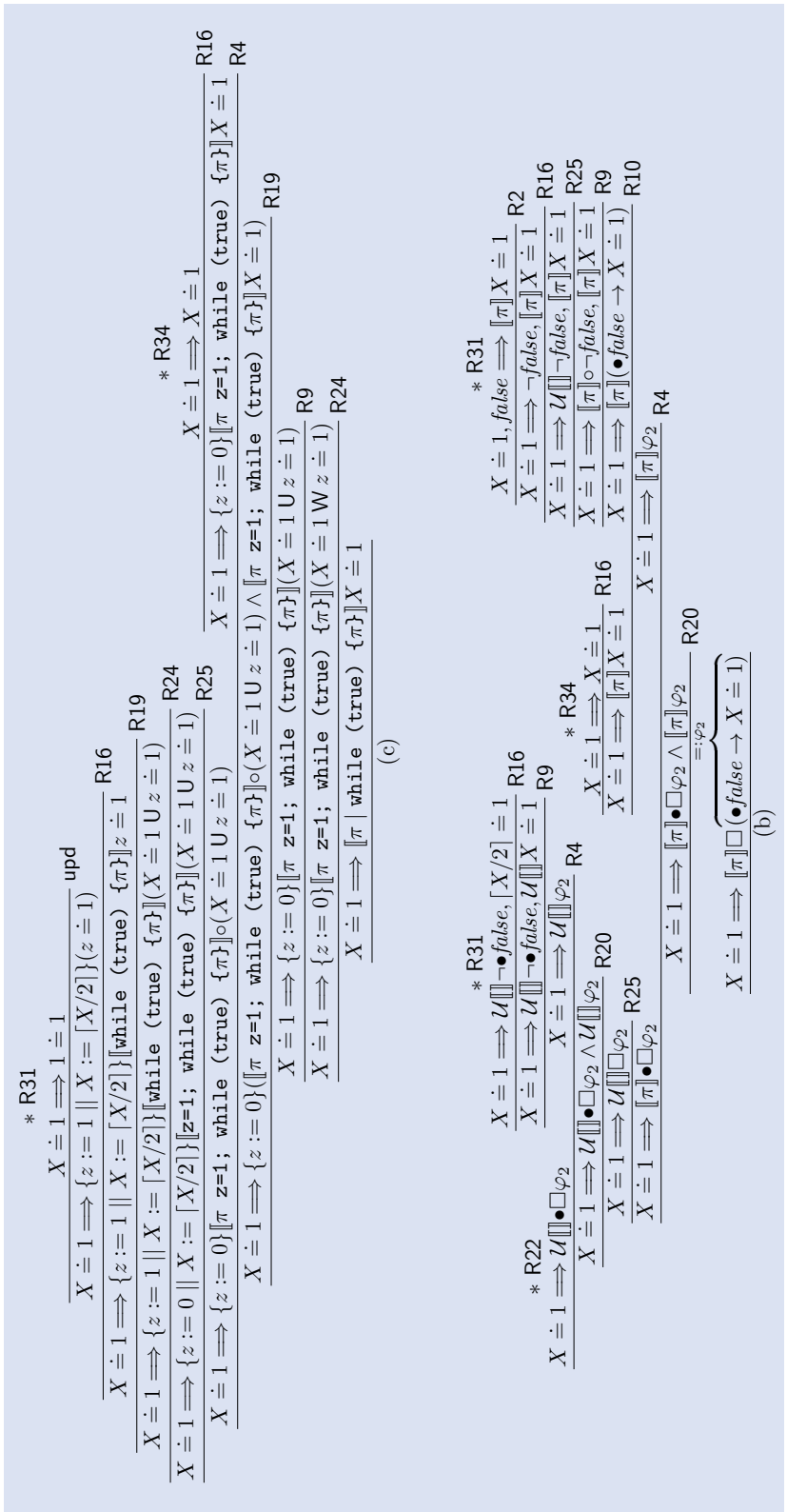
(where `a` is a local variable and `X` is a global variable). Remember that the slash symbol is interpreted as integer division without remainder,[14] i.e., `X` is assigned $X - \lfloor X/2 \rfloor = \lceil X/2 \rceil$ on each iteration, where $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ denote the floor and ceiling function, respectively. This program obviously does not terminate for any value of `X`. However, for positive values of X, any execution trace eventually stabilizes in a state where $X \doteq 1$ and maintains this property. This property cannot be expressed using standard dynamic logics. In our logic DTL, it can be expressed as $[\![\ldots]\!]\Diamond\Box(X \doteq 1)$.

Figure 4.10 on the following double page (pages 80f.) shows a complete proof tree (in four parts). The proof starts in the lower-most part (recto page). In order to keep it readable, we hide all sequent formulae that are not involved in the further proof. We also omit explicit heap objects and simply write 'X' for a *select* operations on location X.

The overall structure of the proof is as follows: first, we apply the invariant rule for $\Diamond$. In the use case branch branch (shown in the upper half of the recto page), the $\Diamond$ operator can be unwound, and we apply the invariant rule for $\Box$. The 'invariant preservation' and use case branches are shown in the lower and upper part of the verso page, respectively. All of these branches require one or more unwinding of temporal operators.

In detail, the first rule application is the invariant rule for diamond, R29, where the invariant is $y \doteq X - 1$ and the accompanying update is just $\{y := u\}$. The branch on the left (i.e., there exists an integer u such that the invariant is initially valid) can be closed within a few steps. In the center branch, we have to prove the 'step case:' If y is strictly greater than zero, then it is greater or equal to zero after one loop iteration. Temporal properties (in the more narrow sense) do not appear here, but the 'postcondition pattern' $\Diamond(\bullet \mathit{false} \wedge \varphi)$ where the formula φ is called postcondition. We abbreviate the antecedent by Γ and the postcondition by φ_1. The $\Diamond$ operator is unwound for a first time (rule R21), where we are only interested in the 'future' part, which allows us to produce the empty program (rule R25). Then, the remaining $\Diamond$ operator is unwound for a second time. This time, the 'present' formula

[14] Euclidian and Java integer division coïncide on nonnegative operands, therefore we do not distinguish them here.



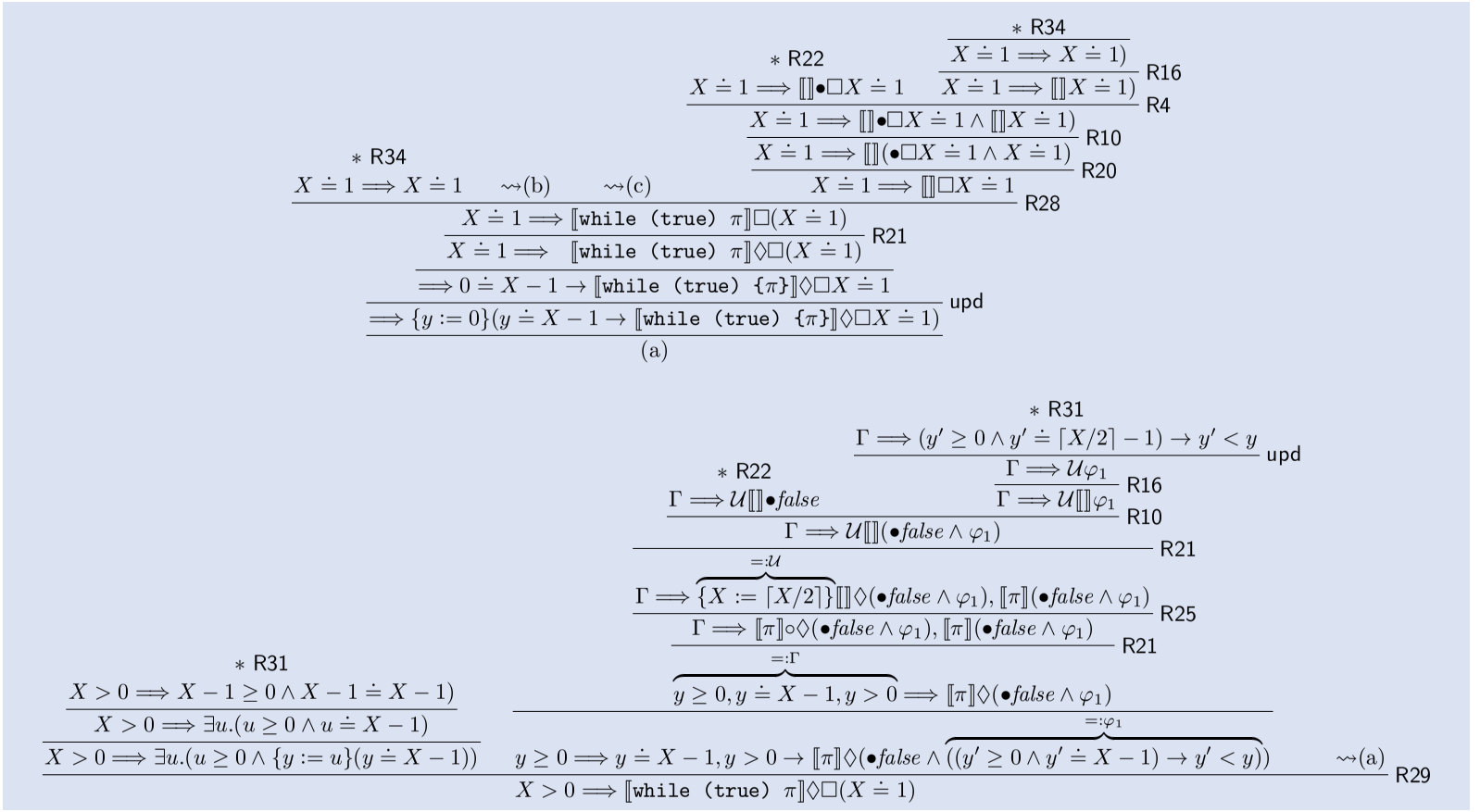


Figure 4.10: Example proof for the trace formula ⟦while (true) π⟧◇□(X ≐ 1)

$\bullet \mathit{false} \wedge \varphi_1$ is of interest to us. Splitting the conjunction, both branches can be closed with either rule R22 or arithmetic.

The third branch (use case) of the initial invariant rule application is shown above. In the invariant y is now exactly zero, meaning that $X \doteq 1$ holds. It remains to prove that this property is preserved throughout the remaining trace. We apply rule R21 and hide the 'future' part, leaving only the $\Box$ operator. Now we can apply the invariant rule for $\Box$ with the invariant $X \doteq 1$. Please note that this is not an invariant for the loop in general, but only for states in which this property already holds. The first premiss closes immediately. In the fourth branch (use case), we unwind the $\Box$ operator once (rule R20); both resulting branches can be closed easily. The remaining premisses, the formula $X \doteq 1$ is indeed an invariant and $X \doteq 1$ holds throughout the subtrace induced by the loop body, are shown in the lower and upper part of the verso page, respectively.

In the former branch, we have the 'invariant pattern' $\Box(\bullet \mathit{false} \rightarrow X \doteq 1)$. Unwinding the $\Box$ operator through rule R20 yields two branches, that can be closed without much effort. In the 'future' branch, we have to apply the unwind rule once more to produce a sequent $X \doteq 1 \Longrightarrow \lceil X/2 \rceil \doteq 1$, which can be closed by arithmetic.

The latter branch, shown above, we have the auxiliary modality $[\![\cdot \mid \cdot]\!]$, that represents the property of $X \doteq 1$ being preserved throughout the subtrace induced by the loop body π. As defined on page 74, it is just a shorthand for a modality with a 'weak until' temporal formula, where $\varphi \,\mathsf{W}\, \psi$ is again a shorthand for $\varphi \,\mathsf{U}\, \psi \vee \Box\varphi$; this is why 'weak until' is replaced by 'until' through rule R9 (and we hide the 'box' formula). Then, we unwind the 'until' operator (rule R19), yielding two branches, one of which can be closed easily. In the other branch, we can apply the assignment rule R25, such that only the loop remains in the program modality. The 'until' operator in the temporal formula is unwound onece more; we hide the 'future' part. It only remains a nontemporal formula, which can be simplified to $1 \doteq 1$ through update simplification. Here, the update can be applied to the nonmodal formula as a substitution. This closes the proof.

4.6.2 Trace Decomposition Rules

State based dynamic logics, both for deterministic and indeterministic languages, have the well-known property of compositionality. For example, the formulae $[\pi\ \omega]\,\varphi$ and $[\pi]\,[\omega]\,\varphi$ are logically equivalent. This is important since program complexity imports much to the overall complexity of a DL formula. This does not apply to our situation as traces may not be decomposed in general.

For purposes like loop invariants (see Tab. 4.7), however, program decompositions are indispensable. This has lead us to the auxiliary notation $[\![\pi \mid \omega]\!]\varphi$, that talks about all traces beginning in π but extending into ω.

In some sense, this modality has similar semantics to the 'chop' operator **;** in ITL [Cau et al., 2002], that divides a temporal interval. The difference is that the chop operator denotes *some* point in time to split, whereas our modality explicitly states the final state of $trc_\Sigma(\cdot, \pi)$.

Another possibility to make proofs more tractable would be to introduce additional rules for special, commonly used patterns of trace formulae—such as $\Box\Diamond\gamma$ where γ is a state formula—for which we know that decompositions are sound. For instance, the related dTL^2 logic by Jeannin and Platzer [2014] (cf. Sect. 10.1) does only apply rules on temporal formulae of such patterns.

The following rule is an invariant rule for the special case that the temporal formula is of the shape $\Box\varphi$, where φ is a nontemporal formula.

$$\frac{\Gamma \Longrightarrow \mathcal{U}\gamma, \Delta \qquad \gamma, b \Longrightarrow [\![\pi]\!]\Box(\varphi \wedge (\bullet \mathit{false} \rightarrow \gamma)) \qquad \gamma \Longrightarrow b, [\![\omega]\!]\Box\varphi}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Box\varphi, \Delta} \quad \text{R36}$$

It lacks the fourth branch involving the auxiliary modality; instead the 'invariant preservation' branch contains the throughout property. With this rule, the above example proof would be shorter, essentially lacking the left branch in the uppermost subtree.

4.6.3 Loop Invariants for Concurrent Programs

Like the control flow, invariants can depend on shared locations (cf. Sect. 3.6). This means that the original invariant rules shown in Tab. 4.7—while being sound w.r.t. the given semantics—are not appropriate for modeling concurrent programs. The reason is that they introduce a kind of program decomposition that is affected by interleavings. A concurrently executed thread may influence whether the invariant γ holds. For this reason, we slightly adapt the invariant rules as shown in Tab. 4.11 on the next page. They only differ from the original ones by additional instrumentation. In all rules, in the first premiss the invariant γ is replaced by $[\![\_\_]\!]\gamma$ (i.e., a program modality with the instrumentation of the empty program). This adds exactly one interleaving point at the end of the empty program. In the second and the last premiss of all rules, the loop body π is instrumented, which effectively adds an interleaving point at the end of π.

Further instrumentations (i.e., on the trailing program ω or the complete loop) are not necessary since the programs are either already properly instrumented or the same invariant rules apply. The principle extends to all rules that perform case distinctions during symbolic execution. Yet, loop invariants are the only rules of this kind in our calculus. The JML specification language (see Chap. 8) knows more auxiliary specification elements that are evaluated within a program. In extending the calculus to cater for those, one has to take great care to apply the appropriate instrumentations.

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\![\_]\!]\gamma, \Delta \qquad \gamma, b \Longrightarrow [\![\underline{\pi}]\!]\Box(\bullet \mathit{false} \to \gamma) \qquad \gamma \Longrightarrow b, [\![\omega]\!]\Box\varphi \quad \gamma, b \Longrightarrow [\![\underline{\pi} \mid \texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\varphi}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Box\varphi, \Delta} \text{ R28}'$$

$$\frac{\Gamma \Longrightarrow \exists u.(u \geq 0 \wedge \mathcal{U}\mathcal{V}_u[\![\_]\!]\gamma), \Delta \qquad n \geq 0 \Longrightarrow \mathcal{V}_{n+1}(\gamma \to (b \wedge [\![\underline{\pi}]\!]\Diamond(\bullet \mathit{false} \wedge \mathcal{V}_n\gamma))) \Longrightarrow \mathcal{V}_0(\gamma \to [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Diamond\varphi)}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Diamond\varphi, \Delta} \text{ R29}'$$

$$\frac{\Gamma \Longrightarrow \exists u.(u \geq 0 \wedge \mathcal{U}\mathcal{V}_u[\![\_]\!]\gamma), \Delta \qquad n \geq 0 \Longrightarrow \mathcal{V}_{n+1}(\gamma \to (b \wedge [\![\underline{\pi}]\!]\Diamond(\bullet \mathit{false} \wedge \mathcal{V}_n\gamma))) \quad \Longrightarrow \mathcal{V}_0(\gamma \to [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\varphi_1 \,\mathsf{U}\, \varphi_2) \qquad n > 0 \Longrightarrow \mathcal{V}_n(\gamma \to [\![\underline{\pi} \mid \texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\varphi_1)}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\varphi_1 \,\mathsf{U}\, \varphi_2, \Delta} \text{ R30}'$$

Table 4.11: Modified invariant rules with instrumentation

4.6.4 Implementation

The sequent calculus $\mathcal{C}_{\mathrm{DTL}}$ for the sequential subset of the language has been prototypically implemented on top of the legacy version 2.2 of the interactive KeY prover.[15] Instead of the simple language introduced in this paper, the implemented calculus works on actual Java programs. The implementation benefits from the fact that most complex statement in Java can be transformed into a sequence of *simple* statements. This is a key element of the symbolic execution in the JavaDL calculus of the KeY system [Beckert, 2001; Beckert, Klebanov, and Schlager, 2007b]; see also Chap. 7. Most calculus rules dealing with this kind of program normalization can be adapted straight away from the present rules for the $[\cdot]$ modality in the JavaDL calculus.

4.6.5 Proof Search for DTL

Beyond completeness, another desirable feature of a calculus would be that many proofs can be found automatically. Of course, the general problem is undecidable and therefore there cannot be a decision procedure for all problems. Yet, work with the KeY system has revealed that for many 'sensible' examples, heuristical proof search provides automated proofs.

Proof confluent calculi are amenable to automation. Confluence means the property that if a proof node is reachable (through finitely many applications of valid rules), then it is still reachable after a valid rule application. In particular, for a valid formula φ, the empty sequent must be reachable on any branch after any sequence of rule applications. This avoids the necessity of backtracking. Unfortunately, neither the JavaDL calculus of Beckert et al. [2007b] nor $\mathcal{C}_{\mathrm{DTL}}$ is proof confluent. The lack of confluence in $\mathcal{C}_{\mathrm{DTL}}$ arises from the intricate interplay of temporal and program rule applications.

Nevertheless, adequate heuristics can still provide good support for automation. In standard dynamic logic calculi (for deterministic languages), program transformation rules usually have a high priority since most of them do not split the proof while there are few rules that rewrite subformulas below modalities. This is different for our calculus. The temporal unwinding rules R20 and R19 lead to situations where we have multiple program modalities on the trace. This causes confusion to the strategies and leads to significantly larger proofs, in particular if the programs are complex. Therefore, it becomes an issue of proof complexity whether first to symbolically execute the program or to rewrite the formula below the program modality. An efficient strategy for DTL needs to decide which formula containing a program modality is 'the interesting one' heuristically.

[15] This version is no longer maintained and only available on request.

4.6.6 Outlook on the Following Chapters

So far, we only presented the calculus for basic DTL, i.e., containing only noninterleaved programs. Since concurrent interleavings are made explicit in dWRF, the calculus is largely identical to the calculus for sequential programs presented in [Beckert and Bruns, 2013]. The main difference is that validity is defined w.r.t. a scheduler. In the following chapter, we develop the missing calculus rules for interleavings and thread creation based on the rely/guarantee technique. The correspondence of actual concurrent programs to sequential interleaved programs is discussed in that chapter.

In Sect. 6.4, we use an extension of CDTL to specify *strong noninterference*. Noninterference is a relational property that compares two traces produced by the same program. Unfortunately, this property cannot be expressed using the LTL-like temporal operators of CDTL. To solve this issue, we will introduce an extension to CDTL that allows to count states in the trace through the operator $\bullet^n$, where $\bullet^n\varphi$ intuitively means 'in exactly n steps, φ holds.'

Another extension to express (absence of) information flow has been presented by the author [Bruns, 2014b]. It employs explicit temporal information flow operators H ('hide until') and L ('leak while'), based on the work by Dimitrova et al. [2012], to express strong noninterference, including temporal declassification.

In Sect. 7.3, we discuss how reasoning about concurrent programs along the lines of Chap. 5 can be implemented in the KeY verification system. One challenge is to provide an implementation that is minimally invasive to the core system of KeY. The implementation presented on the preceding page is not suitable as it involves many core changes. The other challenge is to devise a calculus for the full Java language. As already discussed in this chapter, extending the DTL calculus to Java does not pose any conceptual obstacles, but is rather laborious.

5 Deductive Verification of Concurrent Programs

In the previous chapter, we have introduced a calculus for DTL, based on symbolic execution of programs in the sequential fragment of dWRF. In this chapter, we extend this symbolic execution to the full multi-threaded language, including interleavings and thread creation, as introduced in Chap. 3. The result is a calculus for the full CDTL. Our goal is to allow modular reasoning in *open programs*, that can be extended with further threads. The distant goal is to enable information flow analysis for multi-threaded programs, as explained in the following chapter. We do not aim for a full-fledged methodology to verify parallel algorithms. In Chap. 7, we describe an implementation of this approach in the KeY verification system.

The rely/guarantee approach allows to reason compositionally about the behavior of shared memory concurrent programs. We investigate on a single thread (executing a sequential interleaved program—as defined in Sect. 3.2) in isolation, with possible spontaneous state transitions induced by the environment. This means that we still have a deterministic program semantics, with underspecified (i.e., havocked) heap states. Rely/guarantee uses functional specification to restrict the effect of these environment transitions. In this sense, reasoning about interleavings is similar to reasoning about sequential method invocations through contracts. We present a calculus for CDTL based on rely/guarantee, that is amenable to automation. It contains a rule for `release` statements that havocs the heap state, while it adds a rely condition to the assumptions. Proof obligations that ensure soundness of this rule are generated. We only consider these proof obligations here (i.e., proof of well-behaved concurrency according to specification) in order to keep proof obligations modular and reusable.

5.1 Concurrent Verification

Besides the obvious *soundness*, it is attainable for analysis techniques to be compositional and modular. *Compositionality* means that single components can be composed together in a way such that any result on the compound can be derived from results on the components. This derivation is provided by an 'adaptation rule' [de Roever et al., 2001, Sect. 1.6]. Compositionality is essential to a bottom-up approach to verification of a complex system. *Modularity* means that modules can be assessed in isolation, i.e., without concrete knowledge of the environment. This way, modules can be deployed in different environments without invalidating previously obtained results. Modularity is essential to a top-down approach to a complex system. Both compositionality and modularity are obviously related, but neither does subsume the other. There are approaches that are compositional, but not modular, and vice versa. The overview monograph by de Roever et al. [2001] discusses the concept of compositionality extensively, while modularity is mentioned only marginally. Yet, both are linked by the requirement for assertional reasoning.

Several approaches to formally reason about shared memory concurrent programs have been developed since the mid-1970s, such as [Ashcroft and Manna, 1971; Ashcroft, 1975; Keller, 1976; Hoare, 1978; Lamport, 1980]. Widely known is the one by Owicki and Gries [1976], that is considered the first practical approach to concurrency verification.[1] They define an extension to Hoare logic for programs with parallel composition. The major issue with this approach is that the rule for parallel composition requires isolated threads. This means that, in addition to proving local correctness of programs, one needs to prove noninterference of parallel executions. This technique has some limitations: 1. the number of concurrent threads is fixed;[2] 2. noninterference proofs tend to be complicated;[3] 3. it is not compositional; and 4. it requires a considerable specification overhead (every program point where an interleaving may occur needs to be annotated). Given all these issues, applying the Owicki and Gries method is not practical.

The rely/guarantee approach [Jones, 1983; Stirling, 1988; Stølen, 1991] (sometimes also called "assume/guarantee") attempts to overcome these issues. Based on an earlier idea by Francez and Pnueli [1978], it abstracts away from concrete interferences to only consider the *effects* of possible interleavings. The main idea is similar to the concepts of *contracts* for sequential modules (see Sect. 8.2), though not on the level of a public interface, but of atomic program steps. This allows to modularly reason about one single thread in isolation, while there may be an unbounded

[1] See also the discussion in Sect. 10.3.

[2] The generalization by Prensa Nieto [2001] is parametric in the number of threads.

[3] The proofs grows exponentially with the number of threads.

number of other threads in an only partially specified environment. It is, in particular, not of any interest which sequential program other threads execute or in which (thread local) state they are. Sequential programs (i.e., single threads) are evaluated over traces of states.

The original paper by Jones sketches this fundamental idea—as a means of developing correct programs by design. But we note that it may be applied to ex post factō verification as well. A comprehensive account on the rely/guarantee approach can be found in the article by Xu, de Roever, and He [1997], that includes a Hoare style calculus and proofs of soundness and completeness for a fixed number of threads. A completeness proof for a system that is parametric in the number of threads can be found in [Prensa Nieto, 2002]. Xu et al. further "observe that the rely-guarantee method is [...] a reformulation of the classical non-compositional Owicki & Gries method." A technique for distributed systems with message passing, that is similar to rely/guarantee, is called *assumption/commitment* [Misra and Chandy, 1981]. We present an overview over related work in Sect. 10.3.1.

Functional specifications, like preconditions or assumptions, help to describe behavior that must be established by a module. Conversely, we also need to ensure that "nothing else changes" in order to provide modular specification. In sequential programs, this *frame problem* [McCarthy and Hayes, 1969; Borgida et al., 1993] is well known; see Sect. 8.2.3.

We present an implementation in dynamic logic with 'contracts' for each heap read access. To reduce the specification overhead, we complement functional rely/guarantee specifications with framing, that restricts havoc to defined partitions of the heap. Our framework of proof obligations does not include postconditions (i.e., functional correctness properties) directly. This decision is rooted in our effort to separate rely/guarantee, i.e., proof of benevolent thread interactions, from other program properties. This permits future extensions of our approach to apply rely/guarantee to properties *beyond* postconditions, such as information flow security.

Chapter Overview

In this chapter, we describe how rely/guarantee can be integrated into our logical framework. Section 5.2 gives some fundamental definitions regarding our rely/guarantee approach and the kind of programs that we consider. We describe the differences to Jones [1983] and other related work shortly there. A more comprehensive treatise of related work can be found in Sects. 10.3ff.

In Sect. 5.3, we develop correctness conditions, that are partly thread-local, partly on the system level. If they hold, then they assure soundness of a calculus rule to deal with environment interleavings. This calculus rule is to be introduced in Sect. 5.4.1. We give a proof of soundness relative to specification for this rule. The overall proof structure develops bottom-up and follows the same lines as Coleman and Jones [2007]. Central to this is

Thm. 5.13, which states that a valid thread specification is sufficient for rely conditions describing interleavings.

A rule for dynamic thread creation is introduced in Sect. 5.4.2. While it is trivially sound, the verification challenge lies in proving that the soundness criteria of the interleaving rule are maintained. We argue that the resulting calculus is relatively complete w.r.t. a modular analysis of the proposed target programing language dWRF.

We provide two minimal examples to show how rely/guarantee proofs work (Sect. 5.5). It is only natural to include rely/guarantee specifications in JML contracts. A proposal, introducing a few new keywords, will be presented in Sect. 8.3 as part of the chapter on JML specification. We conclude this chapter with a discussion on future work in Sect. 5.6. It concerns the addition of synchronization primitives to dWRF. While we do not provide a formal underpinning, we sketch the idea of how synchronized threads can achieve a common task coöperatively.

5.2 Rely/Guarantee Reasoning

The central idea of rely/guarantee is to describe concurrent program behavior symbolically through specification. If specifications are sufficiently abstract, then they can describe *any* concurrent behavior that is possible. We specify both the result of 'own' atomic steps and the combined effect of environment interference. Recall the transition functions from Sect. 3.5.2, σ_t (of the thread t under investigation / our 'own' thread) and $\sigma^*_\Sigma(t)$ (of the environment / 'them'). We use formulas *rely* and *guar* to describe those. Those are *two-state invariants*, i.e., they are preserved throughout the execution and are evaluated over two succeeding states. The formula *rely* describes $\sigma^*_\Sigma(t)$, i.e., it defines on which properties the execution of t may rely upon. The formula *guar* describes σ_t, i.e., it defines which properties the execution of t has to guarantee.

Obviously, there always are strongest formulae satisfying these conditions (if the environment is perfectly known): The strongest rely relation is the reflexive/transitive closure of the union of guarantee relations. However, the transitive closure of guarantees is not expressible in first order logic.

In practice, this strongest condition will not be necessary. It is sufficient that *rely* is strong enough to imply the overall goal (e.g., a postcondition) and that *guar* is strong enough—in disjunction with the *guar* specification of other threads—to imply the *rely* conditions of a third party thread. Since they describe the behavior of zero or more atomic environment transitions, rely conditions have to be always reflexive and transitive. Jones [1983] additionally requires guarantees to be reflexive and transitive. But this restricts the possible specifications and requires an additional proof obligation, while it does not provide any advantages since the union of transitive relations

is not necessarily transitive again. Prensa Nieto [2003] also requires reflexive guarantees. This is motivated by the fact that in her formalization, there may be component steps that do not lead to a different global state, e.g., evaluation of expressions. This effect does not occur in our semantical framework as only write actions to global memory induce a local step. Xu et al. [1997] do not require rely conditions to be transitive. Instead, they require that pre- and postconditions are maintained through environment transitions. While this is more liberal on the shape of rely conditions, it severely restricts pre- and postconditions in practice.

Extensions to 'Classical' Rely/Guarantee

In typical situations, the memory partitions to which different threads write are strongly separated and only a few locations are actually shared. Therefore we combine the well-known two state invariant specification of threads with *framing*, to specify what locations a thread writes to at most (and what locations it can rely on not to be changed). Frame specifications alone can be very expressive, in the dynamic frames approach [Kassios, 2011; Weiß, 2011], location sets describing frames can depend on the program state and can be constructed through comprehensions (see Sect. 8.2.3).

Through framing, we take the burden of specifying the 'nonbehavior' of threads in addition to its behavior. This allows us to formulate the functional guarantees and rely conditions in a more concise and focussed way. Otherwise, a typical specification would consist of many statements of equality of variable values between states. In the corner case in which the memory accessed by threads is perfectly separated, the thread specification can consist of frame conditions only (with the functional specification declared as *true*). Our approach is thus similar to rely/guarantee combined with separation logic [Vafeiadis and Parkinson, 2007]; cf. Sect. 10.3.3. A difference is that our proof obligations are expressed in a dynamic logic that builds on classical FOL.

In modular specification and verification for sequential programs, the concept of method contracts is well known (see Sect. 8.2). We borrow *preconditions* from the contract methodology to restrict the states in which fresh threads can be created. Like framing, this does not increase the expressiveness of the approach, but it is very effective in reducing the specification overhead. Following the approach by Weiß [2011], we do not include implicit class invariants in this framework, but leave it to the specifier to refer to invariants explicitly in specifications.

Dynamic Thread Creation vs. Parallel Composition

Most works on rely/guarantee, including [Jones, 1983; Prensa Nieto, 2002], define concurrent programs as one syntactical entity, involving parallel com-

position operators. This way, programs are *closed*: there is no further environment beyond its syntactical representation. This is motivated by the fact that Jones proposed rely/guarantee as a technique to *develop* programs, where the developer has perfect knowledge of the environment (because it is part of the development process). However, this is in contrast to our concept (cf. Chap. 3) of having *open* programs consisting of a set of threads, where the 'own' thread runs in an underspecified environment. This environment does not manifest syntactically and it can be expanded dynamically (through `fork` statements).

This idea of open programs is essential to modular proofs: a correctness proof about a thread should be valid under any environment. For sequential verification, this concept is well established and enshrined in the methodology of contracts.[4] In particular, it allows to evolve programs in a natural way without invalidating previous verification results.

It is one of our goals in this thesis to establish modular proofs in the sense that they must not be invalidated by the addition of threads. For reasoning about such programs, we observe that we cannot assume a 'top-level entry point' or 'master thread' from which all others are forked. This bars us from the assumption by Coleman and Jones [2007] "that whole programs are run without external interference." In particular, we allow any thread to fork new threads (i.e., we consider nested concurrency), whereas Prensa Nieto [2002] assumes a single master thread, which is the only thread that may fork others.

5.3 Proof Obligations

A *thread specification* is a tuple $(pre_t, rely_t, guar_t, R_t, M_t)$ where pre_t is a state formula, $rely_t$ and $guar_t$ are two-state formulae, and R_t and M_t are terms of type $\mathbb{L}$. The intuitive understanding is that the active thread can rely on the relation $rely_t$ to hold between state transitions induced by the environment, while the locations in R_t never change due to the environment, and at the same time, it guarantees to write only to the locations in M_t and to maintain the relation $guar_t$ between all own atomic steps. The precondition pre_t restricts the states in which fresh threads may be created.

A formal definition of a thread specification being valid is given in Def. 5.11 on page 99. The formulas *rely* and *guar* are still state formulae in the sense that they must not include temporal operators, but they are expected to refer to the built-in heap variables `heap` and `heap'`, for that we justify it as 'two-state.' Just like the semantics of state formulae can be represented

[4] The approach by Beckert et al. [2007a] to verification of object-oriented programs includes a concept of openness, too: there proofs must be valid w.r.t. extensions of the subtype relation.

as the set of states in which it is valid, two-state formulae represent binary relations on states.

The location set expressions can be nontrivial, e.g., depending on the state or including if-then-else operators. This makes location set expressions as expressive as the logic itself.

Thread specifications relate only to one thread, not to a complete concurrent program. This is important in order to have modular specifications for reasoning about open systems. The choice of a set R_t of locations that must *not change*, instead of the set of locations that may change, may seem counterintuitive at first sight. But in a strictly modular setting, we (consequently) cannot name the locations that are allowed to change.

Auxiliary Definitions and Lemmas

We use the following definitions to express unchanged behavior on the semantical level.

Definition 5.1. Let $\mathcal{L}$ be a location set. Two heap mappings h and $h' \in \mathcal{D}_{\mathbb{H}}$ are *$\mathcal{L}$-equivalent*, written as $h \approx_{\mathcal{L}} h'$, if $h(F) = h'(F)$ for all global variables $F \in \mathcal{L}$. Two states s and s' are *$\mathcal{L}$-equivalent*, written as $s \approx_{\mathcal{L}} s'$, if $\mathtt{heap}^s \approx_{\mathcal{L}} \mathtt{heap}^{s'}$. A binary relation $A \in \mathcal{S}^2$ *respects* $\mathcal{L}$ if $A \subseteq \approx_{\mathcal{L}}$.

Note that a relation that respects equivalence is not necessarily reflexive nor transitive. For the following definitions and lemmas, we take the liberty of identifying a state s with its heap state $\mathtt{heap}^s$. The following two lemmas follow immediately from the definition.

Lemma 5.2. *$\approx_{\mathcal{L}}$ is an equivalence relation.*

Lemma 5.3. *Let $\mathcal{L}_1 \subset \mathcal{L}_2$ be location sets. We have $\approx_{\mathcal{L}_1} \supset \approx_{\mathcal{L}_2}$.*

The last lemma states that $\approx$ is strictly antimonotonic in its location set parameter. For a location set *expression* L, we write $\approx_L$ as shorthand for '$\approx_{L^s}$ for all $s \in \mathcal{S}$.' We will use this style of relations included in one another throughout the proofs below.

Lemma 5.4. *Let φ be a formula and $\mathcal{L}$ a location set. We say $\mathcal{L}$ is* φ-invariant *if for all states s with $s \vDash \varphi$ and s' with $s \approx_{\mathcal{L}} s'$, it holds $s' \vDash \varphi$.*

1. *Let φ be a formula and let $\mathcal{L}$ be a φ-invariant location set. Let s be a state with $s \vDash \varphi$ and L a location set expression with $L^s = \mathcal{L}$. Then it holds that $s \vDash \{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, L^{\complement})\}\varphi$.*

2. *Let $\mathcal{L} \subseteq \mathcal{L}'$ be location sets. If $\mathcal{L}$ is φ-invariant for some formula φ, then $\mathcal{L}'$ is φ-invariant.*

Note the two inversions in the above lemma: in 1, we use the complement $L^{\complement}$ of the location set expression L, while in 2, we describe an inverse monotonicity relation. The lemma intuitively states that anonymization is ineffective to the validity of a formula if the formula does not depend on the locations that we anonymize. There is also an upper bound on ineffective anonymization;[5] when we know that anonymization is ineffective for some location set, it is also ineffective for a smaller location set. We will use this lemma to prove Thm. 5.16.

Proof. Ad 1: This is a logic representation of the semantical property described by the definition of *anon* (cf. Sect. 3.3). Ad 2: This follows from the (inverse) $\mathcal{L}$-monotonicity of the $\mathcal{L}$-equivalence relation. ◁

We define the notion of states that are reachable (through some transition function) from a state in which a precondition holds and relations that are constrained to reachable states. We use this definition to develop a thread specification that is not universally valid, but valid in all reachable states.

Definition 5.5 (Reachability from precondition)**.** The set of reachable states w.r.t. a transition function $\sigma : \mathcal{S} \to \mathcal{S}$ and a formula φ, written $\mathit{reach}(\sigma, \varphi)$, is defined recursively: $\mathit{reach}(\sigma, \varphi) := \bigcup_{k \in \mathbb{N}} \mathit{reach}(\sigma, \varphi, k)$ where $\mathit{reach}(\sigma, \varphi, 0) := \{s \in \mathcal{S} \mid s \vDash \varphi\}$ and $\mathit{reach}(\sigma, \varphi, k+1) := \{\sigma(s) \mid s \in \mathit{reach}(\sigma, \varphi, k)\}$. The restriction $\sigma{\downarrow_\varphi}$ of σ to precondition φ is defined as $\{(s_1, s_2) \in \sigma \mid s_1 \in \mathit{reach}(\sigma, \varphi)\}$.

Note that $\mathit{reach}(\sigma, \mathit{true}) = \mathcal{S}$ and $\sigma{\downarrow_{\mathit{true}}} = \sigma$ for any σ.

5.3.1 Guarantees

In order to establish that a thread t of a concurrent system $(\mathcal{T}, \Sigma)$ satisfies a thread specification $(\mathit{pre}_t, \mathit{rely}_t, \mathit{guar}_t, R_t, M_t)$, we need to prove that—under the assumption that environment steps are constrained by rely condition rely_t and R_t—for its own steps, it only writes to locations specified in M_t and that it fulfills the two state invariant guar_t. This relation needs to be proven for any two succeeding states in the trace. The precondition pre_t will be used to relax the 'guarantee' proof obligation in a way such that it only needs to hold for states resulting from the creation of new threads; see below in Sect. 5.4.2.

The property that only locations in M_t may be changed throughout the program execution is also known as a *strict modifies clause*, which is essential to concurrency verification. This is in contrast to *weak modifies* properties

[5] In the literature on framing (cf. [Grahl et al., 2016]; Sect. 8.2.3), often the notion of a *footprint* of an expression is used. A footprint denotes the set of locations that the expression depends on, thus the footprint of a formula φ is the smallest φ-invariant location set.

as imposed in standard JavaDL [Beckert et al., 2007a; Weiß, 2011], that still allow locations outside M_t to be changed temporarily. We use the special boolean variable `estp` to distinguish between environment steps and 'own' steps of the thread under investigation, as introduced in Sect. 3.4. I.e., on a program trace, `estp` is true if and only if an environment step was performed last (cf. Lemma 3.19). A proof obligation can be formulated in CDTL using the trace modality as

$$\{h^{\text{pre}} := \texttt{heap}\}[\![\pi_t]\!]\bullet\Box(\texttt{estp} \vee (\mathit{frame}_t \wedge \mathit{guar}_t)) \tag{5.1}$$

where frame_t stands for the following formula:

$$\forall F{:}\mathbb{F}.\ (F \mathrel{\dot\in} \{\texttt{heap} := h^{\text{pre}}\}M_t \vee \mathit{select}(\texttt{heap}, F) \doteq \mathit{select}(\texttt{heap'}, F)) \tag{5.2}$$

The formula frame_t is similar to the one used by Weiß [2011, Sect. 6.4.1] to formalize weak modifies properties. The values of the location set expressions are state dependent, but evaluate in the initial state of the trace. This is assured through the updates, that store the initial heap h^{pre}. A difference is that not only the very first and the final state are in relation, but every pair of consecutive states that are produced by 'own' write actions.[6] The second part of the formula entails the two state invariant property. Note that the proof obligation of (5.1) could be written as two separate ones, since the formula $[\![\pi]\!]\bullet\Box(\psi \vee (\varphi_1 \wedge \varphi_2))$ is equivalent to $[\![\pi]\!]\bullet\Box(\psi \vee \varphi_1) \wedge [\![\pi]\!]\bullet\Box(\psi \vee \varphi_2)$.

Lemma 5.6. *Let s_0 be a state; let t be a thread with a thread specification $(\mathit{pre}_t, \mathit{rely}_t, \mathit{guar}_t, R_t, M_t)$. Let $\tau = \mathit{trc}_\Sigma(s_0\{h^{\text{pre}} \mapsto \texttt{heap}^s\}, \pi_t)$. If $\tau \vDash \bullet\Box(\texttt{estp} \vee (\mathit{frame}_t \wedge \mathit{guar}_t))$ (as in (5.1)), then*

1. *guar_t describes a binary relation $\gamma_t \supseteq \sigma_t$ and*
2. *σ_t respects $(M_t^{\complement})^{s_0}$.*

Proof. Ad 1: The trace of an interleaved program contains both environment steps (i.e., σ_Σ^*-steps) and 'own' steps (i.e., σ_t-steps). Without loss of generality, assume $|\tau| > 1$. According to Lemma 3.19, for all $i > 0$ with $\tau[i] \not\vDash \texttt{estp}$, the step from $\tau[i-1]$ to $\tau[i]$ is induced by an 'own' step. It further follows from Lemma 3.19 that γ_t is a binary relation on consecutive heaps in the trace (projected to σ_t and heaps).

Ad 2: The two-state formula frame_t formalizes $s' \approx_{\mathcal{L}} s''$ with $\mathcal{L} = \left(M_t^{\complement}\right)^{s'''}$ for all states $s', s'', s''' \in \mathcal{S}$ with $\texttt{heap'}^{s'} = \texttt{heap}^{s''}$ and $(h^{\text{pre}})^{s'} = \texttt{heap}^{s'''}$ (cf. Def. 5.1). From the assumption, frame_t is valid for $s''' = s_0$ (notice the update to M_t in the formula) and $s' = s_i$ for any $i \in [1, |\tau|)$ and

[6] Note that we do not have to use a temporal construct to refer to the previous state (there are no past operators in our logic, anyways), but through the variable `heap'` since we do not need the complete state, but just the heap state.

$\tau[i] \not\models \texttt{estp}$. Lemma 3.19 then gives $s'' = s_{i-1}$. Thus it is $s_{i-1} \approx_{(M_t^{\mathfrak{C}})^{s_0}} s_i$ and $\tau[i] = \sigma_t(\tau[i-1])$, for all $i \in [1, |\tau|)$. By reflexivity and transitivity of $\approx$ (Lemma 5.2), it follows $s_i \approx_{(M_t^{\mathfrak{C}})^{s_0}} s_j$. ◁

Remark. A slight alteration (i.e., removing 'next' and setting an initial value for `heap'`) of Formula 5.1 to

$$\{\texttt{heap'} := \texttt{heap} \mid\mid h^{\text{pre}} := \texttt{heap} \mid\mid \texttt{estp} := \mathit{false}\}[\![\pi_t]\!]\Box(\texttt{estp} \vee (\mathit{frame}_t \wedge \mathit{guar}_t))$$

requires guar_t to describe a reflexive relation. (The formula guar_t is true in the initial state of τ. The only restriction on τ is that $\texttt{heap'}^{\tau[0]} = \texttt{heap}^{\tau[0]}$, thus the result is universally valid.)

The above formula considers all possible initial states, which is impractical. To restrict the possible initial states in formula (5.1), we relax this formula using a precondition. The following formula is valid in all states in which (5.1) is valid or the formula pre_t is not valid. Hence, guar_t describes a relation γ on the restricted transition relation, as stated by the ensuing lemma.

$$\mathit{pre}_t \rightarrow \{h^{\text{pre}} := \texttt{heap}\}[\![\pi_t]\!]\bullet\Box(\texttt{estp} \vee (\forall F : \mathit{Field}.\ (F \mathrel{\dot\in} \{\texttt{heap} := h^{\text{pre}}\}M_t \vee \mathit{select}(\texttt{heap}, F) \doteq \mathit{select}(\texttt{heap'}, F)) \wedge \mathit{guar}_t)) \tag{5.3}$$

Lemma 5.7. *Let everything be as in Lemma 5.6. If Formula* (5.3) *is valid, then* $\sigma_t{\downarrow_{\mathit{pre}_t}} \subseteq \gamma_t$ *and* $\sigma_t{\downarrow_{\mathit{pre}_t}}$ *respects* $M_t^{s_0}$*; where* $\gamma_t \subseteq \mathcal{S}^2$ *is the relation described by* guar_t*.*

Proof. Assume (5.3) valid, fix some state $s \in \mathcal{S}$. If $s \not\models \mathit{pre}_t$, then $s \notin \operatorname{dom}\sigma{\downarrow_{\mathit{pre}_t}}$. If $s \models \mathit{pre}_t$, then (5.1) must be valid in s, and Lemma 5.6 applies with σ_t replaced by $\sigma_t{\downarrow_{\mathit{pre}_t}}$ applies. ◁

5.3.2 Rely Conditions

The idea of rely conditions is that they impose an upper bound on environment actions. This entails two items: 1. Since a `release` denotes an environment macro step (i.e., zero or more atomic environment steps), rely conditions must describe reflexive and transitive relations. 2. They must not be stronger than the combined guarantees that the environment provides. These items are formalized in (5.4) and (5.6) below, respectively. While (5.4) is a 'local' property—i.e., it is a property of one rely condition alone—property (5.6) is concerned with the relation of rely conditions to guarantee conditions in the combined system.

$$\begin{aligned}\forall h_1, h_2, h_3{:}\mathbb{H}.\ (\quad & \{\texttt{heap'} := h_1 \mid\mid \texttt{heap} := h_1\}rely_t \\ \wedge\ (\quad & \{\texttt{heap'} := h_1 \mid\mid \texttt{heap} := h_2\}rely_t \\ \wedge\ & \{\texttt{heap'} := h_2 \mid\mid \texttt{heap} := h_3\}rely_t \quad) \\ \rightarrow\ & \{\texttt{heap'} := h_1 \mid\mid \texttt{heap} := h_3\}rely_t \quad)\end{aligned} \tag{5.4}$$

Lemma 5.8 (Reflexivity/transitivity of *rely*)**.** *Let φ be the formula in* (5.4)*. It is a valid formula if and only if $rely_t$ describes a binary relation $\rho_t \subseteq \mathcal{S}^2$ that is reflexive and transitive.*

Proving the lemma is trivial since (5.4) is a direct formalization of the property. Note that, in general, guarantees are allowed to describe relations that are neither reflexive nor transitive—while this is required by Jones. The essential point is that rely conditions are reflexive/transitive, as mentioned above. Since the union of transitive relations is not necessarily transitive again, we still need (5.4) as a proof obligation anyway.

Of course, there is always a strongest rely condition to fulfill obligations (5.4) and (5.6). It is the reflexive/transitive closure of the union of guarantee conditions. Since, however, transitive closure cannot be expressed in plain first order logic, see, e.g., [Ebbinghaus and Flum, 1995], we require these conditions here explicitly and charge the responsibility on the specifier.

System Properties

Similar to the functional rely condition, also the frame conditions for threads of a system need to be aligned. The following formula states that the locations that t relies on not being changed are disjoint with the locations changed by any other thread t'.

$$\left(\dot{\bigcup_{t' \in T \setminus t}} M_{t'}\right) \dot{\cap}\, R_t \doteq \dot{\varnothing} \tag{5.5}$$

Lemma 5.9. *Let T be a thread pool. Formula* (5.5) *is valid if and only if, for all threads $t \neq t' \in T$, it is $\approx_{(M_{t'}^{\complement})^s} \subseteq \approx_{R_t^s}$, for any state $s \in \mathcal{S}$.*

Proof. Formula (5.5) formalizes that the sets $M_{t'}^s$ and R_t^s are disjoint for any $s \in \mathcal{S}$. Since (5.5) is universally valid, it is a direct formalization of the disjointness property (cf. location set theory by Weiß [2011]) for all $s \in \mathcal{S}$. Equivalence of the latter two follows from elementary set theory. Standard set theory gives us that disjointness is equivalent to $(M_{t'}^{\complement})^s \supseteq R_t^s$, which is equivalent to $\approx_{(M_{t'}^{\complement})^s} \subseteq \approx_{R_t^s}$, according to Lemma 5.3. ◁

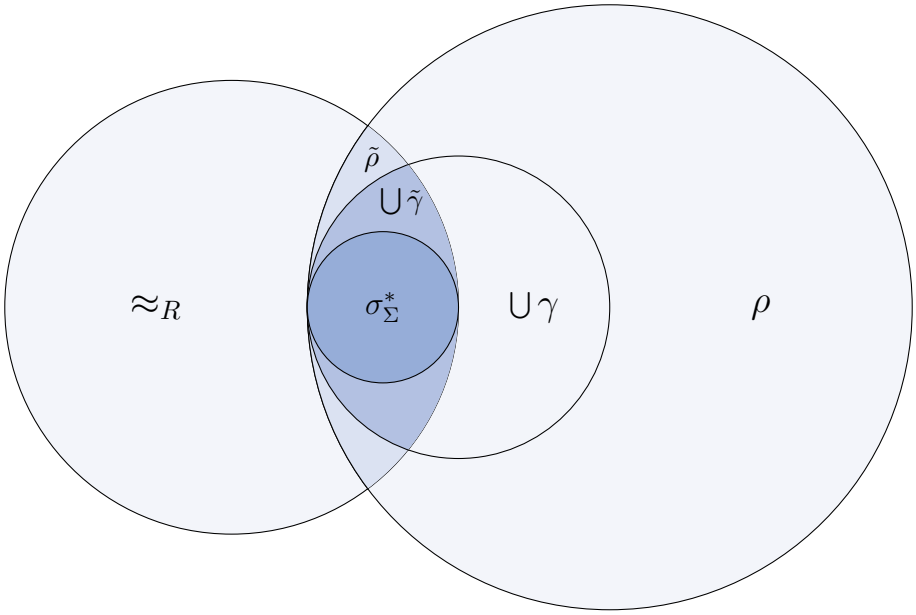


Figure 5.1: Inclusion and intersection of the relations used in this chapter if the proof obligations are valid

As already mentioned above, rely conditions must not be stronger than the combined guarantees of the system. This is formalized in (5.6) below. However, we relax the proof obligation by restricting the pairs of states that we look at to only those that respect R. The resulting condition is weaker, but still sufficient, since our overall goal includes proving R-invariance of any interleaving, anyhow. We define the 'tilde' version $\tilde{\alpha}$ of a relation $\alpha \subseteq \mathcal{S}^2$ as a shorthand for $\alpha \cap \approx_{R_t}$ where $t \in \mathcal{T}$ should be clear from the context. Figure 5.1 displays how these relations are meant to intersect with each other.

$$\{\texttt{heap'} := \texttt{heap} \mid\mid \texttt{heap} := \mathit{anon}(\texttt{heap}, R_t^{\complement})\} \left(\left(\bigvee_{t' \in T \setminus t} \mathit{guar}_{t'} \right) \rightarrow \mathit{rely}_t \right) \tag{5.6}$$

Lemma 5.10. *Let $\gamma_t, \rho_t \subseteq \mathcal{S}^2$ be the relations introduced in Lemmas 5.6 and 5.8 for some $t \in \mathcal{T}$. If formula* (5.6) *is valid, then it is $\bigcup_{t' \in T \setminus t} \tilde{\gamma}_{t'} \subseteq \tilde{\rho}_t$.*

Proof. The part of the formula on the right hand side (without the update) is a straightforward formalization of the property $\bigcup_{t' \in T \setminus t} \gamma_{t'} \subseteq \rho_t$ (cf. Lemmas 5.6 and 5.8). The update weakens this result as it restricts the pairs of states that are related to $\approx_{R_t}$, according to Lemma 5.4. Thus it is $((\bigcup_{t'} \gamma_{t'}) \cap \approx_{R_t}) \subseteq (\tilde{\rho}_t \cap \approx_{R_t})$. ◁

5.3.3 Valid Thread Specifications

For the correctness of the rely/guarantee method, we need to establish the notion of valid thread specifications w.r.t. a particular thread pool. The rule for reasoning about interleavings can then be defined in a thread pool agnostic way. We use three different levels of validity: 1. universal validity, 2. validity modulo precondition (VMP), and 3. local validity; where each one includes the subsequent item.

Definition 5.11 (Valid thread specification)**.** Let $t \in \mathcal{T}$ be a thread.

1. A thread specification $(pre_t, rely_t, guar_t, R_t, M_t)$ is *universally valid* for a thread pool $T \in 2^{\mathcal{T}}_{\text{fin}}$ if the formulae (5.1), (5.4), (5.5), and (5.6) are universally valid.

2. A thread specification is *valid modulo precondition (VMP)* if the formulae (5.1), (5.4), (5.5), and (5.6) are valid for all states $s \in reach(\sigma_t, pre_t)$.

3. A thread specification is *locally valid* if only formulae (5.3) and (5.4) are valid, i.e., t fulfils its guarantees (assuming the precondition) and the rely condition is reflexive and transitive.

4. We call the set $\{(pre_t, rely_t, guar_t, R_t, M_t) \mid t \in T\}$ a univerally valid/ VMP thread specification for T if all $(pre_t, rely_t, guar_t, R_t, M_t)$ are universally valid/VMP thread specifications w.r.t. T.

We use the simpler term 'valid' in contexts which apply to both universal valid and VMP thread specifications. Note that the semantical thread pool T is independent of the state of evaluation of the formulae.

In another point, the properties denoted by formulae (5.1), its derivative (5.3), and (5.4) are thread-local, i.e., if they are valid for a particular environment, then they are also valid for *any* environment. This is the reason to introduce local validity. The properties denoted by formulae (5.6) and (5.5) refer to the concrete, complete concurrent system instead. But they do not contain any program, only first order formulas over the theories of location sets and heaps.

Lemma 5.12. *A VMP thread specification is also locally valid.*

In contrast to the proof obligations by Jones [1983]; Stirling [1988]; Xu et al. [1997], we do not include a postcondition in our specification framework. The reason is to decouple the proof of well-behaved concurrency from proofs for other properties, like functional correctness or information flow security.

Theorem 5.13. *Let $(\mathcal{T}, \Sigma)$ be a concurrent system and $t \in \mathcal{T}$ some thread. If there is a universally valid thread specification for $\mathcal{T}$, then 1. we have $\sigma^*_\Sigma(t) \subseteq \rho_t$, where ρ_t is the semantical relation represented by $rely_t$; and 2. $\sigma^*_\Sigma(t)$ respects R_t.*

Proof. According to its definition, $\sigma^*_\Sigma(t)$ consists of atomic environment transitions $\sigma^*_\Sigma(t) = \sigma_{t_1} \circ \cdots \circ \sigma_{t_k}$ with (not necessarily different) threads $t_i \in \mathcal{T} \setminus \{t\}$ and $k \in \mathbb{N}$ as determined by Σ. We start showing the 'respect part' (2) of the theorem. Ad part 2: Lemma 5.7 gives us $\sigma_{t_i} \subseteq \approx_{M^{\mathfrak{C}}_{t_i}}$. From Lemma 5.9, it follows $\sigma_{t_i} \subseteq \approx_{R_t}$. Since $\approx$ is an equivalence relation (Lemma 5.2), we obtain $\sigma^*_\Sigma(t) \subseteq \approx_{R_t}$. Ad part 1: Let $\tilde{\gamma}$ and $\tilde{\rho}$ be the relations introduced in Lemma 5.10. According to Lemma 5.7, it is $\sigma_{t_i} \subseteq \gamma_{t_i}$ and thus $\sigma^*_\Sigma(t) \subseteq \gamma_{t_1} \circ \cdots \circ \gamma_{t_k}$. We may further widen this inclusion by replacing any concrete γ_{t_i} by the union $\bigcup_{t' \neq t} \gamma_{t'}$, i.e.,

$$\sigma^*_\Sigma(t) \subseteq \left(\bigcup_{t' \neq t} \gamma_{t'} \right)^k \quad .$$

Lemma 5.10 provides $\bigcup_{t'} \tilde{\gamma}_{t'} \subseteq \tilde{\rho}_t \subseteq \rho_t$. Since we already proved $\sigma^*_\Sigma(t) \subseteq \approx_{R_t}$, it follows $\sigma^*_\Sigma(t) \subseteq \tilde{\rho}^k_t$. Since ρ_t is reflexive and transitive (Lemma 5.8), i.e., $\rho_t = \rho^k_t$ for any $k \in \mathbb{N}$, we conclude $\sigma^*_\Sigma(t) \subseteq \rho_t$. ◁

Corollary 5.14. *Let $(\mathcal{T}, \Sigma)$ be a concurrent system and $t \in \mathcal{T}$ some thread. If there is a VMP thread specification for $\mathcal{T}$, then 1. we have $(\sigma^*_\Sigma(t))\downarrow_{pre_t} \subseteq \rho_t\downarrow_{pre_t}$ and 2. $(\sigma^*_\Sigma(t))\downarrow_{pre_t}$ respects R_t, where pre_t is the precondition in the thread specification for t.*

5.4 A Calculus for Concurrent DTL

We present the calculus $\mathcal{C}_{\text{CDTL}}$ for the full CDTL. It consists of most of the rules of the calculus $\mathcal{C}_{\text{DTL}}$ for DTL as defined in Sect. 4.4 (and originally by Beckert and Bruns [2013]). The only exceptions are that the original invariant rules R28–R30 are replaced with those defined in Tab. 4.11 on page 84. As explained in Sects. 3.6 and 4.6.3, the original invariant rules are sound w.r.t. the semantics provided, but are inappropriate to model concurrent behavior. We additionally define rules R37 and R38 for the symbolic execution of interleavings and dynamic thread creation, to be introduced in the following subsections.

5.4.1 Reasoning About Interleavings

We now define a calculus rule that can be applied on a program modality where `release` is the active statement, performing symbolic execution. This rule R37, shown below, is the centerpiece of our CDTL calculus. By assigning a calculus rule to the synthetic `release` statement, we can establish that it is sufficient for a sequential program π to be correct w.r.t. a given specification in a concurrent setting if the instrumented program $\underline{\pi}$ is, as described in Sect. 3.6. We use the results of the previous section (in particular Thm. 5.13) to prove soundness of this rule.

$$\frac{\Gamma, \mathcal{U}\mathcal{V} rely_t \Longrightarrow \mathcal{U}\mathcal{V}[\![\omega_t]\!]\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{release; } \omega_t]\!]^{\bullet}_{\circ}\varphi, \Delta}\ \mathsf{R37}$$

where $\mathcal{V} = \{\texttt{heap'} := \texttt{heap} \parallel \texttt{heap} := anon(\texttt{heap}, R_t^{\complement}) \parallel \texttt{threads} := \texttt{threads} \dot{\cup} T \parallel \texttt{estp} := true\}$, T is a fresh symbol of type $\mathbb{T}$, and t is the thread whose program appears in the modality.[7] The symbol $^{\bullet}_{\circ}$ stands for either 'weak next' ($\bullet$) or 'strong next' ($\circ$).

The effect of the above rule is that the `release` statement in the conclusion is symbolically executed in a scheduler-independent way. (Recall that universal validity is defined as validity under any scheduler, cf. Def. 4.7.) As `release` denotes a step, the 'next' operator is removed in the premiss. Since there is a remaining trace (induced by ω_t), this applies to both 'weak next' and 'strong next.' In the premiss, the effect of `release` on the heap and the thread pool is described through the 'havoc' update $\mathcal{V}$. For both heap and thread pool, we have some partial knowledge: the heap is updated to another heap that coïncides with the old heap on R_t; the thread pool can only increase—only the additional part, represented by T, is unknown. To compensate the havoc, the $rely_t$ formula is inserted on the left hand side of the premiss sequent. It functionally specifies the environment changes.

We follow the usual approach in software verification that specifications live as background theories and therefore are not part of formulae or sequents. This has been pursued by, e.g., Beckert et al. [2007a]; Weiß [2011] for method contracts in sequential programs. For concurrent programs, there is a similar situation with rely conditions. This means that we cannot assess the soundness of rule R37 on grounds of the rule itself, but only w.r.t. the specification framework of thread specifications.

Theorem 5.15 (Conditional soundness of R37)**.** *Assume that the premiss of rule* R37 *is valid. Let $s \in \mathcal{S}$ be some state that includes the effects of the update $\mathcal{U}$. If there exists a VMP thread specification for* $\texttt{threads}^s$ *and $s \in reach(\sigma_t, pre_t)$, then the conclusion is valid.*

Proof. For the general case, this proof is extensive. Without loss of generality, we apply some simplifications. We prove the claim for the 'strong next' operator; the proof for the 'weak next' operator follows the same lines. Assume the sequent $\mathcal{U}\mathcal{V} rely_t \Longrightarrow \mathcal{U}\mathcal{V}[\![\omega_t]\!]\varphi$ valid. According to Lemma 4.12, it suffices to prove validity of the sequent $\Longrightarrow \mathcal{U}[\![\texttt{release; } \omega_t]\!]\circ\varphi$. As the update $\mathcal{U}$ appears as prefix to all formulae involved, we may also disregard it. Fix a state $s \in reach(\sigma_t, pre_t)$ with $s \vDash \mathcal{V} rely_t$ and $s \vDash \mathcal{V}[\![\omega_t]\!]\varphi$ as in the claim. Since we already have assumed reachability from the precondition, we omit the restriction notation $\downarrow_{pre_t}$ from Def. 5.5 in the following for the sake of readability.

[7] We assume that this is derivable from the program context.

Let s' be the state which coïncides with s, except for the effect of update $\mathcal{V}$, i.e., $s' = s\{\texttt{heap'} \mapsto \texttt{heap}^s, \texttt{heap} \mapsto h_{anon}, \texttt{threads} \mapsto \texttt{threads}^s \cup T^s, \texttt{estp} \mapsto \mathit{true}\}$ where $h_{anon} \in \mathcal{D}_{\mathbb{H}}$ is an arbitrary, but fixed, heap mapping such that $h_{anon} \approx_{R_t^s} \texttt{heap}^s$. We represent the fact that $s' \vDash \mathit{rely}_t$ through the relation $\rho_t(s, s'')$ where it exists some state s'' such that $s \approx_{R_t^s} s''$ (i.e., s'' is described through the function *anon*).

Given that we have a valid thread specification for thread pool $\mathcal{T} := \texttt{threads}^s$ and following Thm. 5.13 and Corollary 5.14, we may choose $s'' = \sigma^*_\Sigma(s, t)$, where $t = \Sigma(s)$ and Σ is some scheduler. It follows that $s' \vDash [\![\omega_t]\!]\varphi$ is equivalent to $s''' \vDash [\![\omega_t]\!]\varphi$ with $s''' := s''\{\texttt{heap'} \mapsto \texttt{heap}^s, \texttt{estp} \mapsto \mathit{true}\}$. This is equivalent (cf. Def. 4.7) to $\mathit{trc}_\Sigma(s''', \omega) \vDash \varphi$. It follows that $\langle s \rangle \cdot \mathit{trc}_\Sigma(s''', \omega) \vDash \circ\varphi$ is also valid, which is equivalent to $\mathit{trc}_\Sigma(s, \texttt{release; } \omega) \vDash \circ\varphi$ (cf. Def. 3.17), which concludes the proof. ◁

Abadi and Lamport [1995]; Gotsman et al. [2009] point out that the classical rely/guarantee approach—using two-state rely conditions—is unsound for reasoning about global liveness properties. This does not apply to our CDTL calculus, as we only consider temporal properties that are *local* to a thread. All environment transitions are modeled as a single step in the trace of the local thread (cf. Def. 3.17). Since we assume a fair scheduler (cf. Def. 3.15), the number of atomic environment transitions represented by one `release` is always finite. Furthermore, given the absence of locks (and thus deadlocks) in our model, it is obvious that there is always progress in a concurrent execution. A relaxation of our concurrency model to allow unfair schedulers would incorporate the possibility of livelocks. An interleaving semantics (along with the appropriate reasoning) in the presence of livelocks and deadlocks will be part of future work.

Note that soundness of Rule R37 is independent of the kind of modality and the formula φ. This means that a derived rule using the $[\cdot]$ ('box') or the $\langle\cdot\rangle$ ('diamond') modality from standard dynamic logic is also sound as those modalities can be expressed using trace formulae.

The program instrumentation with the synthetic `release` statement (cf. Sect. 3.6) and this corresponding calculus rule allow to extend the present calculus for purely sequential DTL in a conservative manner. As explicit releases do not appear in mainstream real-world programing languages, it is naturally desirable to overcome this instrumentation. In Sect. 7.3, we devise calculus rules to be implemented for Java, in the KeY prover. These do not involve explicit releases, but instead anonymization at any reading assignment or assertion. Obviously, a formal proof of conservatism for that calculus would involve considerably more effort.

5.4.2 Reasoning About Thread Creation

In this section, we define the second additional rule for our calculus, dealing with dynamic thread creation. Below, we will introduce a symbolic execution rule R38 that can be applied on a program modality where `fork` is the active statement. Proving soundness of this rule is trivial. However, an interesting property for *overall* soundness of the calculus is preservation of thread specification validity, thus enabling validity of R37 (cf. Thm. 5.15). As we have defined it in Def. 5.11, a thread specification is only valid *for some thread pool*. When we expand the thread pool through executing `fork`, we have to maintain validity for the expanded pool.

The following theorem states which conditions are necessary to extend a thread pool while preserving validity of thread specifications. This frees us from unwieldy proof obligations that are stated in terms of 'for all threads.'

Theorem 5.16 (Thread pool expansion)**.** *Let T be a thread pool and let $S_T = \{(pre_t, rely_t, guar_t, R_t, M_t) \mid t \in T\}$ be a valid thread specification for T. Let $t'' \notin T$ be another thread with locally valid specification $S_{t''} = (pre_{t''}, rely_{t''}, guar_{t''}, R_{t''}, M_{t''})$. If the following formulae are valid for any $t \in T$, then $S_T \cup \{S_{t''}\}$ is a valid thread specification for $T \cup \{t''\}$.*

(a) $\{\texttt{heap'} := \texttt{heap} \parallel \texttt{heap} := anon(\texttt{heap}, R_t^{\complement} \dot\cup M_t)\}((rely_t \vee guar_t) \rightarrow rely_{t''})$

(b) $\{\texttt{heap'} := \texttt{heap} \parallel \texttt{heap} := anon(\texttt{heap}, R_t^{\complement})\}(guar_{t''} \rightarrow rely_t)$

(c) $(R_t^{\complement} \dot\cup M_t) \dot\cap R_{t''} \doteq \dot\varnothing$

(d) $M_{t''} \dot\cap (M_t^{\complement} \dot\cup R_t) \doteq \dot\varnothing$

Following this theorem, we can expand a purely symbolic thread specification system. The original thread pool T does not appear in the formulae to be proven, but only one single thread t.

Proof. Fix some $t \in T$. S_t is locally valid by construction (i.e., the proof obligations do not depend on T, cf. Def. 5.11). As S_t and $S_{t''}$ are locally valid, it remains to show that formulae (5.5) and (5.6)—expressing disjointness of frames and that rely conditions are not stronger than guarantees—are valid for t and t'' w.r.t. the expanded thread pool $T \cup \{t''\}$.

Ad (5.5): From t having a valid thread specification w.r.t. T, we get $\vDash \dot\bigcup_{t' \in T \setminus t} M_{t'} \subseteq R_t^{\complement}$ and $\vDash \dot\bigcup_{t' \in T \setminus t} R_{t'} \subseteq M_t^{\complement}$ (cf. Lemma 5.9). Replacing $R_t^{\complement}$ and $M_t^{\complement}$ in formulae (c) and (d), respectively, gives us $\vDash \dot\bigcup_{t' \in T} M_{t'} \dot\cap R_{t''} \doteq \emptyset$ and $\vDash M_{t''} \dot\cap \dot\bigcup_{t' \in T} R_{t'} \doteq \emptyset$, that are equivalent to (5.5) for t or t'', respectively.

Ad (5.6): For t, this immediately follows from (b). For t'', the formula (a) can be weakened according to Lemma 5.4(2). Let $\mathcal{V}_L$ stand for the update $\{\texttt{heap'} := \texttt{heap} \parallel \texttt{heap} := \mathit{anon}(\texttt{heap}, L)\}$. We obtain $\vDash (\mathcal{V}_{R_t^{\complement}} \mathit{rely}_t \vee \mathcal{V}_{R_t^{\complement} \dot\cup M_t} \mathit{guar}_t) \rightarrow \mathcal{V}_{R_t^{\complement} \dot\cup M_t} \mathit{rely}_{t''}$ by update distributivity and Lemma 5.4(2) applied on rely_t. From the valid thread specification for t w.r.t. T, we obtain $\vDash \mathcal{V}_{R_t^{\complement}}(\mathit{guar}_{t'} \rightarrow \mathit{rely}_t)$ for all $t' \in T$. From (c), it follows that this is equivalent to $\vDash \mathcal{V}_{R_t^{\complement} \dot\cup M_t} \mathit{guar}_{t'} \rightarrow \mathcal{V}_{R_t^{\complement}} \mathit{rely}_t$. We then replace $\mathcal{V}_{R_t^{\complement}} \mathit{rely}_t$ by $\mathcal{V}_{R_t^{\complement} \dot\cup M_t} \mathit{guar}_{t'}$ in the above formula to obtain $\vDash (\mathcal{V}_{R_t^{\complement} \dot\cup M_t} \mathit{guar}_{t'} \vee \mathcal{V}_{R_t^{\complement} \dot\cup M_t} \mathit{guar}_t) \rightarrow \mathcal{V}_{R_t^{\complement} \dot\cup M_t} \mathit{rely}_{t''}$. Pulling out the update and applying Lemma 5.4(2) with formula (c) finally leads us to $\vDash \mathcal{V}_{R_{t''}^{\complement}}((\mathit{guar}_{t'} \vee \mathit{guar}_t) \rightarrow \mathit{rely}_{t''})$, that is what we needed to prove. ◁

We use this result to cast it into a symbolic execution rule for `fork`. The following calculus rule deals with the creation of new threads in the program modality through symbolic execution.

$$\frac{\begin{array}{cc} \text{(a)}\ \Gamma \Longrightarrow \mathcal{U}\mathcal{W}[\![\omega_t]\!]\varphi, \Delta & \text{(d)}\ \Gamma \Longrightarrow \mathcal{U}\mathit{pre}_{t'}, \Delta \\ \text{(b)}\ \Longrightarrow \mathcal{V}_1(\mathit{guar}_{t'} \rightarrow \mathit{rely}_t) & \text{(e)}\ \Longrightarrow M_{t'} \mathbin{\dot\cap} (R_t \mathbin{\dot\cup} M_t^{\complement}) \doteq \dot\varnothing \\ \text{(c)}\ \Longrightarrow \mathcal{V}_0((\mathit{rely}_t \vee \mathit{guar}_t) \rightarrow \mathit{rely}_{t'}) & \text{(f)}\ \Longrightarrow (R_t^{\complement} \mathbin{\dot\cup} M_t) \mathbin{\dot\cap} R_{t'} \doteq \dot\varnothing \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{fork \{}\pi\texttt{\}; }\ \omega_t]\!]\varphi, \Delta}\ \mathsf{R38}$$

where t is the current thread, $t' \in \mathcal{T}$ is a fresh thread with program π (modulo renaming of local variables), $\mathcal{V}_0$ stands for the update $\{\texttt{heap'} := \texttt{heap} \parallel \texttt{heap} := \mathit{anon}(\texttt{heap}, R_t^{\complement} \mathbin{\dot\cup} M_t)\}$, $\mathcal{V}_1$ stands for the update $\{\texttt{heap'} := \texttt{heap} \parallel \texttt{heap} := \mathit{anon}(\texttt{heap}, R_t^{\complement})\}$, and $\mathcal{W}$ stands for the update $\{\texttt{threads} := \texttt{threads} \mathbin{\dot\cup} \{t'\}\}$; and $(\mathit{pre}_t, \mathit{rely}_t, \mathit{guar}_t, R_t, M_t)$ is a (not necessarily valid) thread specification for t (respectively for t').

Since the `fork` statement does not induce a step, Rule R38 can be applied to any formula φ. It includes 6 premisses, where premiss (a) captures the (local) effect of forking a new thread, namely the thread pool expansion expressed by update $\mathcal{W}$. The other premisses represent additional proof obligations that the fresh thread does not destroy thread specification validity. Premiss (d) has the precondition for the fresh thread on the right hand side. In this branch, the sequent environment Γ, Δ is still present, i.e., we prove the precondition for the (symbolic) state in which the thread is forked. The remaining premisses correspond to the formulae of Thm. 5.16. For this theorem, they need to be universally valid; for this reason, the sequent environment is not present.

Lemma 5.17. *Rule* R38 *is sound.*

Proof. Let premiss (a) be valid, then the conclusion follows from Def. 3.10. ◁

Since soundness of Rule R38 does not depend on premisses (c)–(f), we could devise a simpler sound rule with only premiss (a). But the interesting property is the propagation of thread specification validity. It ensures that subsequent applications of R37 are actually sound (cf. Thm. 5.15)—otherwise we could not close a proof except for trivial postconditions. This propagation is expressed through the premisses (b), (c), (e), and (f). As these premisses contain exactly the formulae of Thm. 5.16, the following lemma follows directly from the theorem.

Lemma 5.18 (Propagation of thread specification validity)**.** *Let T be a thread pool for some concurrent system. Let S be a valid thread specification for T. If the premisses* (b)*,* (c)*,* (e)*, and* (f) *in* R38 *are valid, and if formulae* (5.3) *and* (5.4) *are valid, then $S \cup (pre_{t'}, rely_{t'}, guar_{t'}, R_{t'}, M_{t'})$ is a valid thread specification for $T \cup \{t'\}$.*

Finally, through premiss (d), a closed proof includes validity of a precondition $pre_{t'}$. This allows the 'guarantee' proof obligation for thread t to be relaxed to states reachable from $pre_{t'}$ (cf. (5.3)). In the following section, we use this property to prove that the calculus is relatively sound.

5.4.3 Soundness

We first observe the rules for the purely sequential DTL, as presented in Chap. 4, are also sound for the concurrent setting.

Lemma 5.19. *The $\mathcal{C}_{\mathrm{DTL}}$ rules* R1–R35*, including the modified invariant rules* R28′*,* R29′*, and* R30′ *from Tab. 4.11 on page 84, are sound w.r.t. CDTL.*

Proof. According to Lemma 4.8, CDTL is a semantical conservative extension [Shoenfield, 1967] of DTL. This means that the soundness result of Thm. 4.17 also applies to CDTL. The modified invariant differs from the original one only in the additional instrumentation. Further instrumentation with `release` statements does not affect soundness negatively: in case the program is already sufficiently instrumented, then the σ^*_Σ-step denoted by an additional `release` is just an identity step. ◁

Together with the rules R37 and R38, the aforementioned rules constitute the calculus $\mathcal{C}_{\mathrm{CDTL}}$. We have already observed that R37 cannot be considered sound per se, but only relative to a valid thread specification. The overall (relative) soundness of the calculus depends on the interplay of rules R38 (i.e., showing that the thread specification can be expanded) and R37 (i.e., requiring a valid thread specification to be sound). Thus, proving overall soundness requires a structural analysis over the complete proof tree.

The following definition formalizes the intuitive notion of thread pools 'in a sequent.' Like rule schemata, the resulting set $\mathcal{T}(Seq)$ is to be seen schematic, i.e., it corresponds to a rule or a symbolic rule application.

Definition 5.20. Let *Seq* be a sequent. Let $\{\psi_i\}$ be the set of subformulae with a program modality as top level operator, with each ψ_i appearing in the scope of a sequential update $\mathcal{U}_i$. By $\mathcal{T}(Seq) \subseteq 2^{\mathcal{T}}_{\text{fin}}$ we denote the set of thread pools T_i such that T_i represents the equivalence class $\{s \mapsto \texttt{threads}^{s \cdot \mathcal{U}_i}\}$, where $s \cdot \mathcal{U}$ denotes the state that coïncides with s except for the effect of the update $\mathcal{U}$. For a rule application $\frac{(Seq_j)}{Seq}$, we define $\mathcal{T}\left(\frac{(Seq_j)}{Seq}\right) = \mathcal{T}(Seq)$.

Lemma 5.21. *Let $\Gamma \Longrightarrow \Delta$ be a sequent that is derivable in $\mathcal{C}_{\text{CDTL}}$ through applications $A_1, \ldots, A_n$ of rule* R37*. If there is a valid thread specification for every $T \in \bigcup_i \mathcal{T}(A_i)$, then $\Gamma \Longrightarrow \Delta$ is valid.*

Proof. The lemma depends on all rules appearing in the proof tree for $\Gamma \Longrightarrow \Delta$ being sound. All rules except R37 are sound without further prerequisites. This follows from Lemmas 5.17 and 5.19. Since there is a valid thread specification for the (symbolic) thread pool appearing in the conclusion, following Thm. 5.15, every application A_i of rule R37 is sound in its respective context, too. ◁

Note that we make no assumption about the location of the modalities within the sequent; they may appear on both sides of the sequent and may be nested. Following this intermediate result, we develop the following theorem, that states that it is sufficient for the thread pools 'in the root' to have a valid specification.

Theorem 5.22 (Relative soundness of $\mathcal{C}_{\text{CDTL}}$)**.** *Let $\Gamma \Longrightarrow \Delta$ be a derivable sequent. If there is a valid thread specification for every $T \in \mathcal{T}(\Gamma \Longrightarrow \Delta)$, then $\Gamma \Longrightarrow \Delta$ is valid.*

Proof. We show by structural induction over the proof tree (starting in the root $\Gamma \Longrightarrow \Delta$) that all thread pools appearing in a sequent have a valid specification, and thus the result of Lemma 5.21 applies. For most rules, the induction step is trivial since they do not touch the thread pool. (This means that, if thread pool appears in one of the premisses, it is also present in the conclusion.) Rule R38 is the only rule to introduce additional threads to the pool in a premiss. Since the root is derivable, i.e., the proof tree is closed, it follows from Lemma 5.18 that the new thread pool has a valid specification. ◁

5.4.4 Completeness

We believe that our calculus is almost complete, although we do not provide a formal definition of this notion of completeness or a formal proof of this property here. An argument in favor are the respective proofs by Stølen [1990] and Prensa Nieto [2002] that the original calculus by Jones [1983] is relatively complete[8] w.r.t. the concurrent program semantics by Owicki and Gries [1976]. As expected, their proofs are extensive and rich in technical detail. The proof by Stølen [1990, Chap. 18] constructs *strongest* rely/guarantee specifications and postconditions (w.r.t. a closed concurrent system) and uses auxiliary *history* variables to describe local progress. An essential intermediate result states that strongest specifications are always expressible.

However, there are some subtle differences in their and our approaches that bars us from an immediate adaptation of their results. One particular issue is the concrete definitions of schedulers; another one is the fact that our approach is completely thread-modular, whereas the aforementioned approaches regard programs that are closed under the parallel composition operator (see Sect. 5.2). We have made the fundamental assumptions that schedulers are fair (see Sect. 3.1.2). A particular result that follows from this assumption is Lemma 3.22, that states that if all threads terminate, then there is a unique final state. The rely/guarantee methodology is incomplete w.r.t. this semantics; cf. [Stølen, 1990, Sect. 19.8]. Even with strongest rely conditions, it cannot be proven that all threads have reached their final state.

Our addition of frame conditions to the approach is not an issue, since frame conditions are theoretically redundant and may be replaced by functional specifications.

Finding the appropriate notion of 'relative' completeness can be challenging itself as Stølen [1991] notes—his system is complete w.r.t. a very particular notion of fairness. Developing an appropriate notion of completeness and a formal proof thereof is left to future work. We admit that the discussion on completeness is more a theoretical argument. It seems more promising to focus future work on *practical completeness*, thus making the approach more effective in practice. In particular, this would include providing appropriate, sufficiently abstract specification means. A first step towards this goal is presented in Sect. 8.3, where we integrate rely/guarantee-style specifications into the JML specification framework. Other possible developments towards more effectiveness include more specialized rules and proof strategies that are dedicated to the goals described in this chapter.

[8] Completeness w.r.t. postconditions requires ghost variables that are updated atomically, as explained in Sect. 5.6 below.

5.5 Examples

We will discuss two examples involving two threads each. In the first example, both threads operate on disjoint heap locations. Framing specifications make it easy to prove that the threads do not interfere. In the second example, both threads read and write to a common location. The rely/guarantee conditions as well as postconditions are of the kind that this inference is well behaved. As can be seen from the examples, even for these small programs, proof sizes tend to explode (even though the proofs are displayed in a simplified way). The given examples can be seen as the limit that is manageable with pen and paper. More elaborate proofs will require appropriate tool support (see Chap. 7 and the concluding remarks in Sect. 11.3). A third (small) example is provided in Sect. 5.6 below, that includes an extension to the present approach, regarding synchronization and ghost variables.

5.5.1 Noninterfering Threads: Guarantee

In this example, we show a formal proof of the guarantee condition for a thread being satisfied, using the calculus that has been introduced above. We have two threads r and s with the uninstrumented programs $\pi_r =$ `t = X; X = t+1;` and $\pi_s =$ `t = Z; Z = t+1;` that each increment a different global variable (with intermediate storage in local variable `t`). For the sake of brevity in the presentation, we use rules for update simplification, that have not been introduced before. One replaces sequential updates by parallel in the obvious way. The other one simplifies *select* on anonymized heaps. We also simplify away logical constants *true* and *false*. See [Rümmer, 2006; Beckert et al., 2007b] for details on update simplification. We also hide formulae from the sequent that are not relevant to the remainder of the proof.

Let the following thread specification be given for thread r: $rely_r = guar_r = pre_r = true$ and $R_r = M_r = \{\texttt{X}\}$. Conversely, for thread s, we specify $rely_s = guar_s = pre_s = true$ and $R_s = M_s = \{\texttt{Z}\}$. Since the respective heap partitions on which the threads work are perfectly separated, it suffices to set the functional rely/guarantee specifications to a trivial value. The separation is conveniently expressed in the frame specification.

We show the proof for (the instrumentation of) r satisfying its guarantee condition in Fig. 5.2 on the next page. As usual in tableau-like calculi, it is to be read bottom up. The trivial guarantee condition is simplified away in the beginning. In step (1), the first `release` statement is symbolically executed (R37). The effect is that all locations except `X` are anonymized. Then, the value of `X` is read from the heap (R25). Note that the fresh update in step (2) has $select(\texttt{heap}, \texttt{X})$ on the right hand side, not an anonymized heap. The reason is that this update appears *sequentially* after the anonymizing update that was produced by (1). Only in the following step, that simplifies updates,

$$\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{anon}(\mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1), \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true}\}[\![]\!]\bullet\Box(\mathtt{estp} \vee \mathit{frame})$$

(c)

$$\dfrac{\dfrac{\dfrac{\dfrac{\Longrightarrow \mathit{true}}{\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{anon}(\mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1), \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true}\}(\mathtt{estp} \vee \mathit{frame})}}{\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{anon}(\mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1), \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true}\}[\![]\!](\mathtt{estp} \vee \mathit{frame}) \quad \rightsquigarrow\text{(c)}}}{\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{anon}(\mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1), \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true}\}[\![]\!]((\mathtt{estp} \vee \mathit{frame}) \wedge \bullet\Box(\mathtt{estp} \vee \mathit{frame})}}{\dfrac{\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{anon}(\mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1), \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true}\}[\![]\!]\Box(\mathtt{estp} \vee \mathit{frame})}{\dfrac{\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1) \mid\mid \mathtt{estp} := \mathit{false}\}\{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true}\}[\![]\!]\Box(\mathtt{estp} \vee \mathit{frame})}{\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1) \mid\mid \mathtt{estp} := \mathit{false}\}[\![\mathtt{release;}]\!]\bullet\Box(\mathtt{estp} \vee \mathit{frame})}}}$$

(b)

$$\dfrac{\dfrac{\Longrightarrow \mathit{true}}{\Longrightarrow \{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true} \mid\mid \mathtt{t} := \mathit{select}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X})\}(\mathtt{estp} \vee \mathit{frame})}}{\Longrightarrow \{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true} \mid\mid \mathtt{t} := \mathit{select}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X})\}[\![\mathtt{X = t+1;\ release;}]\!](\mathtt{estp} \vee \mathit{frame})}$$

(a)

$$F_0 \neq \mathtt{X} \Longrightarrow \mathit{true}$$

$$F_0 \neq \mathtt{X} \Longrightarrow \mathit{select}(\mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1), F_0) \doteq \mathit{select}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), F_0))$$

$$\Longrightarrow \forall F : \mathrm{Field}.(F \in \{\mathtt{X}\} \vee \mathit{select}(\mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1), F) \doteq \mathit{select}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), F))$$

$$\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1) \mid\mid \mathtt{estp} := \mathit{false}\}\mathit{frame}$$

$$\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1) \mid\mid \mathtt{estp} := \mathit{false}\}[\![\mathtt{release;}]\!](\mathtt{estp} \vee \mathit{frame}) \ (4) \quad \rightsquigarrow\text{(b)}$$

$$\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1) \mid\mid \mathtt{estp} := \mathit{false}\}[\![\mathtt{release;}]\!]((\mathtt{estp} \vee \mathit{frame}) \wedge \bullet\Box(\mathtt{estp} \vee \mathit{frame}))$$

$$\Longrightarrow \{\mathtt{heap'} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{heap} := \mathit{store}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X}, \mathit{select}(\mathtt{heap}, \mathtt{X}) + 1) \mid\mid \mathtt{estp} := \mathit{false}\}[\![\mathtt{release;}]\!]\Box(\mathtt{estp} \vee \mathit{frame})$$

$$\Longrightarrow \{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{t} := \mathit{select}(\mathtt{heap}, \mathtt{X}) \mid\mid \mathtt{estp} := \mathit{true} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement})\}\{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{X}, \mathtt{t} + 1) \mid\mid \mathtt{estp} := \mathit{false}\}[\![\mathtt{release;}]\!]\Box(\mathtt{estp} \vee \mathit{frame})$$

$$\Longrightarrow \{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{t} := \mathit{select}(\mathtt{heap}, \mathtt{X}) \mid\mid \mathtt{estp} := \mathit{true} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement})\}[\![\mathtt{X = t+1;\ release;}]\!]\bullet\Box(\mathtt{estp} \vee \mathit{frame}) \ (3)$$

$$\rightsquigarrow\text{(a)} \quad \Longrightarrow \{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true} \mid\mid \mathtt{t} := \mathit{select}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X})\}[\![\mathtt{X = t+1;\ release;}]\!]\bullet\Box(\mathtt{estp} \vee \mathit{frame})$$

$$\Longrightarrow \{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true} \mid\mid \mathtt{t} := \mathit{select}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X})\}[\![\mathtt{X = t+1;\ release;}]\!]((\mathtt{estp} \vee \mathit{frame}) \wedge \bullet\Box(\mathtt{estp} \vee \mathit{frame}))$$

$$\Longrightarrow \{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true} \mid\mid \mathtt{t} := \mathit{select}(\mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}), \mathtt{X})\}[\![\mathtt{X = t+1;\ release;}]\!]\Box(\mathtt{estp} \vee \mathit{frame})$$

$$\Longrightarrow \{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true}\}\{\mathtt{t} := \mathit{select}(\mathtt{heap}, \mathtt{X})\}[\![\mathtt{X = t+1;\ release;}]\!]\Box(\mathtt{estp} \vee \mathit{frame})$$

$$\{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement})\}\mathit{true} \Longrightarrow \{\mathtt{heap'} := \mathtt{heap} \mid\mid \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{X}\}^{\complement}) \mid\mid \mathtt{estp} := \mathit{true}\}[\![\mathtt{t = X;\ X = t+1;\ release;}]\!]\Box(\mathtt{estp} \vee \mathit{frame}) \ (2)$$

$$\Longrightarrow [\![\mathtt{release;\ t = X;\ X = t+1;\ release;}]\!]\bullet\Box(\mathtt{estp} \vee (\mathit{frame} \wedge \mathit{true})) \ (1)$$

Figure 5.2: Proof that thread r satisfies its guarantee condition

a single parallel update is produced, in which *select* has the anonymized heap as parameter. Unwinding the 'box' operator leads to a branch (a), which can be closed in two steps: `estp` is true, indicating an environment step, in which case, the guarantee condition is not to be proven.

Staying on the main proof branch, in step (3), the update is simplified again: the access to `X` on a heap that is anonymized on all locations but `X`, can be replaced by an access to `X` on the original heap. In the following step, the assignment to `X` is symbolically executed. This leaves only the final `release` statement in the program modality. The leading sequential update is simplied to a parallel update subsequently. The 'box' operator is unwound once again to yield a branch (b). Branch (b) closes easily—similar to branch (a)—with `estp` being true. However, this requires unwinding the 'box' operator for a third time, branch (c) closes with rule R22.

Back on the main branch, there is no temporal operator left in the sequent. Rule R16 removes the program modality in (4); `estp` is simplified away. The proof is finally closed through simplifications of the heap terms in the formula *frame*. For any location F_0 that is not `X`, its value is the same on the anonymized heap and the heap obtained from it by updating `X`.

5.5.2 Interfering Threads: Postcondition

The above example showed how two threads behave that are perfectly separated. Let us change the setup such that both threads work on a common location. We consider two threads concurrently increasing a variable `X` by 1: $\pi_r = \pi_s =$ `t = X; X = t+1;` It is clear to the naked eye that, in this program, there is a data race without proper synchronization. For this example, we let the data race happen and provide arguably very weak specifications. We declare thread specifications as follows:

$$rely_r = rely_s = guar_r = guar_s = \quad select(\texttt{heap}, \texttt{X}) \geq select(\texttt{heap'}, \texttt{X})$$

$$R_r = R_s = \emptyset \qquad \text{and} \qquad M_r = M_s = \{\texttt{X}\}$$

The proof obligations that the thread specifications are valid are trivial again (or at least similar to the first example above). We now prove a functional contract for π_s: given that it is non-negative in the beginning, the final value of `X` is strictly greater than zero; i.e., under precondition $select(\texttt{heap}, \texttt{X}) \geq 0$, the postcondition $select(\texttt{heap}, \texttt{X}) > 0$ holds.

Figure 5.3 on the facing page shows a proof tree with rule R37 applied. For the sake of readability, we apply some simplifications: 1. We use the 'box' modality $[\cdot]$ from classical dynamic logic here, since it can be embedded in CDTL. The rules presented for the $[\![\cdot]\!]$ modality would applied accordingly, apart from some additional branches (from unwinding the postcondition embedding) that are closed instantaneously. 2. We indicate the `release`

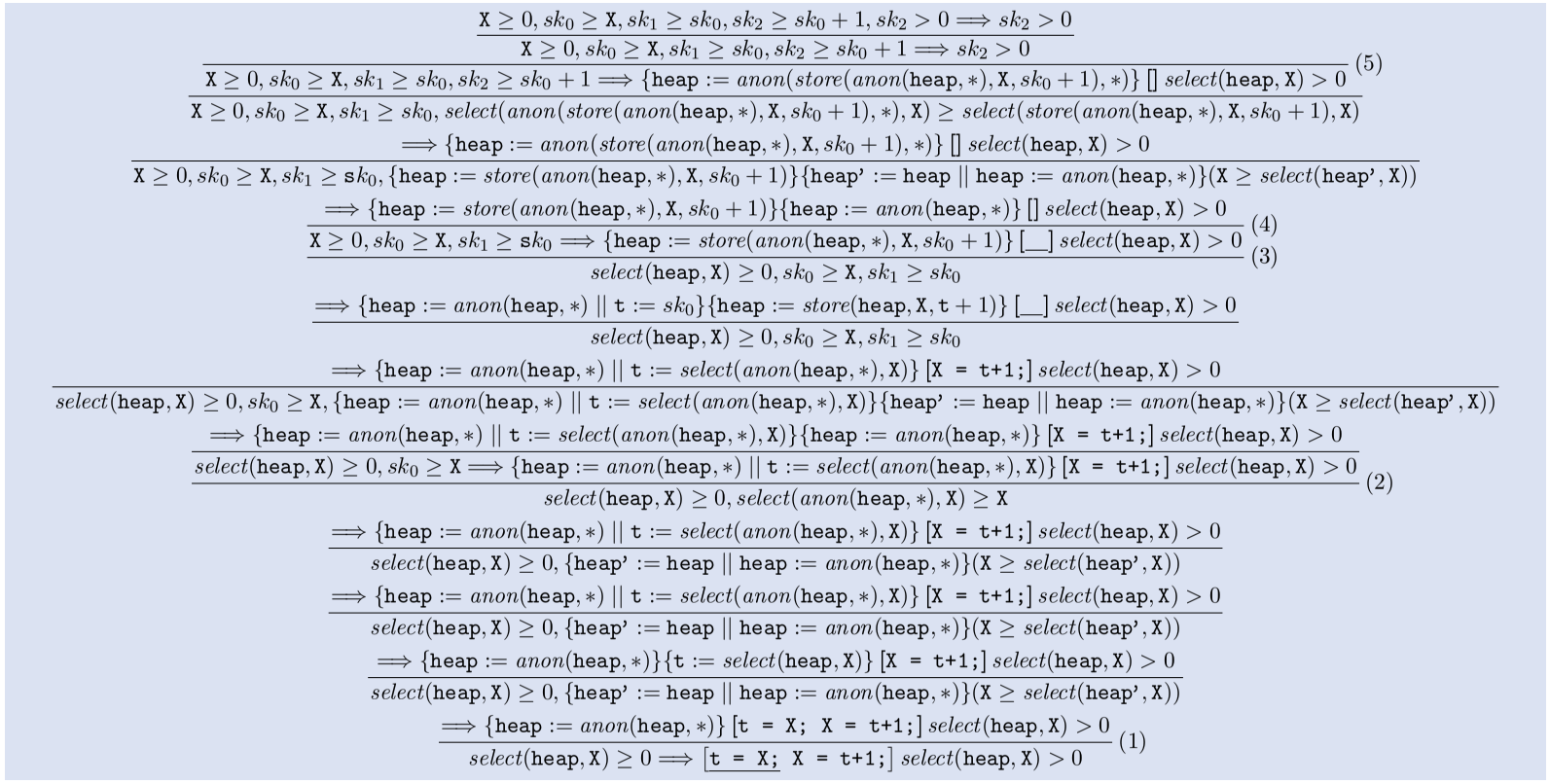


Figure 5.3: Proof tree for a postcondition provided by racy threads

statement through underlining. 3. We use the symbol $*$ as a shorthand for the constant location set expression $\dot{\varnothing}^{\complement}$ ('all locations').

In this example, the anonymization upon symbolically executing an (implicit) interleaving point affects the entire heap This first occurs in step (1). Each subsequent *select* on an anonymized heap results in a Skolem term sk_0, etc.; cf. step (2). All knowledge required to prove the postcondition is enclosed in the rely condition. In particular, the effect of the assignment to `X`, performed in step (3), is immediately 'lost.' The symbolic execution of the terminal `release` in step (4) again havocs the entire heap. Nevertheless, selecting the value of `X` on the *same* heap results in the same value, represented by sk_2 in the proof, as seen in step (5). Finally, the proof can be closed with the fact that—independent of their concrete values—the value of sk_2 is greater or equal than the value of sk_0 plus 1; and thus the value of sk_2 is always greater than zero.

5.6 Synchronization

So far, we have considered programs that work either on separated parts of the memory or involve (possibly malicious) data races. In both cases, multi-threading has not brought a benefit. To actually use multi-threading in a *sensible* way, threads need to work in concert. This can only be achieved through synchronization and specifications using ghost variables. We have not yet considered synchronization in this thesis because it would further complicate the presentation. In particular, our programing language dWRF would need to be extended (in a nonconservative way). A careful integration of synchronization primitives into our approach will be part of future work. In this outlook section, we sketch the fundamental ideas and demonstrate them using an example consisting of a program and its specification.

Mutual exclusion locks form the simplest kind of synchronization pattern. Locks can be acquired by one thread at a time and released again later (both without external interference). This is represented by `lock;` and `unlock;` statements. If the lock is already taken by another thread, the current thread waits for it to become available again. Locks (as well as other synchronization means) can not be modeled using the basic constructs defined in Sect. 3.2. For this reason, we introduce `lock;` and `unlock;` as additional statements with their intuitive semantics. Be aware that actions performed while holding a lock are not atomic: other threads may still write to variables. A working, race-free, implementation has to rely on all participating threads adhering to the protocol, i.e., not touching shared variables when they are not in their critical region.

The dark side of locking is the possibility of *deadlocks*. If two threads each hold a lock and wait for the other lock to become available, then progress is

impossible. While this may not be a considerable problem in practice, many of our definitions in the present and the preceding chapters rely on progress and would be subject to changes.

Ghost variables are important to specify concerted behavior of threads. Ghost variables are similar to regular variables, but exist only in specifications. Ghost variables are folklore in formal specification; see also Sect. 8.2.2 on their role in JML. They are meant to form a conservative extension to the original program [Filliâtre et al., 2014].

Stølen [1990, 1991]; Jones [2003] mention the particular importance of ghost variables in rely/guarantee specifications to track progress. Without them, the rely/guarantee approach is "hopelessly incomplete" [Stølen, 1990, Sect. 6.3]. In order to make use of ghost variables, we assume them to be *thread-local*, which allows them to be usable in specifications while being accessible (i.e., read or written) atomically. Atomicity is essential to describe progress without external interference.[9] In order to establish guarantees that mention both global and ghost variables, the write effect of ghost variables must occur simultaneously with the write effect of the global variable which it describes. This will become clearer through the example below.

Unfortunately, this atomicity assumption can not be proven within the system since it is not sound in general.[10] To cater for this different semantics of ghost variables, would require to change language semantics and the heap theory. We do not introduce this change formally, but believe that it is intuitively clear how it should be done.

A lock can be represented by an implicit ghost variable of integer type, where a positive value designates the thread holding the lock and any other value means the lock is available. To that end, each thread is assigned a numerical identifier greater than zero. This is an exception to the above premiss, where we postulated that ghost variables are thread-local. It should be obvious that atomicity of lock acquisition is necessary. This modeling through ghost variables is not sound in general, as we assume the variable `L` not to be written, even though it may be written as other threads acquire the lock. This setup suffices in this example if all participants adhere to the mutual exclusion scheme.

Example

We pick up the program example from Sect. 5.5.2. It considers two threads each trying to increase the value of a shared variable simultaneously, thus

[9] In Stølen's system, ghost variables are specified in rely conditions to not being changed, alike global variables. This is not necessary in our setup.

[10] Note that we make this assumption only here in the context of the rely/guarantee approach; traditionally, ghost variables do not have different semantics from regular variables.

competing in a data race. Now, we add synchronization to the picture. It allows the threads to read the shared variable `X` without interference before writing it. This way, an unbounded number of threads can work together.

We use one ghost variable $\texttt{C}_i$ per thread to account whether thread i has committed its update yet. There is only one lock, the lock holder is given through the implicit ghost variable `L`. We regard the program for thread i as follows:

π_i : `lock; release; t = X;` $\texttt{C}_i$ `= true; X = t+1; unlock; release;`

In contrast to the weak postcondition in Sect. 5.5.2, we now specify that the value of `X` has increased by the number of threads that have completed their operation. Here, ghost variables are essential to track which threads have finished yet. Note that the postcondition is weak: it does not prescribe that all threads have terminated. As discussed above in Sect. 5.4.4, this property cannot be proven with the rely/guarantee approach. We use the following specifications (where the sum comprehension operator can be defined as usual). Ghost variables are not included in the sets R_i and M_i as we assume them not to be changed as explained above.

$$
\begin{aligned}
\mathit{pre}_i &= \neg \mathit{select}(\texttt{heap}, \texttt{C}_i) \wedge \neg \mathit{select}(\texttt{heap}, \texttt{L}) \doteq i \\
\mathit{post}_i &= \mathit{select}(\texttt{heap}, \texttt{X}) \doteq \mathit{select}(h^{\text{pre}}, \texttt{X}) \\
&\qquad + \textstyle\sum_j \text{ite}(\mathit{select}(\texttt{heap}, \texttt{C}_j), 1, 0) \\
\mathit{rely}_i &= \mathit{post}_i \wedge (\mathit{select}(\texttt{heap}, \texttt{L}) \doteq i \rightarrow \mathit{select}(\texttt{heap}, \texttt{X}) \doteq \mathit{select}(\texttt{heap'}, \texttt{X})) \\
\mathit{guar}_i &= \mathit{select}(\texttt{heap}, \texttt{L}) \not\doteq i \rightarrow \\
&\qquad (\mathit{select}(\texttt{heap}, \texttt{X}) \doteq \mathit{select}(\texttt{heap'}, \texttt{X}) + 1 \wedge \mathit{select}(\texttt{heap}, \texttt{C}_i)) \\
R_i &= \dot{\varnothing} \\
M_i &= \{\texttt{X}\}
\end{aligned}
$$

The proof of the postcondition for thread k is displayed in Fig. 5.4 on the next page. We use the same simplifications to the presentation as in Sect. 5.5 above (with `release` indicated through underlining). In step (1), R37 is applied. It introduces an update including $\texttt{heap} := \mathit{anon}(\texttt{heap}, *)$. As we have explained above, in this section we do consider this not to havoc the complete heap, but ghost variables are implicitly not affected. Yet, the rely condition states that the value of `X` is still the same.

In step (2), the implication in the rely condition on the left-hand side is split. In the left branch, we have to show that k indeed holds the lock. In the right branch, we can use the equation for the rely condition, representing preservation of the value of `X`, and apply it to the update on the right-hand side. (The sequential update is simplified to a parallel update.)

The intermediate update to the heap, induced by the write to `X`, is destroyed again by the final `release`. However, it is sufficient that the rely condition implies the postcondition as shown in (3).

$$\dfrac{\dfrac{pre, \mathcal{U}_3 post \Longrightarrow \mathcal{U}_3 post}{pre, \mathcal{U}_3 rely \Longrightarrow \mathcal{U}_3 post}\ (3)}{pre \Longrightarrow \left\{ \begin{array}{l} h^{\text{pre}} := \texttt{heap} \mid\mid \texttt{heap'} := store(anon(store(\texttt{heap}, \texttt{L}, k), *), \texttt{C}_k,) \\ \mid\mid \texttt{heap} := store(store(store(anon(\texttt{heap}, *), \texttt{C}_k,), \texttt{X}, select(\texttt{heap}, \texttt{X}) + 1), \texttt{L}, 0) \end{array} \right\} [\_\_]\ post}$$

$$pre \Longrightarrow \left\{ \begin{array}{l} h^{\text{pre}} := \texttt{heap} \mid\mid \texttt{heap'} := store(anon(store(\texttt{heap}, \texttt{L}, k), *), \texttt{C}_k,) \\ \mid\mid \texttt{heap} := store(store(anon(store(\texttt{heap}, \texttt{L}, k), *), \texttt{C}_k,), \texttt{X}, select(\texttt{heap}, \texttt{X}) + 1) \end{array} \right\} [\texttt{unlock;}]\ post$$

$$pre \Longrightarrow \mathcal{U}_2 \{\texttt{heap} := store(\texttt{heap}, \texttt{C}_k,)\}\ [\texttt{X = t+1; unlock;}]\ post$$

$$pre \Longrightarrow \mathcal{U}_2\ [\texttt{C}_k\ \texttt{= true; X = t+1; unlock;}]\ post$$

$$\dfrac{\dfrac{\Longrightarrow k \doteq k}{\Longrightarrow \mathcal{U}_1 select(\texttt{heap}, \texttt{L}) \doteq k} \qquad pre, \mathcal{U}_1 (select(\texttt{heap}, \texttt{X}) \doteq select(\texttt{heap'}, \texttt{X})) \Longrightarrow \mathcal{U}_1 \{\texttt{t} := select(\texttt{heap}, \texttt{X})\}\ [\texttt{C}_k\ \texttt{= true; X = t+1; unlock;}]\ post}{pre, \mathcal{U}_1 rely \Longrightarrow \mathcal{U}_1 \{\texttt{t} := select(\texttt{heap}, \texttt{X})\}\ [\texttt{C}_k\ \texttt{= true; X = t+1; unlock;}]\ post}\ (2)$$

$$\dfrac{\dfrac{pre, \mathcal{U}_1 rely \Longrightarrow \mathcal{U}_1\ [\texttt{t = X; C}_k\ \texttt{= true; X = t+1; unlock;}]\ post}{pre \Longrightarrow \{h^{\text{pre}} := \texttt{heap} \mid\mid \texttt{heap} := store(\texttt{heap}, \texttt{L}, k)\}\ [\underline{\texttt{t = X;}}\ \texttt{C}_k\ \texttt{= true; X = t+1; unlock;}]\ post}\ (1)}{pre \Longrightarrow \{h^{\text{pre}} := \texttt{heap}\}\ [\pi]\ post}$$

The following abbreviations for updates are used:

$\mathcal{U}_1 = \{h^{\text{pre}} := \texttt{heap} \mid\mid \texttt{heap'} := store(\texttt{heap}, \texttt{L}, k) \mid\mid \texttt{heap} := anon(store(\texttt{heap}, \texttt{L}, k), *)\}$

$\mathcal{U}_2 = \{h^{\text{pre}} := \texttt{heap} \mid\mid \texttt{heap'} := store(\texttt{heap}, \texttt{L}, k) \mid\mid \texttt{heap} := anon(store(\texttt{heap}, \texttt{L}, k), *) \mid\mid \texttt{t} := select(\texttt{heap}, \texttt{X})\}$

$\mathcal{U}_3 = \{h^{\text{pre}} := \texttt{heap} \mid\mid \texttt{heap'} := store(store(store(anon(\texttt{heap}, *), \texttt{C}_k,), \texttt{X}, select(\texttt{heap}, \texttt{X}) + 1), \texttt{L}, 0) \mid\mid \texttt{heap} := anon(store(store(\texttt{heap}, \texttt{C}_k,), \texttt{L}, 0), *)\}$

Figure 5.4: Proof of postcondition for synchronized threads

6 Information Flow Analysis in Concurrent Programs

While analysis of secure information flow has been an active topic of research for some time, most approaches are deliberately incomplete for the sake of a speedy analysis (or, at least, automation itself). Incomplete analyses only overapproximate semantically defined information flow properties such as noninterference. This lack of precision not only leads to many innocuous programs being rejected, it also impedes analyses that consider subject declassification. Approaches based on theorem proving are able to cater for the necessary precision. They build on a faithful representation of the semantical property in logic and (interactive or auto-active) verification of the ensuing formulae. In previous work, these techniques have been brought to maturity, being applied to sequential Java and implemented in the KeY system. However, lifting the approach from sequential to concurrent programs is not trivial.

In this chapter, we introduce the concepts of secure information flow. Intuitively, security means that information may only flow to an information sink that has a security level higher or equal than its source. There are different concrete instantiations of security policies (cf. Chap. 2). We first review the traditional notion of *noninterference* for sequential programs. Since there are many variations to it, our definition is kept most general, in order to accomodate all of these variations. The general intuition behind noninterference is that low output must not depend on high input.

Similar to the functional verification approach in Chap. 5, we analyze concurrent programs in a modular way. Following this, we regard sequential programs, executed by a thread, in isolation. Concurrent write actions can be dealt with using the rely/guarantee approach. Regarding information flow, however, leakage may occur already during program execution. This requires us to additionally take read actions into account.

6.1 Introduction and Overview

Joshi and Leino [2000]; Amtoft and Banerjee [2004]; Barthe, D'Argenio, and Rezk [2004]; Darvas, Hähnle, and Sands [2005] introduced theorem proving approaches to language based information flow analysis. These are based on a semantical notion of information flow and therefore bear the advantage of semantical precision over established static techniques like type checking (see Sect. 2.4). Through this kind of formalization, confidentiality is reduced to a safety problem (cf. [Terauchi and Aiken, 2005]), that is checkable with off-the-shelf theorem provers. Some program logics such as dynamic logic are—unlike Hoare logic [Hoare, 1972]—readily able to express relational properties like information flow.[1] In particular, the KeY system is capable to formally verify information flow properties about *sequential* Java programs [Scheben and Schmitt, 2012a; Scheben, 2014]. The goal of this chapter is to present a first step on how these techniques can be lifted to reasoning about *concurrent* programs.

6.1.1 Information Flow in Concurrent Programs

Scheduling in concurrent programs may depend on unknown parts of the system state. Many models of concurrent program executions regard programs as indeterministic. For instance, Zdancewic and Myers [2003] postulate that reasoning about confidentiality in "concurrent languages is problematic because these languages are naturally nondeterministic; the order of execution of concurrent threads is not specified by the language semantics." We deliberately do not follow this paradigm (cf. Chap. 3). There is both a practical and a theoretical rationale behind this decision. Firstly, nondeterminism is just *a model* for unknown behavior. In the physical world, there is no such thing as nondeterminism (save for subatomic processes). Secondly, many definitions are much easier to give in terms of deterministic programs. E.g., the effect of a program should be the same if it is run twice from the same initial state under the same scheduler. This allows us to talk about exactly one computation trace, which greatly improves tractability of reasoning. We agree with Zdancewic and Myers about the execution order not being *entirely* determined by program semantics. But instead of nondeterminism, we prefer to view this as a kind of parametricism. We therefore model unknown behavior through *underspecification*.

As Cohen's original definition of noninterference only applies to deterministic sequential and terminating programs, several extensions have been proposed. Low-security observational determinism (LSOD) [McLean, 1992] is a well-known extension for concurrent programs, where schedulers are considered nondeterministic. It requires that each public output is computed

[1] Relational properties are sometimes known as "hyper-properties" [Clarkson and Schneider, 2010].

in an *observably* deterministic way, thus independent of scheduler indeterminism. However, LSOD both leaves the scheduler out of the picture and thus rejects programs that would be secure under specific (deterministic) schedulers, and at the same time, it does not report timing leaks that are induced by the relative order of memory updates, i.e., *internal* timing leaks.

The scenario that we investigate is concerned with a sequential program (i.e., thread) π in a deterministic language runs on shared memory with a possibly hostile environment. The control and information flow of thread π can be influenced by the environment. The attacker is able to observe low locations *at any time* and to observe the order of changes. This is different from the purely sequential setting—where the attacker can only observe initial and final values—and from distributed systems, where information does not flow through the memory, but only through declared channels. Attackers are not able to mount *external* timing attacks (i.e., attacks w.r.t. wall clock time or abstractions thereof). Thread-local information and control flows are expected to be instantaneous and are not observable.

We allow confidential information to be declassified. Since theorem proving approaches are founded semantically, precise subject declassification (i.e., *what* information is released) already comes for free. In this chapter, we additionally consider *timing* of declassification. Just like subject declassification can be expressed as a relational property between states, temporal declassification can be expressed as a relation between traces. In our logic, we formalize this through temporal operators. Controlling the temporal dimension of declassification is essential in state based software systems. Consider, for instance, an electronic voting system, that has different declassification policies before and after the election has been closed: only afterwards the result (i.e., the sum of votes) may be published.

6.1.2 Motivating Examples

To later evaluate our formal definitions against them, we present some small examples that can be classified intuitively into 'secure' or 'insecure' w.r.t. different capability classes of attackers. Consider the (uninstrumented) one-liner dWRF programs in Listing 6.1 on the following page. `L` is a low global variable of type $\mathbb{Z}$, `H` etc. are high global variables of type $\mathbb{B}$, `b` is a local variable of type $\mathbb{B}$, and `x`, `y` are local variables of type $\mathbb{Z}$. Let us so far assume that the above mentioned global variables are not written by concurrently running threads (if they were, we could not even consider the empty sequential program as secure, in general).

Programs 1 to 3 on the next page are both secure in purely sequential execution, as well as in a concurrent environment,[2] since, in any state, the

[2]Here, the assumption of no external interference is essential. An execution in which some environment thread writes the secret back after erase would not be considered secure.

```
1 { x = H; L = x * 0; }
2 { H = 0; x = H; L = x; }
3 { b = H==0; if (b) { x = L; y = H; L = x+y; } else {} }
4 { x = H; L = x; L = 23; }
5 { b = H0>0; if (b) { x = L; H1 = x; H2 = x; } else {} }
6 { while (b) { x = H; H = x-1; b = H>0; } }
7 { while (b) { x = L; L = x-1; b = H>0; } }
8 { b = H>0; if (b) {/* bubble sort */}
9             else {/* quick sort */} L = 1; }
```

Listing 6.1: Intuitively secure or insecure programs

value of `L` does not depend on the initial value of `H`. However, *proving* that these programs are secure can only be achieved with precise program semantics: In each case, there is a syntactical assignment for `H` to `L` present. They are rejected by most security type systems or PDG-based approaches (cf. Sect. 2.4). For 2, we have to consider that `H` has been erased and no secret *value* is leaked. In both 1 and 3, precise security analysis requires to consider the exact values and the semantics of addition and multiplication.

Program 4 is considered secure in the sequential setting, as the intermediate flow to `L` is erased in the final assignment. This is not true in a concurrent setting where an attacker may read low locations intermediately: There may be an interleaving between the two assignments in which the confidential information (temporally) stored in `L` is leaked to other threads.

Programs 5–8 contain control structures with high conditions. They possibly have different run time on low equivalent runs and thus expose *timing channels*. They are thus insecure if an attacker can compare traces component-wise. Programs 5 and 6 are secure if stuttering on the 'high' partition is tolerated. Their respective traces only differ in high component values; there is no assignment to `L`. In Program 7, the value of `L` obviously depends on high values; even with stuttering, it is insecure. Finally, if we would lift the restriction that `L` must not be modified concurrently, i.e., instrumenting the program with `release` statements, then neither program could be proven secure. The reason is that other threads could write secret information at any time, in particular at the end of these programs.

In the scope of this thesis, we only consider timing channels in so far that the number and order of global write events is an observable figure, i.e., *internal* timing channels. Program 8 exposes another kind of timing channel, an *external* timing channel. While there is only one global write event, the program branches on a high value and performs local computations of different complexity classes. A stronger attacker, that can observe run time in terms of computational complexity (or even in wall clock time) would be able to deduce the secret here. Table 6.2 on the facing page summarizes the different kinds of leaks in the examples of this section.

Table 6.2: Summary of information leaks in the examples in this section

program	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
direct leak (sequential)	✗	✗	✗	✗	✗	✗	✗	✗
direct leak (concurrent)	✗	✗	✗	✓	✗	✗	✗	✗
timing leak w/o stuttering	✗	✗	✗	✗	✓	✓	✓	✗
timing leak w/ stuttering	✗	✗	✗	✗	✗	✗	✓	✗
timing leak (complexity)	✗	✗	✗	✗	✗	✗	✗	✓

6.1.3 Approach

In this chapter, we will use CDTL as introduced in Chap. 4 to express information flow properties, in particular noninterference with declassification, for single threads of concurrent programs as explained above. We follow a precise logic-based semantic approach to information flow analysis along the lines of Scheben and Schmitt [2012a]. This allows to faithfully formalize the semantical security properties to be introduced in this chapter. We benefit from the power of our logic to express relational properties about programs readily. A formalization of noninterference for sequential Java programs in dynamic logic has already been presented by Scheben and Schmitt [2012a, 2014]; Scheben [2014]. We review the basic concept of noninterference in Sect. 6.2. It includes a formalization in Sect. 6.2.5. This presentation is simplified in contrast to the one by Scheben and Schmitt since we only cover the dWRF language.

As explained in Sect. 2.2, it suffices to prove this property for every pair of direct sub and super levels in the hierarchy. Without loss of generality, we will restrict the considerations made in this chapter to a two-element lattice with elements named ‘high’ and ‘low.’ This can be extended to any security lattice in a natural way.

Formalizing noninterference in a program logic essentially reduces the relational property to a safety property. Theorem provers can be employed to discharge the ensuing proof obligations, as Scheben and Schmitt pursue with KeY. Given a sound and complete calculus, this enables sound and complete analysis of direct and indirect flows as well as full semantical declassification. For purely sequential program, such a calculus has been presented in Sect. 4.4. For concurrent programs, we can use the rely/guarantee approach to restrict environment changes,[3] as explained in Chap. 5. The rely/guarantee approach allows to abstract away from concrete interleavings (w.r.t. a specific scheduler).

[3]In many situations, there may be even stronger guarantee conditions like perfect separation that could be checked with other methodologies, like type checking or runtime checking, that are less precise but more efficient.

This is proven to be sound. We are also convinced that the approach is complete relative to the interleaving semantics defined in Chap. 3.

In Sect. 6.3, we discuss how the basic notion of noninterference for sequential programs from Sect. 6.2 is transferable to multi-threaded programs. We develop a scheduler-independent notion of security in Sect. 6.3.1, which can be reduced to single threads in order to obtain a thread-modular security property (Sect. 6.3.2). The final Corollary 6.15 states that it is sufficient to prove noninterference for each sequential program (that is to be executed by some thread) in isolation.

In Sect. 6.4, we discuss even stronger security properties for concurrent programs. There are several possibilities to extend noninterference to a *trace-based* notion; we introduce a natural notion, based on pairwise equivalence, in Sect. 6.4.1, that can be formalized in an extension to CDTL as described in Sect. 6.4.2. We discuss how this notion of trace-based noninterference can be relaxed through equivalence up to stuttering (Sect. 6.4.3) or temporal declassification (Sect. 6.4.4). We compare our notion to others from the literature in Sect. 6.4.5.

Object-orientation and concurrency constitute largely independent dimensions of features of the Java language. For this reason, the considerations so far did not involve objects. Yet, objects play a central role in the Java language. In Sect. 6.5, we refine our notion of noninterference in a way such that it is appropriate to reason about information flows in Java programs. In contrast to the numerical pointers of C/C++, object references in Java are opaque. They only reveal identity of objects when directly compared, but not more information, like internal memory addresses. The presentation follows the structure of the work by Beckert, Bruns, Klebanov, Scheben, Schmitt, and Ulbrich [2014], leading to a notion of *object-sensitive* noninterference in Sect. 6.5.5. Section 6.5.6 explains how this property again can be formalized in dynamic logic.

We conclude this chapter with a discussion in Sect. 6.6 on how the results of this dissertation can be combined to obtain a usable and precise information flow analysis for concurrent Java. As mentioned above, there is an implementation of noninterference proof obligations for sequential Java programs in the KeY verification system. Furthermore, below in Chap. 7, we describe how verification of concurrent programs—based on rely/guarantee—can be added to KeY. Since KeY has been designed for the verification of (sequential) Java programs, there is relatively little overhead in lifting the techniques presented in this thesis from the dWRF language to Java. In principle, an extension to the calculus presented in this thesis should be realizable without much effort. We expect, however, that several optimizations will be necessary in order to efficiently prove information flow properties about nontrivial programs.

6.2 Noninterference

The most well known security policies for sequential programs is *noninterference* [Cohen, 1977; Goguen and Meseguer, 1982] (also known as absence of *strong dependency*). Intuitively, it means that "the value of public outputs does not depend on the value of secret inputs" [Barthe et al., 2004].[4] This means that a program that satisfies noninterference is secure against an attacker with unbounded deductive powers that may set low inputs and observe low outputs (and compare inputs with outputs; cf. Sect. 2.1). This includes information flows through both direct or indirect channels. The extension to termination sensitivity also includes termination channels. For a graphical rendition of the noninterference property, see Fig. 2.7 on page 24.

In language-based information flow security, sources and sinks are program constructs. Since our language introduced in Sect. 3.2 does not have method calls, the only program entities to which this applies are memory locations. A location may be both a source and a sink depending on the context, i.e., it is a source when it is read from and a sink when it is written.[5] For other programing languages, there may be other kinds of sources and sinks. For instance, Scheben and Schmitt [2012a] consider method parameters as sources in Java programs, as well as the method return value and the exception raised during execution as additional sinks.

6.2.1 Indistinguishability

To formalize noninterference, we first introduce the notion of *agreement* relations on states, or equivalently, indistinguishability to an adversary.

Definition 6.1. An agreement is a relation $R \subseteq \mathcal{D}_{\mathbb{L}} \times \mathcal{S}^2$ on a location set and two states, common written $R_{\mathcal{L}} \subseteq \mathcal{S}^2$ in infix style for a fixed $\mathcal{L} \in \mathcal{D}_{\mathbb{L}}$, that 1. is reflexive and symmetric (in $\mathcal{S}^2$), 2. (weakly) antimonotonic in $\mathcal{D}_{\mathbb{L}}$ (i.e., $\mathcal{L}_1 \subset \mathcal{L}_2$ implies $R_{\mathcal{L}_1} \supseteq R_{\mathcal{L}_2}$), and 3. has the property that $\mathtt{heap}^s = \mathtt{heap}^{s'}$ implies $(s, s') \in R$.

The final condition states that only the global memory, represented by the variable `heap`, has an influence of the relation. Typically, agreement relations are also equivalence relations. Since we only require weak monotonicity, in general, the requirement of item 2 is fulfilled by any constant. We denote

[4] Other prose definitions include: "One group of users [...] is noninterfering with another group of users if what the first group does [...] has no effect on what the second group of users can see." [Goguen and Meseguer, 1982]
"Observations of the initial and final values of k do not provide any information about the initial value of h." [Joshi and Leino, 2000]
"If two input states share the same low values, then the behaviors of the program executed on these states are indistinguishable by the attacker." [Sabelfeld and Myers, 2003a]

[5] Since this distinction does not matter in the following, we do not use the terms 'source' or 'sink,' but only speak of equivalences in the memory.

the universal agreement relation by $\divideontimes$. In Chap. 5, we already encountered the $\mathcal{L}$-equivalence relation $\approx_{\mathcal{L}}$, where $\mathcal{L}$ is a set of global variables. It relates states that agree on the $\mathcal{L}$ partition of the heap. While in Chap. 5, we use this relation to express that there is no change in a single run, in this chapter, we use it to express that the respective changes of two runs are equivalent.

Lemma 6.2. *The relation $\approx_{\mathcal{L}}$ from Def. 5.1 on page 93 is an agreement.*

Lemma 5.2 states that $\approx_{\mathcal{L}}$ is an equivalence relation; and according to Lemma 5.3, it is $\mathcal{L}$-antimonotonic. In particular, $\approx_{\emptyset}$ is the universal relation and $\approx_{\mathcal{D}_{\mathbb{L}}}$ is the empty relation. We will call $\approx_{\mathcal{L}}$ the *low-equivalence* relation when $\mathcal{L}$ denotes the set of low locations in a given security context. This relation can be considered very strict in some contexts. Other possibilities include the object-sensitive equivalence relation $\backsimeq^{\rho}$, which is introduced below in Sect. 6.5.

We say that two states s, s' are *R-insdistinguishable*, or just *indistinguishable* for short, if $(s, s') \in R$ for an agreement $R \subseteq \mathcal{S}^2$. It is important that the notion of 'observable' means that an attacker does not only know the anonymous *values* of variables, but knows their *evaluation*, i.e., the function from syntactic identifiers to values.

6.2.2 Basic Noninterference

We are now able to give a first definition of noninterference—for noninterleaved dWRF programs. It compares two possible runs of a program, starting from indistinguishable initial states (i.e., input), for indistinguishable terminal states (i.e., output). We give two variants of noninterference that especially consider nontermination.

We use the symbol $\overset{\pi}{\rightsquigarrow}$ for the big-step state transition relation, i.e., relating initial and terminal states, denoted by a noninterleaved sequential program π (cf. Sect. 3.4):

$$\overset{\pi}{\rightsquigarrow} := \{(s, s') \in \mathcal{S}^2 \mid \exists n \in \mathbb{N}.\ (n = |trc(s, \pi)| \wedge s' = trc(s, \pi)[n])\} \tag{6.1}$$

Note that this relation is actually a partial function, due to deterministic program semantics. It is empty in case of nontermination.[6]

Our definition of noninterference below is parametric in the sense that it can be instantiated with arbitrary agreement relations $\simeq_I$ (for *input*) and $\backsimeq_O$ (for *output*). This notion is very flexible and can be used to express many information flow security policies based on input/output, that otherwise would be considered extensions, e.g., declassification.

[6] We prefer a definition of a partial function over a total function, since the latter would force us to include a notion of a synthetic 'non-state,' that would have to be considered in all definitions.

Definition 6.3 (Noninterference)**.** Let two agreement relations $\simeq_I$ and $\backsimeq_O$, and two $\simeq_I$-indistinguishable states $s_1, s_1' \in \mathcal{S}$ (i.e., $s_1 \simeq_I s_1'$) be given. A program π is *termination-insensitive $\simeq_I/\backsimeq_O$-noninterferent* (TINI) if, started in s_1 or s_1' and there are respective terminal states s_2 and s_2', then these final states are $\backsimeq_O$-indistinguishable (i.e., $s_2 \backsimeq_O s_2'$):

$$\forall s_1, s_1', s_2, s_2'.\ (s_1 \simeq_I s_1' \wedge s_1 \overset{\pi}{\rightsquigarrow} s_2 \wedge s_1' \overset{\pi}{\rightsquigarrow} s_2' \rightarrow s_2 \backsimeq_O s_2')$$

A program π is *termination-sensitive $\simeq_I/\backsimeq_O$-noninterferent* (TSNI) if, starting π in s_1 or s_1', then it either does not terminate in both cases or it terminates in s_2 or s_2', respectively, and s_2, s_2' are $\backsimeq_O$-indistinguishable:

$$\begin{aligned}\forall s_1, s_1', s_2, s_2'.\ (\ \ & (s_1 \simeq_I s_1' \wedge s_1 \overset{\pi}{\rightsquigarrow} s_2 \wedge s_1' \overset{\pi}{\rightsquigarrow} s_2' \rightarrow s_2 \backsimeq_O s_2') \\ & \wedge (s_1 \simeq_I s_1' \rightarrow (\exists s_3.\ s_1 \overset{\pi}{\rightsquigarrow} s_3 \leftrightarrow \exists s_3'.\ s_1' \overset{\pi}{\rightsquigarrow} s_3')\ ))\end{aligned}$$

Please note that the above definition of noninterference is not a formula in our logic, but a meta-level property. We will present a formalization in Sect. 6.2.5 below.

Meta-properties of Noninterference

Both definitions capture the intuition that terminal states must agree on $\backsimeq_O$. The difference lies in the issue of termination, which is captured in the respective second line. In the formalization of TSNI, we additionally use the equivalent statement[7] that a run must terminate if and only if the other one does. TINI, on the other hand, allows either one (or both) to diverge. Thus TSNI is a strictly stronger policy than TINI (i.e., the set of programs satisfying it is a subset of programs satisfying TINI); see also the lemma below. Cohen's original definition of noninterference can be obtained from TINI through changing universal quantification over the terminal states by existential quantification.

Both versions are strictly monotonic in both input and output agreement: weakening the input agreement $\simeq_I$ or strengthening the output agreement $\backsimeq_O$ yields a stronger noninterference property. If the input agreement $\simeq_I$ is the same as the output agreement $\backsimeq_O$, we write $\simeq$-noninterference for short. This is the most common notion of noninterference, cf. [Sabelfeld and Myers, 2003a].

Remark. Another common pattern appears when a third security type is used, which means that variables of this type are neither considered high sources, nor low sinks. This 'don't care' type can be modeled through an output agreement that is weaker than the input agreement, i.e., initial states must aditionally agree on 'don't care' variables.

[7] This equivalence only holds for deterministic program semantics, i.e., where the relation $\overset{\pi}{\rightsquigarrow}$ is actually a (partial) function.

The following (folklore) lemma states that the question of termination (i.e., whether the termination channel is secure) can be decoupled from the remaining policy specification.

Lemma 6.4. *A program is $\simeq_I/\backsimeq_O$-TSNI if and only if it is $\simeq_I/\backsimeq_O$-TINI and $\simeq_I/\divideontimes$-TSNI (where $\divideontimes$ denotes the universal agreement relation).*

Noninterference is sequentially compositional. Our definition allows to declare the legal and illegal flows precisely. This allows composition without losing any precision. Scheben [2014] exploits this to define 'information flow contracts' for components such as loops or methods in Java.

Theorem 6.5 (Compositionality of noninterference)**.** *Let π_1 be a $\simeq_1/\simeq_2$-TINI program. Let π_2 be a $\simeq_2/\simeq_3$-TINI program. Then the sequential composition $\pi_1\pi_2$ is $\simeq_1/\simeq_3$-TINI. The analogous holds for TSNI.*

Proof. Ad TINI: Let $s_1, s_1' \in \mathcal{S}$ with $s_1 \simeq_1 s_1'$. If π_1 diverges from one of them, then $\pi_1\pi_2$ diverges and the claim is trivially fulfilled. Otherwise, the two runs of π_1 terminate in states s_2, s_2', respectively, with $s_2 \simeq_2 s_2'$, as assumed. Again, if π_2 diverges from either s_2 or s_2', then the claim is trivially true. Otherwise, from the assumption, both runs of π_2 terminate in states s_3, s_3', respectively, with $s_3 \simeq_3 s_3'$.

Ad TSNI: The case that the composed program $\pi_1\pi_2$ does terminate has been covered above. Assume π_1 diverges from s_1. According to the assumption, it diverges from s_1' as well. Hence, $\pi_1\pi_2$ diverges from both s_1 and s_1'. Now, assume π_1 terminates, for the compositum to diverge from s_1, this means that π_2 diverges from a state s_2, which is determined by the initial state s_1. Again, by the assumption, π_2 also diverges from s_2'. ◁

Noninterference vs. Stricter Policies

Noninterference is a *semantical* information flow policy. It is defined in terms of values of locations, not syntactical entities as assignments or control flow statements. The definition is independent of a particular analysis or enforcement technique. It means absence of both direct and implicit flows, but allows intermediate write actions to low locations that are erased at a later point.

Noninterferent programs may be considered insecure under stronger criteria. For instance the simple program `low = high * 0;` is obviously $\approx_{\{\texttt{low}\}}$-TSNI, but will be reported as insecure by most type systems[8] as they just check for the syntactical occurrence of the direct assignment, but not for the actual value. Another example is `if (high) low = p(); else low = p();`

[8]It should be noted that type systems *target* semantical information flow policies such as noninterference, but cannot *decide* whether a program is noninteferent due to their inherent incompleteness; cf. Sect. 2.4. For instance, Sabelfeld and Myers use the weaker terminology of "noninterference-style" type systems.

where we branch on a high variable, but the programs executed on both branches are the same. In principle, it does not even have to be the same computation, but only within the given agreement relation. Consider, for instance, a program that sorts an array of integers, but choses between quick sort and bubble sort depending on a high variable (cf. Listing 6.1).

6.2.3 Semantical Declassification

Declassification means the deliberate release of otherwise confidential information. We consider *semantical* declassification, i.e., we precisely describe the released information qualitative. Declassification can be expressed in our noninterference framework through a variation of the input agreement relation. Declassification policies describe security against attackers that cannot observe memory locations directly, but rather have some public view on the system. This formulation is more natural, as explained by Scheben and Schmitt [2012a].

Given a baseline input agreement relation $\simeq_I$ and a local-variable-free expression x to be declassified, the relation $\simeq_I^{\Delta x} := \{(s, s') \in \mathcal{S}^2 \mid s \simeq_I s' \wedge x^s = x^{s'}\}$ reduces the space of $\simeq_I$-equivalent input states to those that agree on the value of x. Strengthening the input agreement intuitively corresponds to increasing the knowledge of the attacker. This principle can be lifted to any finite set of expressions. We write $\simeq_I^{\Delta X}$ for a set of expressions X.

Lemma 6.6. *Let $\simeq_I$ be an agreement and X a set of expressions. The declassification relation $\simeq_I^{\Delta X}$ is again an agreement; it is an equivalence relation if $\simeq_I$ is an equivalence relation.*

The dual to declassification is called *erasure* (or 'killing'). Through erasing an otherwise revealed secret, the overall security is increased. It can be expressed through strengthening the output agreement relation in the same fashion.

Sabelfeld and Sands [2009] formulate some desirable meta-properties for declassification policies: 1. semantic consistency, 2. conservativity, 3. monotonicity of release, and 4. nonocclusion. A policy is *semantically consistent* if security is invariant under semantics-preserving program transformations. A policy is *conservative* if the empty set of declassification expressions corresponds to the original noninterference policy. A policy is *monotonic* if the set of secure programs is weakly monotonic w.r.t. the set of declassification expressions, i.e., declassification may only be a weakening to the baseline policy. A policy is *nonocclusive* if declassification does not mask other information leaks. Since our definition of declassification is semantical, both consistency and non-occlusion are given trivially. From the concrete

definition, it is also obvious to see that declassification is conservative (i.e., $\simeq_I^{\Delta\emptyset} = \simeq_I$) and monotonic (i.e., $\simeq_I^{\Delta X_0} \subseteq \simeq_I^{\Delta X_1}$ for $X_0 \supseteq X_1$).[9]

6.2.4 Conditional Noninterference

Another possible relaxation of noninterference is to include a functional precondition, that has to be established on both runs. This relaxed property is called *conditional noninterference* [Scheben and Schmitt, 2012a]. It helps to restrict the set of possible execution states. Typical cases are system invariants that are proven to hold by other means or by construction. This particularily includes system properties that are already established by the programing language design and need to be modeled in logic (i.e., *free* preconditions), e.g., the property that the heap is well-typed.[10]

Given a baseline input agreement relation $\simeq_I$ and a state formula *pre*, the relation $\simeq_I^{\Gamma pre} := \{(s, s') \in \mathcal{S}^2 \mid s \simeq_I s' \wedge s \vDash pre \wedge s' \vDash pre\}$ reduces the space of $\simeq_I$-equivalent input states to those that agree on *pre* to hold. As above, this can be combined into a relaxed definition of noninterference.

Lemma 6.7. *Let $\simeq_I$ be an agreement and pre a state formula. The relation $\simeq_I^{\Gamma pre}$ is again an agreement; it is an equivalence relation if $\simeq_I$ is an equivalence relation.*

6.2.5 Formalizing Noninterference

In this section, we explain how noninterferece (both TINI and TSNI) can be formalized in CDTL and its standard DL subset. Formalizations of noninterference in state-based dynamic logic have been first presented by Darvas et al. [2005]; Scheben and Schmitt [2012a] extend this formalization with dynamic classification. These formalizations use a 'box' modality on the left hand side of the implication, which means that nonterminating programs are always insecure. This is in line with Cohen's original definition, that considers program transition as a total function. Beckert, Bruns, Küsters, Scheben, Schmitt, and Truderung [2012b] present a revised version with a 'diamond' modality, which corresponds to TINI. For a formalization of TSNI, we add a simple formula that checks termination.

Related techniques to a direct formalization would be self-composition—that is based on program transformations—or the use of product programs, which would require a completely new calculus. The direct formalization technique is sometimes referred to as "self-composition of formulae" [Scheben, 2014]; nevertheless, we use the term 'self-composition' exclusively to refer to the program transformation technique by Barthe et al. [2004]. See Sect. 2.4 for a discussion of these techniques.

[9] Note the reversed set inclusion here. A larger declassification set means a stricter input agreement, which again is equivalent to a more relaxed noninterference policy.

[10] See [Grahl and Ulbrich, 2016] on free preconditions in JavaDL proof obligations.

Formalizing agreements

We first investigate whether and how agreements can be formalized.

Definition 6.8. An agreement relation $\sim$ is *formalizable* if there is a CDTL state formula φ containing two free logical variables h and h' such that for all states $s, s' \in \mathcal{S}$ and all variable assignments β with $\beta(h) = \mathtt{heap}^s$ and $\beta(h') = \mathtt{heap}^{s'}$, it is $s \sim s'$ if and only if $s^*, \beta \vDash \varphi$ is valid for any $s^* \in \mathcal{S}$.

The condition placed on the variable assignment β can be achieved by using a formula φ that explicitly updates the `heap` variable to one of the logical variables. This extends the above definition to any variable assignment and state s^*.

The limits of formalizability essentially are the usual limits of FOL expressivity.[11] Informalizable agreements could be defined on the meta level, but we consider them pathological. The simple low-equivalence relation $\approx_{\mathcal{L}}$ of Def. 5.1 is formalizable according to Lemma 5.6(2).

Lemma 6.9. *If $\sim$ is a formalizable agreement and X is a finite set of expressions, then the declassification/erasure agreement $\sim^{\Delta X}$ is also formalizable.*

Proof. Let φ be a formula formalizing $\sim$ with free variables h and h'. Let ψ be the formula $\varphi \wedge \bigwedge_{x \in X} (\{\mathtt{heap} := h\}x \doteq \{\mathtt{heap} := h'\}x)$. Since any x is free of local variables, it is $x^{s\{\mathtt{heap} \mapsto \mathtt{heap}^{s'}\}} = x^{s'}$. Hence, ψ formalizes $\sim^{\Delta X}$. ◁

Formalizing noninterference

The program transition relation $\overset{\pi}{\rightsquigarrow}$ (see (6.1) on page 124), that relates initial with terminal states, can be formalized in CDTL using the 'postcondition' pattern in a program modality: As mentioned in Sect. 4.1.3 on page 58, the temporal formula $\Diamond\bullet\mathit{false}$, containing a 'weak next' operator, expresses finiteness of a trace. Considering a sequential program π, it follows that the formula $[\![\pi]\!]\Diamond(\bullet\mathit{false} \wedge \varphi)$, including a nontemporal formula φ, is valid in a state s_1 if and only if there is a state s_2 with $s_2 \vDash \varphi$ and $s_1 \overset{\pi}{\rightsquigarrow} s_2$. Since this is a nontemporal formula, we use the standard DL modality $\langle\cdot\rangle$ as a shorthand: we write $\langle\pi\rangle\,\varphi$ instead of $[\![\pi]\!]\Diamond(\bullet\mathit{false} \wedge \varphi)$.

Lemma 6.10. *Let π be a noninterleaved sequential program, φ a formula, and $s_1 \in \mathcal{S}$. We have $s_1 \vDash [\![\pi]\!]\Diamond(\bullet\mathit{false} \wedge \varphi)$ if and only if there exists a $s_2 \in \mathcal{S}$ with $s_1 \overset{\pi}{\rightsquigarrow} s_2$ and $s_2 \vDash \varphi$.*

Proof. Let $s_1 \vDash [\![\pi]\!]\Diamond(\bullet\mathit{false} \wedge \varphi)$; as already mentioned, this is equivalent to $\mathit{trc}_\Sigma(s, \pi)$ being finite with the φ being valid in the final state. This is exactly the definition of $\overset{\pi}{\rightsquigarrow}$ in (6.1). ◁

[11] The expressivity of CDTL is the same as FOL with arithmetic, cf. [Beckert and Bruns, 2012b, Lemma 21].

As explained in Sect. 3.3, the program heap is a first-class citizen in our logic. Therefore, the formula $h \doteq \texttt{heap}$ with a free logical variable h can be used to 'remember' a heap state. This allows us to refer to the post-execution heap outside the scope of a program modality.

We are now able to formalize $\sim_I/\backsim_O$-TINI (assuming formalizable $\sim_I$ and $\backsim_O$) in (6.2) below. We use the shorthand expressions $h \mathrel{\dot{\sim}_I} h'$ and $h \mathrel{\dot{\backsim}_O} h'$ with heap expressions h and h' to denote formalizations of the semantical relations $\sim_I$ and $\backsim_O$. The formula is a simplified version of the formalization by Beckert et al. [2012b] and uses universal quantification over instances of the heap data type $\mathbb{H}$. Note that in this formalization, it is not necessary to formalize possible divergence. It is already captured in the semantics of the $\langle\cdot\rangle$ modality, that prescribes existence of a final state.

$$\begin{aligned}&\forall h_1, h_1', h_2, h_2' {:}\mathbb{H}.\ (h_1 \mathrel{\dot{\sim}_I} h_1' \\ &\land \{\texttt{heap} := h_1\}\langle\pi\rangle\, h_2 \doteq \texttt{heap} \land \{\texttt{heap} := h_1'\}\langle\pi\rangle\, h_2' \doteq \texttt{heap} \rightarrow h_2 \mathrel{\dot{\backsim}_O} h_2')\end{aligned} \tag{6.2}$$

The formalization of TSNI below in (6.3) results from applying Lemma 6.4. It includes an additional termination argument. It means that it does not depend on high input whether π terminates. Again, termination itself can be expressed with the $\langle\cdot\rangle$ modality alone, without the need for existential quantification. Overall, the formula includes 4 program modalities.

$$\begin{aligned}&\forall h_1, h_1', h_2, h_2' {:}\mathbb{H}.\ ((h_1 \mathrel{\dot{\sim}_I} h_1' \\ &\land \{\texttt{heap} := h_1\}\langle\pi\rangle\, h_2 \doteq \texttt{heap} \land \{\texttt{heap} := h_1'\}\langle\pi\rangle\, h_2' \doteq \texttt{heap} \rightarrow h_2 \mathrel{\dot{\backsim}_O} h_2') \\ &\land (h_1 \mathrel{\dot{\sim}_I} h_1' \rightarrow \{\texttt{heap} := h_1\}\langle\pi\rangle\, \mathit{true} \leftrightarrow \{\texttt{heap} := h_1\}\langle\pi\rangle\, \mathit{true}))\end{aligned} \tag{6.3}$$

Theorem 6.11. *Let $\sim_I$ and $\backsim_O$ be formalizable agreements. A sequential program π is $\sim_I/\backsim_O$-TINI if and only if Formula* (6.2) *is valid; it is $\sim_I/\backsim_O$-TSNI if and only if Formula* (6.3) *is valid.*

Proof. The present formulae are direct adaptations of the definitions of TINI and TSNI on the abstract level. The first part of the theorem follows directly from Def. 6.3 in combination with Lemma 6.10 and the formalizability of $\sim_I$ and $\backsim_O$. For the second part of the theorem, the goal is to prove that the additional conjuction in (6.3) is a formalization of $\sim_I/\divideontimes$-TSNI. This is actually a special case of Lemma 6.10 where $\varphi = \mathit{true}$. Finally, Lemma 6.4 provides that the overall formula corresponds to TSNI. ◁

In combination with the soundness (Corollary 4.18) and completeness (Thm. 4.19) results on the DTL calculus in Chap. 4, this theorem ensures that proving the above formalizations is a sound and precise information flow analysis:

Corollary 6.12. *Let $\sim_I$ and $\backsim_O$ be formalizable agreements. A sequential program π is $\sim_I/\backsim_O$-TINI if and only if Formula* (6.2) *is provable in $\mathcal{C}_{\mathrm{CDTL}}$; it is $\sim_I/\backsim_O$-TSNI if and only if Formula* (6.3) *is provable.*

6.3 Noninterference for Concurrent Programs

Cohen's original definition of noninterference only covers deterministic, sequential, and terminating programs. Depending on the exact formalization, the definition extends differently to indeterministic, parallel, or nonterminating programs. In this thesis, the program semantics is defined as deterministic (see Sect. 3.4). The case of indeterminism will not be considered; see the aside remark on the next page. The issue of termination has been covered above.

This leaves the question of how noninterference extends to concurrent programs. Recall the uninstrumented program 2 from Listing 6.1, `{ H = 0; x = H; L = x;}`, that first erases high data, and then assigns high to low. We have already discussed that it is secure in a purely sequential setting. As Sabelfeld and Myers [2003a] point out, it may be insecure in a multi-threaded setting if another thread assigns a secret to `H` in between the two assignments. Whether this program is secure depends on the functional behavior of the environment. And indeed, using our approach, we can prove that it is noninterferent if and only if it has a valid thread specification that provides that the environment does not change `H` to a value that depends on (other) high information.

Outline of This Section

In Sect. 6.3.1, we first investigate how the definition of noninterference from above can be applied to concurrent programs. In Sect. 6.3.2, we show how this property can be reduced to threads again. As a result, noninterference for concurrent programs can be analyzed using the technique introduced above in Sect. 6.2. In Sect. 6.3.3, we evaluate our definition against some program examples and discuss the relation to other extensions of noninterference to concurrent programs.

6.3.1 Adapting the Noninterference Definition

For a closer inspection of what noninterference means for concurrent programs, let us first recognize that the $\overset{\pi}{\rightsquigarrow}$ relation on sequential programs extends naturally to a concurrent system $(\mathcal{T}, \Sigma)$. We define a similar transition relation $\overset{\mathcal{T},\Sigma}{\rightsquigarrow}$, that denotes the transition to a state that is terminal for all threads.

$$\overset{\mathcal{T},\Sigma}{\rightsquigarrow} := \left\{ (s, s') \in \mathcal{S}^2 \;\middle|\; \exists n \in \mathbb{N}.\ (\sigma_\Sigma^n(s) = s' \wedge s' \in \Omega_{\mathtt{threads}^{s'}}) \right\} \tag{6.4}$$

Again, the relation is empty if there is at least one thread that does not terminate. Otherwise—since we are using deterministic program semantics and the transition from terminal states is the identity function—the relation $\overset{\mathcal{T},\Sigma}{\rightsquigarrow}$ is a function. In the special case that $\texttt{threads}^s$ only contains one element, this is equivalent to $\overset{\pi}{\rightsquigarrow}$. We redefine noninterference as of Def. 6.3 using the $\overset{\mathcal{T},\Sigma}{\rightsquigarrow}$ relation instead of $\overset{\pi}{\rightsquigarrow}$.

Remark. As many models for concurrent programs use indeterministic program semantics, it is worthwhile to investigate how our definition extends to indeterministic transition relations. So far, we have relied on the fact that the $\overset{\pi}{\rightsquigarrow}$ relation is a partial function. Extending to indeterministic programs, i.e., a more general relation, would only allow programs to be $\simeq_I/\backsimeq_O$-noninterferent w.r.t. a trivial output agreement relation $\backsimeq_O$. Our definition demands that all possible terminal states are within the same $\backsimeq_O$-equivalence class. See also [Sabelfeld and Myers, 2003a, Sect. IV B] for an overview over extensions of noninterference to different classes of indeterministic programs. See also our discussion on determinism in Sect. 3.6.

Definition 6.13 (Noninterference for concurrent programs)**.**
Let two agreement relations $\simeq_I$ and $\backsimeq_O$, and two $\simeq_I$-indistinguishable states $s_1, s_1' \in \mathcal{S}$ (i.e., $s_1 \simeq_I s_1'$) be given. A concurrent system $(\mathcal{T}, \Sigma)$ is *termination-insensitive $\simeq_I/\backsimeq_O$-noninterferent* (TINI) if, started in s_1 or s_1' and terminating in s_2 and s_2', respectively, then s_2 and s_2' are $\backsimeq_O$-indistinguishable:

$$\forall s_1, s_1', s_2, s_2'.\ (s_1 \simeq_I s_1' \wedge s_1 \overset{\mathcal{T},\Sigma}{\rightsquigarrow} s_2 \wedge s_1' \overset{\mathcal{T},\Sigma}{\rightsquigarrow} s_2' \rightarrow s_2 \backsimeq_O s_2')$$

A concurrent program Π is $\simeq_I/\backsimeq_O$-TINI if, for any fair scheduler Σ, the system $(\mathcal{T}, \Sigma)$ is $\simeq_I/\backsimeq_O$-TINI, where $\mathcal{T}$ is some set of threads for the program Π. We define *termination-sensitive $\simeq_I/\backsimeq_O$-noninterferent* (TSNI) systems and programs analogously.

Following our fundamental assumptions in Sect. 3.1.2, only fair schedulers are considered in this thesis.

In the attacker model that is implicitly behind the definition of concurrent noninterference, we assume that an attacker has knowledge about the scheduler and is able to exploit this. However, attackers are still passive and cannot control the scheduling (apart from providing low input). The attacker is *not* able to provide a scheduler on their own; such an attacker would be able to deduce just anything. This means that we always compare two program runs under the *same* scheduler. In our language-based approach, we assess the security of programs independently of the runtime environment. This particularly includes the scheduler. Thus, for a concurrent program to be secure it requires that it is secure when executed under *any* (fair) scheduler.

6.3.2 Reducing Noninterference to Threads

The above definition of noninterference for concurrent programs is not directly applicable to our approach, where we consider sequential programs w.r.t. some environment, not complete concurrent systems. In particular, our goal is to obtain a result that implies noninterference for *any* environment, i.e., independently of thread pool or scheduler. This result is provided by Thm. 6.14 below.

For a thread t and a state s, we use the shorthand $s \triangleright t$ that stands for $t \in \texttt{threads}^s$. We use the following notation to express the strengthening of an agreement $\simeq \in \mathcal{S}^2$ that requires that in both states a thread t is in the thread pool.

$$\simeq^{\triangleright t} := \{(s, s') \in \simeq \mid s \triangleright t \wedge s' \triangleright t\} \tag{6.5}$$

Obviously, the resulting relation $\simeq^{\triangleright t}$ is again an agreement (it is a special case of Lemma 6.7). Following Def. 3.12 on schedulers, $s \triangleright t$ is an implicit invariant of the trace of π_t for all states s in which t has been created. It holds in every state that is reachable through program steps. This means that the agreement in (6.5) must not be seen as a restriction in practice, but rather as a sanity assumption.

Theorem 6.14. *Let Π be a concurrent program; fix a concurrent system $(\mathcal{T}, \Sigma)$. If there is some $t \in \mathcal{T}$ for which the sequential program π_t is $\simeq_I^{\triangleright t}/\backsimeq_O$-TINI, then $(\mathcal{T}, \Sigma)$ is $\simeq_I^{\triangleright t}/\backsimeq_O$-TINI. We have $\simeq_I^{\triangleright t}/\backsimeq_O$-TSNI if additionally one of the following holds: 1. for every $t' \in \mathcal{T}$, $\pi_{t'}$ terminates for any initial state $s \triangleright t'$ or 2. there is a $t' \in \mathcal{T}$ such that $\pi_{t'}$ never terminates.*

Proof. Let us consider two states $s \simeq_I s'$. Let $t \in \texttt{threads}^s \cap \texttt{threads}^{s'}$ be a thread with program π_t that is $\simeq_I^{\triangleright t}/\backsimeq_O$-TINI. We have to distinguish three cases:

(i) If there is one thread that never terminates, then so does the concurrent program and it is vacuously TSNI.

(ii) If all threads always terminate, then Lemma 3.22 states that all threads have the same final state, i.e., $\overset{\mathcal{T},\Sigma}{\rightsquigarrow} \subseteq \overset{\pi_{t'}}{\rightsquigarrow}$ for any thread t'. This means that in this case the original definition of noninterference for sequential programs (Def. 6.3) is equivalent to the concurrent extension. Since π_t terminates, it is TSNI by Lemma 6.4. It follows that the concurrent program is TSNI.

(iii) In any other case—by the same argument as in case (ii)—the concurrent program is TINI (but not TSNI). ◁

This result greatly reduces the effort to prove noninterference for multi-threaded programs. In restricting the input relation to states that agree on a thread t being in the thread pool (i.e., using the agreement in (6.5)), it suffices

to prove TINI for t alone. Moreover, the proof obligations are independent of the scheduler Σ. In combination with the rely/guarantee approach of Chap. 5, it also offers to prove noninterference in a thread-modular way. A sufficiently strong rely condition allows us to prove noninterference for any environment.

However, the result is still restricted to states agreeing on the thread pool. In a completely modular setting, we do not know of the value of `threads`. To get rid of the restriction, we have to prove noninterference for *all* threads that may be in the thread pool, i.e., all threads that correspond to the concurrent program Π. Recall that $\mathcal{T}$ consists of copies of the sequential programs in Π that are equal up to renaming of local variables. This means that we only need to prove noninterference for each $\pi \in \Pi$, everything else is given through symmetry. For TSNI, additional termination arguments are required. This leads us to the following corollary.

Corollary 6.15. *Let Π be a concurrent program. Let all $\pi \in \Pi$ be $\simeq_I/\backsimeq_O$-TINI. Then Π is $\simeq_I/\backsimeq_O$-TINI. It is $\simeq_I/\backsimeq_O$-TSNI if additionally one the following holds: 1. all $\pi \in \Pi$ terminate or 2. there is a $\pi' \in \Pi$ that never terminates.*

With this result in hand, we can conclude that proving the formalization of (6.2) suffices to show noninterference in concurrent programs. While the condition for TINI seems to be obvious, the additional condition for TSNI is sufficient, but probably not necessary. Further investigation will be left to future work.

Compositionality

Note that Corollary 6.15 is not a result on parallel compositionality of noninterference. We are often interested in such properties as: if the sequential programs π and π' are noninterferent w.r.t. thread-local agreements, then the concurrent program consisting of threads executing π and π' is noninterferent again (w.r.t. some combined agreement). This is not the case with Def. 6.13. Consider the following uninstrumeted programs:

π : `{ H = 0; x = H; L = x; }`
π' : `{ y = H; H = y; }`

Both sequential programs π and π' are noninterferent (w.r.t. an empty environment): π erases high data before writing to a low sink (cf. Listing 6.1), while in π' high data is written to a high sink. However, a combined system with both programs being executed concurrently is insecure. There exists a schedule in which the erasure on `H` in π is overwritten by the assignment in π' and thus the final assignment of π leaks the initial value of `H` to `L`. Such compositionality results are hard to obtain and are outside the scope of this thesis. In particular, we do not have a notion of 'local' agreements, but every thread must conform to the system-wide policy.

6.3.3 Discussion

We have presented a natural extension of the definition of noninterference to concurrent programs. Our notion of security includes indistinguishability of final execution states under the same deterministic scheduler. A program is secure if the output is indinstinguishable for *any* scheduler. Our security property is strong enough to reject programs with probabilistic leaks—even though probabilities are not modeled. In this regard, our property can be compared to low-security observational determinism (LSOD) [McLean, 1992]. However, it is not comparable to LSOD in other regards, since we consider terminal system states, as opposed to traces of variable values in LSOD.[12]

Program Examples

We review some well-known program examples from the literature and compare the security assessment by the respective authors to our own. All examples have been adapted to (approximately) match the syntax and semantics of dWRF.

Example 6.16. Zdancewic and Myers [2003, p. 3] motivate their definition of LSOD with the following program example.

```
fork { L = 0; release; }
fork { L = 1; release; }
fork { release; x = H; L = x; release; }
```

Zdancewic and Myers reject this program because there exists an interleaving such that the value of `H` is directly leaked to `L`. For the same reason, our property classifies this program as insecure, too. Earlier, weaker information flow properties, such as GNI [McCullough, 1988], would classify it secure, because there exists an interleaving such that the value of `H` is *not* leaked.

Example 6.17. The following concurrent program appeared in the work by Terauchi [2008]. We assume that there is no external interference to `L` (i.e., from outside the two threads that run the program below).

```
fork { L = 0; release; } L = 1; release;
```

According to our definition, it is not secure, even though no high variable is syntactically involved. The reason is that we have to consider all possible interleavings between the two threads (i.e., the one considered and the one forked by it). A scheduler may encode secrets in the schedule, thus leading to different results. Depending on the particular scheduler and the unknown part of the heap (including secrets) on which the scheduler depends, the final value of `L` is either 0 or 1.

[12] In Sect. 6.4 below, we introduce a trace-based notion of noninterference ourselves. We compare the role of program traces with LSOD once more in Sect. 6.4.5.

In general, we cannot exclude the possibility that a schedule depends on secrets, thus we have to consider this program as insecure. The same argument applies to the example by Snelting [2015, Fig. 4 middle], which we consider insecure as well. Some researchers, such as Snelting, argue that this is a kind of benign indeterminism and therefore should be considered secure. However, it remains unclear how to enforce a schedule that does not depend on secrets. On the other hand, from a practical viewpoint, the difference is small. It does only appear with programs that contain data races, which could be avoided altogether.

Example 6.18. The example by Snelting [2015, Fig. 3 right], that is displayed below, is secure according to our definition. This is because, for any possible interleaving, the terminal state is always the same. The relative order of updates to the two variables may be different, though.

```
fork { X = 0; release; } fork { Y = 1; release; }
```

Example 6.19. Considering termination channels, the following example by Huisman et al. [2006, p. 4] is rejected by their security property. However, it is TSNI according to our definition, since—under any scheduler—termination does not depend on high values (i.e., it never terminates).

```
fork { L = 7; }
while (true) { skip; }
```

6.4 Trace-based Notions of Noninterference

If programs are themselves concurrent, "the idea that a program terminates with a single result is less appropriate for a concurrent language, where programs may produce observable effects while continuing to run." [Zdancewic and Myers, 2003] Moreover, noninterference for sequential programs centers around terminating programs—the original definition by Cohen [1977] did only consider those—while concurrent programs are often designed to run forever. This would render the notion of noninterference w.r.t. input/output effectively vacuous. These issues motivate us to revise our attacker model. Above, we assumed an attacker that can set the public part of the initial memory, read the public part of the *final* memory, compare them, and produce arbitrary deductions from the result.

In the following, we assume that a more powerful attacker, can read the public memory at any time—and again, that they can deduce any corollary from that knowledge. To be more precise, by the colloquial 'at any time,' we understand all intermediate program states that are reached throughout execution. This means than an attacker can observe and compare complete program traces. This observation includes the entire low-observable part of each state, but also state changes in which the low-observable part does not

change; i.e., an attacker can observe that changes to the 'high' part occur, yet they cannot observe the change itself. This kind of attacker is able to listen to internal *timing* channels. For instance, they can distinguish traces in which write actions are permuted. Obviously, this attacker is strictly more capable that the one we assumed above in Sect. 6.2. In Sect. 6.4.3, we further refine this attacker model by restricting observations strictly to the 'low' part, i.e., that refined attacker cannot distinguish traces that are equivalent up to stuttering on the 'low' part.

6.4.1 Strong Noninterference

The above noninterference properties from Sect. 6.2 can be lifted to a property on program traces. The use of trace-based semantics for security properties was already suggested by McCullough in 1987. Similar to the sequential setting, we first define indistinguishability of traces. We provide two definitions, one based on strong equivalence requiring that two traces run simultaneously, the other based on weak equivalence allowing stuttering on the low security projection of the state (Sect. 6.4.3). By abuse of notation, we use the same symbols for agreement of states and the resulting notion of equivalence of traces. The notion of *strong noninterference* of traces will again be parametric in the input relation and output relation.

Definition 6.20. Let $\backsim$ be an agreement relation. Program traces τ and τ' are *strongly $\backsim$-equivalent*, again denoted $\tau \backsim \tau'$, if they are of same length and $\backsim$-equivalent on every position:

$$\tau \backsim \tau' :\Leftrightarrow |\tau| = |\tau'| \wedge \forall i \in [0, |\tau|).\ \tau[i] \backsim \tau'[i]$$

We do not strictly require the agreement $\backsim$ to be an equivalence relation, even though we expect this to be the typical case. Therefore, we will call the relation on traces 'equivalence' irregardless whether it is actually an equivalence relation. Note that strong equivalence already includes termination sensitivity.

Definition 6.21 (Strong Noninterference)**.** Let $\sim_I$ and $\backsim_O$ be agreement relations. 1. A sequential program π is *strongly $\sim_I/\backsim_O$-noninterferent* if for any fair scheduler Σ,

$$\forall s, s' \in \mathcal{S}.\quad s \sim_I s' \Rightarrow trc_\Sigma(s, \pi) \backsim_O trc_\Sigma(s', \pi) \qquad .$$

2. A concurrent system $(\mathcal{T}, \Sigma)$ is *strongly $\sim_I/\backsim_O$-noninterferent* if

$$\forall s, s' \in \mathcal{S}.\quad s \sim_I s' \Rightarrow trc_\Sigma(s) \backsim_O trc_\Sigma(s') \qquad .$$

Here, the function trc_Σ is the system trace from Def 3.20. 3. A concurrent program Π is *strongly $\sim_I/\backsim_O$-noninterferent* if all concurrent systems $(\mathcal{T}, \Sigma)$ for Π are strongly $\sim_I/\backsim_O$-noninterferent.

This definition is similar to the well known notion of low-security observational determinism (LSOD) [McLean, 1992] for completely indeterministic systems, in the sense that programs are only considered secure if they produce indistinguishable outputs for any run. We have already discussed this relationship in the context of input/output noninterference above in Sect. 6.3.3. For a comparison with different variants of LSOD regarding the role of the trace in the definitions, see Sect. 6.4.5 below.

By analogy to simple noninterference for concurrent programs, strong noninterference can be reduced to single threads (and thus sequential programs) as in Sect. 6.3.2. The proof for the following lemma follows by analogy from Thm. 6.14 and Corollary 6.15.

Lemma 6.22. *Let* Π *be a concurrent program. Let all* $\pi \in \Pi$ *be strongly* $\simeq_I/\eqsim_O$*-noninterferent. Then* Π *is strongly* $\simeq_I/\eqsim_O$*-noninterferent.*

6.4.2 A Formalization of Strong Noninterference

Strong noninterference can be formalized directly using a conservative extension to CDTL. This extension consists of a 'n-th next' state operator, where n is an expression of type $\mathbb{Z}$. It allows us to count states in a trace, which is not expressible using LTL-like operators. This is necessary since, in strong noninterference, we compare two traces induced by different program runs. Since CDTL already extends FOL, this extension is harmless. In particular, reasoning about the 'n-th next' operator can be delegated to the induction rule R33. This approach is less invasive than the other possibilities to compare runs, that use either program instrumentation following the selfcomposition approach [Barthe et al., 2004] or dedicated *product program* logics [Barthe et al., 2011; Scheben and Schmitt, 2012b]. A preliminary version of this formalization appeared in [Bruns, 2014b], which did not use finite sequences as the 'storage' data type, but *maps* (i.e., a data type representing partial functions, cf. [Wallisch, 2014]). We believe that reasoning about finite sequences is more tractable and that the sequence theory calculus by Beckert et al. [2013b, Appendix A] is sufficiently complete.

Thanks to the expressiveness of CDTL, *simple* noninterference (input/output determinism) can be formalized directly as demonstrated in Sect. 6.2.5. In our formalization using explicit heaps, this means that heaps must equal on the partition induced by a set $\mathcal{L}$ of low locations. There, it is sufficient to compare the respective heaps in the poststates of each run (if they exist).

For strong noninterference, a crucial point is that we need to pairwise compare an unbounded (possibly infinite) number of heaps. A first step towards this goal is the fact that it is sufficient to regard finite prefixes, provided by the well-known principle of natural induction. Note that this prefixing has no effect on finite traces.

Lemma 6.23. *Let τ, τ' be traces of infinite length. τ and τ' are strongly equivalent if and only if all finite subtraces of are pairwise equivalent:*

$$\tau \backsim \tau' \Leftrightarrow \forall k \in \mathbb{N}_{>0}.\ \tau[0,k) \backsim \tau'[0,k)$$

For a pairwise comparison of finite prefixes, we need to 'record' the heap state of the trace through a temporal formula. Although this number of heaps is finite, it is still unbounded. We use the data type of finite sequence (see Sect. 8.2.2; [Beckert et al., 2013b, Appendix A]) to store these heaps.

Extending CDTL With a Counting Operator

We add a unary temporal 'n-th next' operator $\bullet^n$ to CDTL to count the number of states in a finite prefix of a trace. Intuitively, $\bullet^n$ corresponds to n consecutive 'weak next' operators. However, it is not syntactical sugar since the value of n is not static. For simplicity, we require that n is a rigid term, i.e., it does not contain any program variables. Formally, we define its semantics through an extension of the validity relation $\vDash$ (Def. 4.7):

$$\tau, \beta, \Sigma \vDash \bullet^n \varphi \qquad \text{iff} \qquad \tau[n^{\tau[0],\beta}, \infty), \beta, \Sigma \vDash \varphi \text{ or } |\tau| \leq n^{\tau[0],\beta} \tag{6.6}$$

It is easy to see that this definition generalizes the definition of validity for the 'weak next' operator $\bullet$, which can be seen as syntactic sugar for $\bullet^1$. Since n is a rigid term, we may omit the state of evaluation and instead write n^β for its value. The addition of this operator increases the expressivity of the temporal logic part of CDTL. We did not include it in the logic directly because the set of temporal operators that are inspired by LTL is standard and intuitive. The operator does *not* increase the expressivity of the overall logic since that already extends FOL with arithmetic and all temporal operators could be modeled using the natural numbers.

To reason about the extended logic, we provide a rule for temporal unwinding the $\bullet^n$ operator, as shown below. Since the rule focusses only on the trace formula and does not depend on the program π, it does not affect any of the results of Chap. 5. Since they should be obvious, we do not provide formal proofs of soundness and completeness.

$$\frac{\Gamma, n > 0 \Longrightarrow \mathcal{U}[\![\pi]\!]\bullet\bullet^{n-1}\varphi, \Delta \qquad \Gamma, n \leq 0 \Longrightarrow \mathcal{U}[\![\pi]\!]\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\bullet^n\varphi, \Delta}\ \text{R39}$$

The following temporal formula η containing a free logical variable *heaps* of type $\mathbb{S}$ captures the property that *heaps* contains all the heaps of a trace on which it is valid.

$$\eta(heaps) := \qquad \forall n{:}\mathbb{Z}.\ (0 \leq n < |heaps| \rightarrow \bullet^n heaps[n] \doteq \texttt{heap}) \tag{6.7}$$

Lemma 6.24. *Let τ be a trace. Let $\eta(heaps)$ be the formula (6.7). Further let $\ell \in \mathbb{N}$ be the minimum of $|heaps|^{\tau[0]}$ and $|\tau|$.*[13] *Then it holds that $\tau \vDash \eta(heaps)$ is valid if and only if $heaps^{\tau[0]}[0, \ell) = \langle \mathtt{heap}^{\tau[0]}, \mathtt{heap}^{\tau[1]}, \ldots, \mathtt{heap}^{\tau[\ell-1]} \rangle$.*

Proof. We show one implication direction. Assume that $\tau \vDash \eta$ is valid. That is forall $z \in \left[0, |heaps|^{\tau[0]}\right)$, $\tau \vDash \bullet^n heaps[n] \doteq \mathtt{heap}$ where $n^{\tau[0]} = z$. The semantics of the $\bullet^n$ operator gives us that $\tau[z, \infty) \vDash heaps[n] \doteq \mathtt{heap}$ or $|\tau| \leq z$. The subtrace $\tau[z, \infty)$ can be replaced by the state $\tau[z]$ due to the lack of temporal operators in the formula. This disjunction is then equivalent to the combined $\tau[z] \vDash heaps[n] \doteq \mathtt{heap}$ for all z in the restricted range $[0, \ell)$ with $\ell = \min\left(|\tau|, |heaps|^{\tau[0]}\right)$. This means $heaps^{\tau[z]}[z] = \mathtt{heap}^{\tau[z]}$ and, since *heaps* is rigid, $heaps^{\tau[0]}[z] = \mathtt{heap}^{\tau[z]}$ for all $z \in [0, \ell)$, which is what we wanted to prove. The opposite implication direction follows in the same spirit. ◁

Formalizing Strong Noninterference

We are now ready to formalize strong noninterference. Let π be a sequential program and $\sim_I$ and $\backsim_O$ formalizable agreement relations. Let η be the formula from (6.7). We use the shorthand expressions $h \mathrel{\dot{\sim}_I} h'$ and $h \mathrel{\dot{\backsim}_O} h'$ with heap expressions h and h' to denote formalizations of the semantical input and output agreement relations $\sim_I$ and $\backsim_O$ similar to how we did in (6.2) on page 130. If the formula below (6.8) is valid, then π is strongly $\sim_I/\backsim_O$-noninterferent.

$$
\begin{aligned}
\forall h, h^*{:}\mathbb{H}.\ \forall heaps, heaps^*{:}\mathbb{S}.\ (&\\
&|heaps| \doteq |heaps^*| \quad \wedge \quad h \mathrel{\dot{\sim}_I} h^* \\
&\quad \wedge \{\mathtt{heap} := h\ \}[\![\pi]\!]\, \eta(heaps) \\
&\quad \wedge \{\mathtt{heap} := h^*\}[\![\pi]\!]\, \eta(heaps^*) \\
&\rightarrow \forall i : \mathbb{Z}.(0 \leq i < |heaps| \rightarrow heaps[i] \mathrel{\dot{\backsim}_O} heaps^*[i]))
\end{aligned}
\tag{6.8}
$$

The formalization follows the same basic pattern as for simple noninterference in (6.2). Given initial heaps that agree through $\dot{\sim}_I$, we 'record' the resulting heap traces in variables *heaps* and $heaps^*$ as explained above in Lemma 6.24. These are then compared pairwise to each other through the output agreement $\dot{\backsim}_O$ (on the right hand side of the implication). The sequence data type $\mathbb{S}$ represents only *finite* sequences; the natural induction principle is enshrined in the universal quantification over all sequences that record the trace (up to their common length). This ensures that we consider every finite prefix as stated in Lemma 6.23. In the corner case that each run

[13] The length of the trace may be infinite, but the length of a sequence is always a finite number.

has a different runtime, the formula is not valid because of this universal quantification. A formal proof of this formalization being sound and complete is left to future work.

6.4.3 Stutter Tolerant Noninterference

The above definition of strong noninterference gives a security condition against an attacker that can observe any intermediate state. This is a rather strong condition in that purely high computations can be insecure. Stuttering is important since it is more realistic that the attacker can only observe (low) *changes* to locations instead of states of the trace. Traces that are equivalent up to stuttering may include subsequent states that are low-equivalent. We generalize the above relation $\backsim$ to a *equivalence up to stuttering* relation $\backsim^\zeta$, that permits low-inobservable stuttering. This property provides security against an attacker that cannot distinguish succeeding $\backsim$-equivalent states. We define $\backsim^\zeta$ recursively for a pair of finite traces and through the least fixpoint if traces are possibly of infinite length.

Definition 6.25. Let $\backsim \subseteq \mathcal{S}^2$ be an agreement. Two traces τ and τ' are $\backsim$-equivalent up to stuttering, written as $\tau \backsim^\zeta \tau'$ if

- $|\tau|, |\tau'| < \infty$ and $\tau[0] \backsim \tau'[0]$ and
  - $\tau[1, \infty) \backsim^\zeta \tau'[1, \infty)$ (if $|\tau|, |\tau'| > 1$),
  - $\tau[1, \infty) \backsim^\zeta \tau'$ (if $|\tau| > 1$), or
  - $\tau \backsim^\zeta \tau'[1, \infty)$ (if $|\tau'| > 1$);
- **or** for all $k \in \mathbb{N}_{>0}$: $\tau[0, k) \backsim^\zeta \tau'[0, k)$.

It is easy to see that $\backsim^\zeta$ is again an equivalence relation if $\backsim$ is an equivalence relation. Note that this relation is not termination-sensitive: for an infinite trace τ and a finite trace τ', $\tau \backsim^\zeta \tau'$ means that τ has a tail consisting of only the final state of τ'. We define a variant $\backsim^{\zeta_\perp}$ that includes termination as the relation $\{(\tau, \tau') \in \backsim^\zeta \mid |\tau| = \infty \Leftrightarrow |\tau'| = \infty\}$.

Example 6.26. An alternative, graph-theoretical, view is that a transition relation and an agreement form strongly connected components. In general, this is a many to many relation between both traces. If we now contract each trace such that the strongly connected components merge into a single node, then these contracted traces are strongly equivalent if and only if the original traces are equivalent up to stuttering. Figure 6.3 on the next page depicts an example of two traces τ and τ' that are not strictly low-equivalent (as in Def. 6.20), but low equivalent up to stuttering.

Definition 6.27. Strong $\sim_I / \backsim_O^\zeta$-noninterference is called *termination-insensitive semi-strong* $\sim_I / \backsim_O$*-noninterference* (TIS^{2}NI). Strong $\sim_I / \backsim_O^{\zeta_\perp}$-noninterference is called *termination-sensitive semi-strong* $\sim_I / \backsim_O$*-noninterference* (TS^{3}NI).

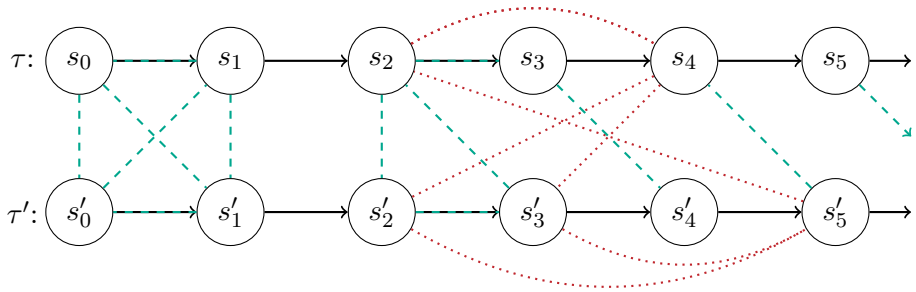


Figure 6.3: Graph-theoretical representation of low-equivalent traces up to stuttering. States connected with dashed green or dotted red edges are in a common equivalence class w.r.t. $\approx_L$. There is a stuttering step from s'_2 to s'_3, while s'_3 is not equivalent to s_3. The dashed green edges constitute strongly connected components of the intersection of the $\approx_L$ relation with the successor relation in traces (solid black edges).

Lemma 6.28. *Let $\backsim \subseteq \mathcal{S}^2$ be an agreement. Let τ, τ' be finite traces with respective lengths ℓ and ℓ'. If $\tau \backsim^\zeta \tau'$, then $\tau[\ell - 1] \backsim \tau'[\ell' - 1]$.*

Proof. This follows by induction over ℓ and ℓ' from Def. 6.25. ◁

Our different notions of noninterference can be organized in a hierarchy (cf. Fig. 6.6 on page 147):

Theorem 6.29 (Hierarchy of security policies)**.** *The following strict inclusions hold:*

1. *Programs that have the strong noninterference property also have the TS³NI property, but not vice versa.*
2. *Programs that have the TS³NI property also have the TSNI property, but not vice versa.*
3. *Programs that have the TIS²NI property also have the TINI property, but not vice versa.*
4. *Programs that have the TIS²NI property also have the TIS²NI property, but not vice versa.*
5. *Programs that have the TSNI property also have the TINI property, but not vice versa.*

Proof. The security policies mentioned here do not differ in the way that input agreements are involved in them. The differences lie only in termination and output agreements/trace properties. Ad 1: Let an agreement $\backsim_O$ be given. If $\tau \backsim_O \tau'$ holds, it obviously fulfills the first condition in Def. 6.25 Thus we have $\backsim_O \subseteq \backsim_O^\zeta$. For the opposite direction, proving that the

property is exclusive, program 6 from Listing 6.1 on page 120 has the semi-strong $\approx_{\{\mathtt{L}\}}$-noninterference property, but not strong noninterference since the number of states depends on `H`.
Ad 2: The cases in which one or both traces are infinite are trivial. Assume both traces are finite with lengths ℓ, ℓ'. Then, the first direction of the claim follows through Lemma 6.28. The other direction is obvious since the trace-based notion includes the comparison of more states.
Ad 3: This follows by the same argument as in item 2.
Ad 4: For any agreement $\backsim_O$, it is $\backsim_O^{\zeta_\perp} \subsetneq \backsim_O^{\zeta}$ by definition.
Ad 5: This follows from Lemma 6.4. ◁

Formalizing and constructing proof obligations for stutter-invariant noninterference will be part of future work.

6.4.4 Temporal Declassification

In some situations, it is desirable to release information at a certain point in time in spite of the general confidentiality policy. This is known as the temporal dimension of declassification. Consider, for instance, an election. The individual votes are to be kept secret, but the result, i.e., the sums of votes for each candidate, is published. The result is the subject of the declassification here. But if the result is published every time that a voter casts their vote, then an attacker can deduce all individual votes. The solution is that the result only should be published after the election process is finished.

Noninterference with temporal declassification can be expressed by relaxing the output relation on traces. One possibility is to restrict a relation $\backsim$ to subtraces. For instance the following relation $\backsim^{\varphi}$ contains pairs of traces that are $\backsim$-equivalent until the (temporal) formula φ holds on either one. Let $\backsim^{\varphi}$ be the commutative closure of the following relation.

$$\left\{ (\tau, \tau') \in (\mathcal{S}^*)^2 \;\middle|\; \begin{array}{l} \tau[i,\infty) \vDash \varphi \text{ and } \tau[0,i) \backsim \tau'[0,i) \text{ for some } i \in [0, |\tau|) \\ \quad \textbf{or } (\tau \vDash \Box\neg\varphi \text{ and } \tau \backsim \tau') \end{array} \right\} \tag{6.9}$$

This ‘weak until’ construction is a common pattern (cf. [Chong and Myers, 2004]). In [Bruns, 2014b], the author presented a relational version of DTL that has an explicit operator representing the above relation on the formula level. Other relations can be defined along the same lines. Note that not all meta-level properties on linear traces are expressible using LTL-style temporal operators.

6.4.5 Comparison to Other Trace-based Notions

Sabelfeld and Myers [2003a] introduce a notion of *timing-sensitive* noninterference: in addition to the final states of two terminating runs being low-equivalent, the number of execution states must be the same (or both runs do not terminate). This can be formalized in our framework as follows:

$$\begin{aligned} \forall s, s' \in \mathcal{S}.\ s \approx s' \\ \rightarrow \exists t \in \mathbb{N} \cup \infty.\ (t = |trc_\Sigma(s,\pi)| \wedge t = |trc_\Sigma(s',\pi)| \\ \wedge (t < \infty \rightarrow trc_\Sigma(s,\pi)[t] \approx trc_\Sigma(s',\pi)[t])) \end{aligned} \tag{6.10}$$

Note that our formalization requires deterministic program semantics and that the length of a trace is the number of write events in one single thread.

It is easy to see that this property implies TSNI as defined above. In addition to the final value of each trace, it additionally considers the length of the trace. On the other hand, there are programs that are termination-sensitive, but not timing-sensitive noninterferent; see the example below. Obviously, termination-sensitive semi-strong noninterference implies timing-sensitive noninterference.

Example 6.30. An example of a program that is TSNI, but not Sabelfeld and Myers timing-sensitive is program 3 from Listing 6.1 on page 120:

```
b = H==0; if (b) { x = L; y = H; L = x+y; } else {}
```

The first branch is taken whenever `H` holds the value zero. But in this case the assignment to `L` is vacuous. For any high input, there is only one possible final state. However, the traces leading to the final state may have different lengthes: length 3 for the former case, and 1 for the latter case.

Low-security Observational Determinism

Our definition of strong noninterference in Def. 6.21 is related to the well known notion of LSOD [McLean, 1992; Roscoe, 1995; Roscoe, Woodcock, and Wulf, 1996; Zdancewic and Myers, 2003; Huisman, Worah, and Sunesen, 2006; Terauchi, 2008]. These references differ in the exact definition; we mainly consider the version by Zdancewic and Myers, that is probably the most widely used one. In both our notion and LSOD, programs are only considered secure if they produce indistinguishable outputs for any run.

There are still some major differences—besides our programs being deterministic and the general flexibility of our notion of agreements. The main difference is that Zdancewic and Myers [2003] only require that traces are equivalent up to prefixing and stuttering. Huisman et al. [2006] also consider stuttering, but drop the prefixing relaxation. They point out that it would permit programs that are insecure under standard TINI to be classified

Table 6.4: Comparison of different notions of LSOD trace equivalence

[Zdancewic and Myers, 2003]	$\forall \ell \in L.\ \exists k \in \mathbb{N}_{>0}.\quad \tau\rfloor_\ell[0,k) =^\zeta \tau'\rfloor_\ell \quad \vee \quad \tau\rfloor_\ell =^\zeta \tau'\rfloor_\ell[0,k)$
[Huisman et al., 2006]	$\forall \ell \in L.\ \tau\rfloor_\ell =^\zeta \tau'\rfloor_\ell$
[Terauchi, 2008]	$\exists k \in \mathbb{N}_{>0}.\ \tau[0,k) \approx_L^\zeta \tau' \vee \tau \approx_L^\zeta \tau'[0,k)$

as secure.[14] However, we allow arbitrary finite numbers of non-observable operations (i.e., local assignments and control statements) to be taken in between observable global assignments, while the languages of Zdancewic and Myers and Huisman et al. only have global variables. Above in Sect. 6.4.3, we presented a relaxation of our security notion to allow stuttering. A precise logic formalization is left to future work.

Neither Zdancewic and Myers nor Huisman et al. consider traces of the complete memory, but for projections to each low variable on its own. This still allows internal timing attacks where the order of assignments to low variables is reversed under a high condition. Zdancewic and Myers point out that this only occurs with programs that have data races, which they exclude from the input set of their analysis.

Example 6.31. Consider the following program. It branches on a high value and assigns the same values to the low variables in different order in each branch. The program is secure according to Zdancewic and Myers and Huisman et al. An attacker who is able to observe intermediate states will deduce the value of `H` from the observation of which low variable is updated first.

```
b = H; if (b) { L1= 3; L2= 7; } else { L2= 7; L1= 3; }
```

To formally represent LSOD considering single values, we introduce a notion of projection from a trace τ to a *value trace* $\tau\rfloor_\ell \in \mathcal{U}^*$, that contains the values of a global variable ℓ for each state on the trace. By abuse of notation, we use the same operations on value traces as on traces without given a formal definition.

Another possible weakening of our definition of strong noninterference would be to allow one trace to be equivalent to a *prefix* of the other one, as in the definitions by Zdancewic and Myers [2003]; Huisman et al. [2006]. Terauchi [2008] also defines LSOD for prefixes, but considers the complete 'low' partion of the heap for equivalence, instead of just value traces. We do not consider prefixing per se, as we assume the attacker to always observe changes to the global state. In particular, noninterference relaxed to prefixes is ignorant to termination leaks. While Zdancewic and Myers argue that termination leaks would 'only' reveal 1 bit of information, the following example shows that arbitrary information is leaked. Nevertheless, prefixing can be

[14] Nevertheless, the type system by Zdancewic and Myers rejects these programs.

modeled through the temporal declassification relation $\smile^{\varphi}$ (see Sect. 6.4.4) with the formula *false* inserted for φ in (6.9).

Example 6.32. Consider the program example in Listing 6.5, that is originally by Giffhorn and Snelting [2015]. The program does not terminate for any input, but writes all values that are less than the secret value of `H` to the low location `L`. An attacker who is able to observe nontermination, can deduce the secret entirely from this. Yet, this program is secure according to Zdancewic and Myers and Terauchi since the trace prefixes are always equivalent (i.e., the secret value only appears in the longer trace).

```
i = 0;
while (true) {
    while (H == i) { skip; }
    L = i;
    i = i+1;
}
```

Listing 6.5: Termination leak that reveals the entire secret

Table 6.4 on the preceding page compares the trace equivalence relation in different notions of LSOD for a 'low' location set L, using our notation. It is easy to see that our definition of $\approx_L$-equivalence implies all of these notions. Figure 6.6 on the next page displays the hierarchy of all information flow properties that have been introduced so far.

6.5 Object-sensitive Noninterference

In this section, we show how our notion of noninterference can be refined such that it is adequate to reason about information flows in Java programs. Object-oriented programing languages, like Java, do not expose internal information of object structures, such as memory addresses—in contrast to pointers and data structures in C/C++. This reduces the range of possible attacks in a language based scenario. Therefore, the definition of low-equivalence needs to be widened to allow that, between different equivalent runs, semantical objects may appear that may not be identical in the mathematical meaning. Instead, they have to be considered equivalent w.r.t. *observable behavior* as permitted by the language.[15] Objects can still be compared for identity for each run locally; but a comparison between different runs is meaningless. One particular implication from that is that object creation does not depend on secrets.

[15]This equivalence is what Java's `equals()` method is *supposed* to implement; however, we cannot rely on `equals()` being implemented this way.

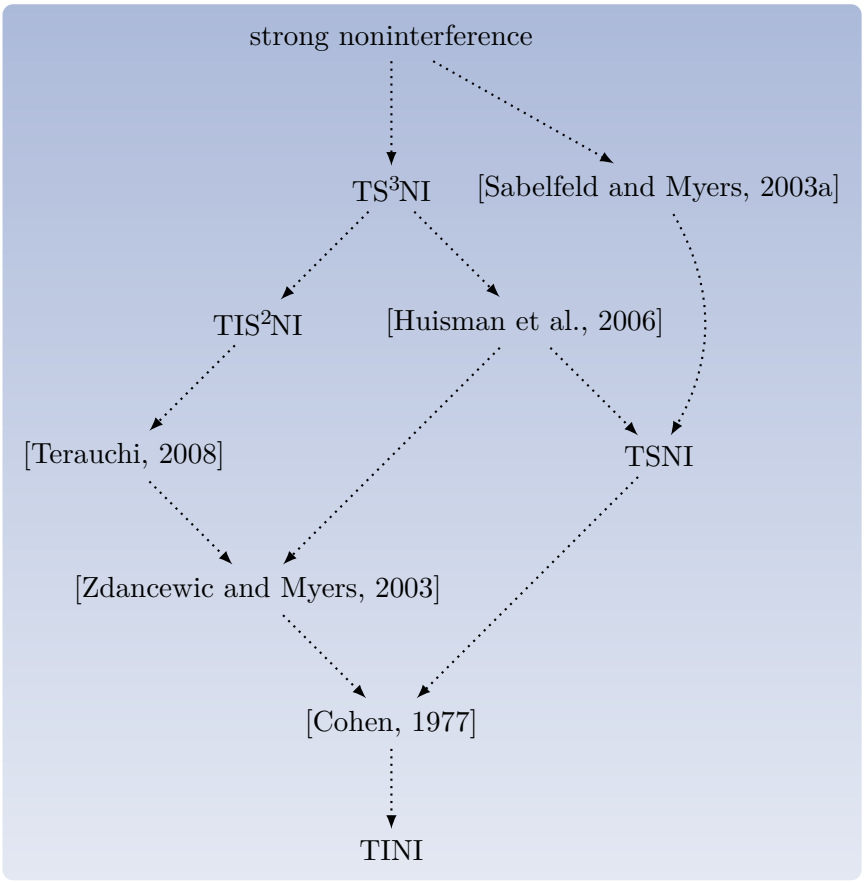


Figure 6.6: Hierarchy of information flow properties

This section is based on a complete revision of the paper by Beckert, Bruns, Klebanov, Scheben, Schmitt, and Ulbrich [2014]. The idea of this revision is to concentrate on the particularities of object-oriented programs here in this thesis, and to simplify everything else. This includes using a more traditional style of noninterference specification using 'high' and 'low' variables (plus declassification, etc.) as above, instead of the more involved concept of observation expressions [Scheben and Schmitt, 2012a]. In order to not having to introduce the complete JavaDL of Beckert et al. [2007a], our discourse here will mainly take place on the semantical level, but not in the logic itself.

6.5.1 Overview

In imperative languages, sources and sinks are of primitive type. In an object oriented context, it is natural to consider sources and sinks of object type, too. In this case, the usual definition of secure information flow—if a system is started in two low-equivalent states s_1, s_2 with all publicly observable

values equal, then it terminates in states s_1', s_2' where all observable values are mathematically equal—is too strong to be sensible. The reason is that in object-oriented languages with opaque pointers, such as Java, object references do not hold much information themselves. The language only allows to compare object references pairwise and to ask for their runtime type.[16]

Banerjee and Naumann [2002] were the first to introduce a formal notion of object-sensitive secure information flow, that has a modified notion of low-equivalence of states: if a system is started in two states s_1, s_2 such that the observable values are related by a partial isomorphism ρ_1, then it terminates in states s_1', s_2' where all observable values are related by a partial isomorphism ρ_2 extending ρ_1. As proven by Beckert et al. [2014], the partial isomorphism in the prestate ρ_1 can be chosen as the identity function, without loss of generality.

For a more precise formalization of object-sensitive noninterference, it is reasonable to encode the partial isomorphisms explicitly in logic. However, this holds the disadvantage that a naïve encoding either increases the burden on the analysis or the burden on the user, the latter by requiring additional annotations (cf. [Naumann, 2006]). We show that additionally restricting the partial isomorphism in the poststate to newly created objects leads to a property that is sufficient for object-sensitive secure information flow and can be proven efficiently.

6.5.2 Information Flow in Java Programs by Example

In Listings 6.7 to 6.11 on pages 149–150 we reproduce typical examples of (secure or insecure) information flow in Java programs. This is to motivate our attacker model and the ensuing notion of object-sensitive noninterference (OSNI). Some examples are intuitively secure, but the standard notion of low-equivalence will lead to a classification as insecure. Table 6.12 on page 151 at the end of this section contains a summary of the examples.

The static fields `x`, `y`, and `z` of a (`final`) reference type `C` are low, as is the instance field `a`, and the static boolean field `h` is the only high source. Note that we consider semantical locations (i.e., pairs of semantical objects[17] and field identifiers) as sources/sinks, not just the syntactical field identifier. When we speak of an instance field being a source, more precisely we mean the set of locations belonging to any possible receiver object.

In method `newC()` in Listing 6.7, a fresh object is allocated and assigned to a low location. Even though there is no confidential source involved at all, this program is not secure w.r.t. to standard low-equivalence (cf. Lemma 6.2). The reason is that the values of `x` in different program runs depend on the

[16] The ability to see the runtime type arises from the fact that Java is type-safe. This information would not be derivable in type-unsafe languages.

[17] For static fields, the location lacks a receiver object.

```
static C x, y, z;  // low locations
int a;             // low location
static boolean h;  // high location

static void newC() {
    x = new C();
}
```

Listing 6.7: This method is insecure w.r.t. standard noninterference definitions.

behavior of the virtual machine which chooses the freshly created objects. The Java Virtual Machine Specification does not impose any restrictions on the choice of new object references apart from the fact that they are not already in use. This is usually modeled through underspecification (cf. [Beckert et al., 2007b, Sect. 3.6.6]).

Therefore, we cannot ensure that the values $\mathtt{x}^s$ and $\mathtt{x}^{s'}$ for respective terminal states s and s' are identical (nor that they are different). This proof would require that the initial states are identical—even with the deterministic program semantics that we assume. On the other hand, method `newC` obviously does not leak information as it does not involve high components at all. Thus, a standard noninterference condition based on object identity is too strict in an object-sensitive setting.[18]

For method `newHigh()` of Listing 6.8, the observation of an attacker depends on the value of the secret variable `h`. Since the allocation process always returns an object that is different from all previously existing objects, the attacker can deduce that `h` is true if and only if the value of `x` changes. Information is leaked here.

```
static void newHigh() {
    if (h) { x = new C(); }
}
```

Listing 6.8: Information leakage through a fresh object

In contrast, method `order()` in Listing 6.9 intuitively does not leak any information. Here, the concrete values of `x` and `y` may also depend on the value of `h` (because of the permutated order). But an attacker can only compare the two references through `==` (object identity) and deduce that they are different in any run. They cannot deduce the order in which the objects were allocated.

[18] Note that in any case, the fresh object must be assigned to a low location; a program that allocates objects, but does not store the reference is vacuously secure.

```
static void order() {
    if (h) {
        x = new C();
        y = new C();
    } else {
        y = new C();
        x = new C();
    }
}
```

Listing 6.9: The order of object creation is not observable.

In method `swap()` in Listing 6.10, it is important to notice that the attacker does not only observe the (unordered set of) *values* of expressions, but knows the evaluation function itself. The sets of values $\{\mathtt{x}^s, \mathtt{y}^s, \mathtt{z}^s\}$ and $\{\mathtt{x}^{s'}, \mathtt{y}^{s'}, \mathtt{z}^{s'}\}$ are equal in any case. However, the change made to `x` is observable since it can be compared to both `y` and `z`.

```
static void swap() {
    if (h) { x = y; }
    else { x = z; }
}
```

Listing 6.10: Information flow through swapping observable locations

The last example involves nontrivial heap structures. In method `deref()` in Listing 6.11 both the values of `x` and `y` and of `x.a` and `y.a` are swapped under a high condition. This means that the values of the compound expression `x.a` (and likewise `y.a`) is equal in any two runs. But since we do regard locations, not expressions, it is distinguishable between the value of the location $(o, \mathtt{a})$ in two runs with different high input, where o is the prestate value of `x`. Table 6.12 on the next page provides a summary of the examples in this section.

```
static void deref() {
    if (h) {
        C c = x; x = y; y = c;
        int a = x.a; x.a = y.a; y.a = a;
} }
```

Listing 6.11: Complex value changes in object structures

Table 6.12: Summary on the examples in this section

	`newC()`	`newHigh()`	`order()`	`swap()`	`deref()`
intuitively secure	✓	✗	✓	✗	✗
noninterferent	✗	✗	✗	✗	✗

6.5.3 Attacker Model

We describe publicly (and thus attacker) visible parts of the program state as sets of semantical locations, i.e., pairs of semantical objects and field identifiers. The attacker can evaluate *primitive type* expressions in the initial and final states of any program run and can compare the values between any two runs. This includes declassification. The attacker is able to provide low security input. Further, we assume that they know the program code. This allows them to trace back the observed differences in low sources in the poststate to high sinks in the prestate. By the phrase 'can compare,' we understand that an attacker

- *can* compare observed values that are of a primitive type to each other and to literals (of that type) as by using `==`;
- *can* compare observed values of object reference type to each other and to `null` as by using the `==` operator, observe their (runtime) type through the `instanceof` operator, and the `length` attribute for array references;
- *cannot* learn more than object identity from object references (e.g., the order in which objects have been generated cannot be learned).

In the Java language, object references are treated as opaque values. as specified in the Java Language Specification (JLS). This is, references can only be compared using the equality operators `==` (equality) and `!=` (inequality), cf. JLS, Sect. 15.21.3, as well as the type predicate operator `instanceof`. In constrast, in programing languages where references have more structure (e.g., numeric pointers in C/C++, cf. [Banahan et al., 1991, Sect. 5.3.4]), stronger attackers are possible. For instance, if a particular memory manager happens to allocate memory in ascending order, an attacker on a C++ program analogous to `order()` could deduce that $\texttt{h}^s$ is true if and only if the numerical value of `x` is less than the value of `y`. Such inference is not possible in the Java language itself. Implementations of native methods, however, may provide some loopholes which leak structural information on references. Most notably, the native method `Object#hashCode()` returns the (encoded) memory address of a reference, as pointed out by Hedin and Sands [2006].

Similar exploits can be found with the class `sun.misc.Unsafe`, that is part of Oracle's Java Development Kit (JDK), but not of the official application programming interface (API) and its usage is officially 'discouraged.' This leakage potential can be dealt with by assigning a high security level to the output of native methods.

The examples show that information may flow through references, but nonidentical behavior is not a sufficient indication of a leak. Executions need not behave *identically* for different high inputs, but they must behave *congruently* with respect to reference comparison. This means that the poststates may be different as long as there is a kind of one-to-one correspondence between their references that is compatible with the identity comparison operation. In particular, the values of two locations storing references need to coïncide in one poststate exactly if they do in the other.

6.5.4 Object Isomorphisms

Our formalization of dWRF programs from Chap. 3 can be extended naturally to accommodate objects and fields instead of global variables. This sufficiently captures the essentials of Java-like object-orientation that are required for this section. We introduce a new type $\mathbb{O}$ representing objects. It may have subtypes, as declared by the program, that are commonly called *reference types*. All reference types have a common subtype containing only the special element *null* and are disjoint otherwise. The exact type hierarchy is not important for our discourse.

Locations are redefined to be pairs of semantical objects of types $\mathbb{O}$ and $\mathbb{F}$ as originally defined by Weiß [2011], i.e., $(o, f) \in \mathcal{D}_{\mathbb{O}} \times \mathcal{D}_{\mathbb{F}}$ is a location; the semantics of location set expressions is changed accordingly, in particular $\mathcal{D}_{\mathbb{L}} \subseteq 2^{\mathcal{D}_{\mathbb{O}} \times \mathcal{D}_{\mathbb{F}}}$. For all instances of non-*null* subtypes of $\mathcal{D}_{\mathbb{O}}$, there is a special boolean field `created`, that indicates whether an object has been completely initialized (as defined by the JLS). For all instances of array types, there is a field for every index in range and a special field `length`. For any reference type A, there is a special A-typed static field A`::nextToCreate`. This field indicates the next object of type A to be allocated, i.e., the value of its `created` field will be set to `true` in a call to `new` A`()`.[19] The signatures of *select* and *store* (cf. Sect. 3.3 on page 38) are changed accordingly. For uniformity of notation we will frequently write $f(o)$ instead of $\mathit{select}(\mathtt{heap}, o, f)$. In addition, there is a unary syntactical predicate $\mathit{exactInstance}_A$ for each type A to express that an object is of that type: $s \vDash \mathit{exactInstance}_A(o)$ if and only if $\mathit{type}(o^s) = A$.

[19] This is the usual way to represent object creation in constant domain program logics (cf. [Beckert et al., 2007b, Sect. 3.6.6]).

Definition 6.33. By $\Omega(L^s)$ we denote the set of objects observable by a location set expression L in state s, that is, $\Omega(L^s) = \{o \in \mathcal{D}_{\mathbb{O}} \mid \exists \ell \in L^s.\ (\ell \mapsto o) \in \mathtt{heap}^s\}$.

In an object-oriented setting, the set of observable objects may depend on the state. For example, if *o.next.val* is observable, then it depends on the state what object *o.next* evaluates to. Moreover, if all locations in a linked list are observed, for instance, then the *number* of observable locations may depend on the state, since the list length does. We cater for this because location set expressions are rich in their expressivity, including both conditional and reachability operators.[20]

Isomorphism

We assume that the reader is familiar with the concept of isomorphism for typed structures and function extensions, as treated, e.g., by Mitchell [1990]. We introduce the notion of object isomorphisms on the computation domain $\mathcal{D}$. We use the notation $(\mathcal{D}, s)$ for a first-order structure where the interpretation I is clear from the context (see Sect. 4.3).

Definition 6.34. Let $\rho : \mathcal{D} \to \mathcal{D}$ be a bijection. We extend ρ to a bijection on states $\rho : \mathcal{S} \to \mathcal{S}$ through $v^{\rho(s)} := \rho(v^s)$ for a state $s \in \mathcal{S}$ and a variable v. We write $\rho : (\mathcal{D}, s_1) \mapsto (\mathcal{D}, s_2)$ if $\rho(s_1) = s_2$.

We call ρ an *automorphism* of $\mathcal{D}$ if for any state s, expression x, and first-order formula φ it holds that $x^{(I,s,\beta)} = x^{(I,\rho(s),\beta)}$ and $(I, s, \beta) \vDash \varphi \Leftrightarrow (I, \rho(s), \beta) \vDash \varphi$ (w.r.t. a fixed first-order structure $(\mathcal{D}, I, s)$ and variable assignment β).

Lemma 6.35 ([Beckert et al., 2014, Lemma 2])**.** *Let $\mathcal{D}$ be a domain and $\rho' : X \to Y$ be a bijection for finite subsets $X, Y \in 2^{\mathcal{D}_{\mathbb{O}}}_{\text{fin}}$ with*

1. *if $null \in X$ then $\rho'(null) = null$ and $null \in Y$ implies $null \in X$,*
2. *ρ' preserves types, i.e., for all expressions h, o, f of types $\mathbb{H}, \mathbb{O}, \mathbb{F}$, we have $type\,((select(h, o, f))^s) = type\left((select(h, o, f))^{\rho(s)}\right)$,*
3. *ρ' preserves the length of array objects, i.e., for all expressions h, o we have $(select(h, o, \mathtt{length}))^s = (select(h, o, \mathtt{length}))^{\rho(s)}$,*

Then there is a total automorphism ρ of the complete domain $\mathcal{D}$ that extends ρ'.

Definition 6.36 (Partial isomorphism w.r.t. L)**.** Let L be a location set expression and s_1, s_2 be two states. A *partial isomorphism* with respect to L from s_1 to s_2 is a partial function $\rho : \mathcal{D} \nrightarrow \mathcal{D}$ such that

[20] For the sake of brevity, we have not introduced reachability in this work; cf. the comprehensive theory of location sets by Weiß [2011].

1. for all primitive values $w \in \mathcal{D} \setminus \mathcal{D}_{\mathbb{O}}$, it is $\rho(w) = w$,
2. $\rho|_{L^{s_1}}$ is a total bijection from L^{s_1} to L^{s_2},
3. $\rho|_{\Omega(L^{s_1})}$ is a total bijection from $\Omega(L^{s_1})$ to $\Omega(L^{s_2})$, and
4. the requirements of Lemma 6.35 hold for the bijection $\rho|_{\Omega(L^{s_1})}$.

Condition 2 amounts to the usual requirements on isomorphisms on mathematical structures. If $p \in L$ for all program variables p, every automorphism extending a partial isomorphism ρ with respect to L according to Def. 6.34 is a total isomorphism from $(\mathcal{D}, s_1)$ onto $(\mathcal{D}, s_2)$ since $\rho(p^{s_1}) = p^{s_2}$ by requirement 2. Not every partial isomorphism can be extended to a total isomorphism, on the other hand. If q is a program variable such that q does not appear as a subterm in L, then $\rho(q^{s_1}) = q^{s_2}$ is not required.

6.5.5 Defining Object-sensitive Noninterference

As mentioned above, we treat object references as opaque. This means in particular that the behavior of a Java program cannot depend on the values of references up to comparison through `==`. Hence, if a program π is started in two isomorphic states, then π also terminates in isomorphic states (if π terminates at all.) Though this assumption is not always made explicit, it is widely used in literature, cf., e.g., Myers [1999]; Banerjee and Naumann [2002]; Naumann [2006]. Opaqueness of references can be formalized in our setting as follows:

Lemma 6.37 (Beckert et al. [2014, Postulate 1])**.** *Let s_1, s_2 be states. Let π be a program which started in s_1 terminates in s_2, and let $\rho : \mathcal{D} \to \mathcal{D}$ be an automorphism.*

Then π started in $\rho(s_1)$ terminates in $\rho'(s_2)$, where $\rho' : \mathcal{D} \to \mathcal{D}$ is an automorphism that coïncides with ρ on all objects existing in state s_1, i.e. for all $o \in \mathcal{D}_{\mathbb{O}}$ with $\texttt{created}^{s_1}(o) = true$ we know $\rho(o) = \rho'(o)$.

The reason why we cannot assume $\rho = \rho'$ directly, is that π may generate new objects and there is no reason why a new element o' generated in the run starting in state $\rho(s_1)$ should be the ρ-image of the new element o generated in the run of π starting in state s_1.

We now define the notion of the object-sensitive low-equivalence relation $\backsimeq^{\Omega}$. It will form the basis of the object-sensitive noninterference (OSNI) property for Java. Apart from the more flexible assignment of security levels, that is introduced through location set expressions, this property does not yet exceed the state of the art in object-sensitive non-interference [Banerjee and Naumann, 2002]. We consider here the termination-insensitive case. Extensions taking termination into account (cf. Sect. 6.2.5), as well as differentiating between normal and abnormal termination, are straightforward (cf. Scheben and Schmitt [2012a]).

Definition 6.38 (Object-sensitive low-equivalence)**.** Let L be a location set expression. We say that two states s, s' are *object-sensitive L-equivalent* w.r.t. ρ, denoted by $s \backsimeq_L^{\Omega\rho} s'$ iff there exists a partial isomorphism $\rho : \Omega(L^s) \to \Omega(L^{s'})$ with respect to L. The partial isomorphism ρ is uniquely determined by L, s and s'. We use the notation $s \backsimeq_L^{\Omega} s'$ to indicate that there exists an isomorphism ρ such that $s \backsimeq_L^{\Omega\rho} s'$.

Notice that because of our tacit definition of partial isomorphisms on primitive values, $s \backsimeq_L^{\Omega\rho} s'$ entails $x^s = x^{s'}$, if x is an expression of primitive type. It is easy to see that the relation $\backsimeq^{\Omega}$ is a generalization of the relation $\approx$ in Def. 5.1, as stated by the following lemma. In particular, $\backsimeq_L^{\Omega id} = \approx_{L^s}$ for the identity isomorphism *id*.

Lemma 6.39. *The relation $\backsimeq_L^{\Omega\rho}$ is an agreement and an equivalence relation.*

We refer to (termination-insensitive/termination-sensitive) $\backsimeq_{L_1}^{\Omega\rho_1}/\backsimeq_{L_2}^{\Omega\rho_2}$-noninterference as (termination-insensitive/termination-sensitive) *object-sensitive noninterference*. The intuition behind this definition is that L_1 describes the low locations in the prestate and L_2 describes the low locations in the poststate. Thus, the values of the variables and locations in L_2 in the poststate may depend at most—up to isomorphism of states—on the values of the variables and locations in L_1 in the prestate and not on anything else. Beckert et al. [2014, Lemma 4] prove that, without loss of generality, the identity isomorphism *id* can be used in the input agreement to yield $\backsimeq_{L_1}^{\Omega id}$, while the output agreement is $\backsimeq_{L_2}^{\Omega\rho_2}$ for some isomorphism ρ_2. Termination-insensitive object-sensitive noninterference is called the "*flow* property" by Beckert et al. [2014].

In the most common case, the low locations before program execution will be the same as the low locations after program execution, i.e., $L_1 = L_2$. Declassification can be added by strengthening the input agreement as described above in Sect. 6.2 on page 123.

6.5.6 Formalizing Object-sensitive Noninterference

Partial object isomorphisms can be difficult to formalize. Beckert et al. [2014] present a formalization of termination-insensitive object-sensitive noninterference in Java Dynamic Logic [ibid., Sect. 6]; as well as a similar, sufficient property, that is provable more efficiently [ibid., Sect. 7]—and more readable. In the following, we present a simplified formalization of this sufficient property, which is called the "*flow*** property" by Beckert et al. and *strong object-sensitive noninterference* by Scheben [2014]. As described by Scheben, these proof obligations are implemented in the KeY system to formally verify information flow in sequential Java programs.

We use the shorthand $h \dot{\approx}_L h'$ to denote—similar to Formula (6.2) on page 130, but with location set expressions—the syntactical equivalent to

the (formalizable) semantical low-equivalence relation $s \approx_{L^s} s'$ (cf. Def. 5.1 on page 93) with $h = \texttt{heap}^s$ and $h' = \texttt{heap}^{s'}$. More formally, it stands for the formula

$$\begin{aligned} & \{\texttt{heap} := h\}L \doteq \{\texttt{heap} := h'\}L \\ \wedge \quad & \forall \ell \in \{\texttt{heap} := h\}L.\ \mathit{select}(h, \ell) \doteq \mathit{select}(h', \ell) \qquad . \end{aligned} \tag{6.11}$$

The formalization of strong OSNI follows the same pattern as in Sect. 6.2.5 above: the formula describes the relation of the four heap objects h_1, h_1', h_2, and h_2', where h_1 and h_1' are two low-equivalent heaps in the prestate and h_2 and h_2' are the heaps reached through the execution of a sequential program π. The formalization is displayed in (6.12) below. Following the result of Beckert et al. [2014, Lemma 4], the input agreement can be chosen with the identity isomorphism.

$$\begin{aligned} \forall h_1, h_1', h_2, h_2'{:}\mathbb{H}. \qquad & \{\texttt{heap} := h_1\}\langle\pi\rangle\,\texttt{heap} \doteq h_2 \\ & \wedge \{\texttt{heap} := h_1'\}\langle\pi\rangle\,\texttt{heap} \doteq h_2' \\ & \wedge h_1 \dot{\approx}_{L_1} h_1' \\ & \rightarrow (\mathit{newIso} \wedge (h_2 \dot{\approx}_N h_2' \rightarrow h_2 \dot{\approx}_{L_2} h_2')) \end{aligned} \tag{6.12}$$

The difference to the formalization of TINI in (6.2) is the weaker output agreement. In strong OSNI, the observable objects that are freshly allocated in the post-state are named explicitly in set N. The poststate isomorphism only differs from the identity function on these new objects. There is no proof obligation that it extends the identity; it does by construction. Yet, it is to be proven that these objects are actually fresh. This is what is formalized in the formula *newIso*, that is displayed in (6.13) below.

$$\begin{aligned} \mathit{newIso} := \quad \forall o \in N.\ (\quad & \mathit{select}(h_1, \mathit{select}(h_2, o), \texttt{created}) \doteq \mathit{false} \\ & \wedge \mathit{select}(h_1', \mathit{select}(h_2', o), \texttt{created}) \doteq \mathit{false} \\ & \wedge \bigwedge_{A \in \mathit{Types}} \mathit{exactInstance}_A(\mathit{select}(h_2, o)) \\ & \qquad\qquad \leftrightarrow \mathit{exactInstance}_A(\mathit{select}(h_2', o)) \\ & \wedge \forall o' \in N.\ (\mathit{select}(h_2, o') \doteq \mathit{select}(h_2, o) \\ & \qquad\qquad \leftrightarrow \mathit{select}(h_2', o') \doteq \mathit{select}(h_2', o))) \end{aligned} \tag{6.13}$$

The following theorem states that formula (6.12) formalizes an implicant of object-sensitive noninterference.

Theorem 6.40. *Let π be a program and let L_1, L_2, N be location set expressions, where all elements of N are of type $\mathbb{O}$. If the formula in* (6.12) *is valid, then π is termination-insensitive $\asymp^{\Omega}_{L_1}/\asymp^{\Omega}_{L_2}$-noninterferent.*

Proof. The theorem follows from [Beckert et al., 2014, Thm. 2] in combination with [ibid., Lemma 6]. ◁

6.5.7 Conclusion

The definition of object-sensitive low-equivalence introduced in this section is a simplified version of the one introduced by Beckert et al. [2014]. The main difference is that Beckert et al. do not use the traditional concept to secure information flow specification where sets of locations (i.e., sources) are classified—modulo declassification. Instead, they allow arbitrary expressions to be classified. This framework unifies location classification and declassification. Location sets are replaced by so-called *observation expressions* [Scheben and Schmitt, 2012a; Scheben, 2014]. See also Sect. 8.4 on specification using observation expressions. The motivation behind this assumes a different attacker model, that is meant to better reflect an observer of the system: The attacker sees an expression and the corresponding evaluation in the prestate and poststate of a method as if they were printed on a screen. A discussion on these different attacker models can be found in Sect. 2.3.

We avoid introducing observation expressions here, as it does not increase expressivity, but incorporates a kind of higher order concept: The semantics of an observation expression is a sequence of syntactical expressions again. This construction is wellfounded, because the set of welldefined expressions is fixed a priori, but still poses a significant extension to the present logic framework. Locations, on the other hand, are well understood semantical entities (cf. the theory of heaps by Weiß [2011]).

Furthermore, since there is more than one expression that has the same semantics, observation expressions have fixed orderings. For instance, in method `swap()` of Listing 6.10 on page 150, it is important to notice that the attacker does not only observe the unordered *set* of values of expressions, but knows the evaluation function itself. The sets of values $\{\mathtt{x}^s, \mathtt{y}^s, \mathtt{z}^s\}$ and $\{\mathtt{x}^{s'}, \mathtt{y}^{s'}, \mathtt{z}^{s'}\}$ are equal in any case. However, the change made to `x` is observable. The usual way to account for this is to impose an ordering on the references that are regarded, i.e., using sequences instead of sets. Imposing some ordering in specification is not intuitive.

Observation expressions are defined as sequences of finite, but not necessarily a priori fixed, length. This means that to any Scheben and Schmitt style noninterference specification, there is an equivalent definition of noninterference (in the sense of Def. 6.3): Scheben and Schmitt noninterference with prestate observation X and poststate observation X' is equivalent to $\circledast_I^{\Delta(X)^s} / \circledast_O^{\Delta(X')^s}$-noninterference with declassification and

erasure (cf. Sect. 6.2.3). However, not all of them can be represented syntactically in our logic, since observation expressions correspond to an unbounded number of declassification expressions.

Discussion

We have not formalized object-sensitive noninterference (OSNI) directly. It can be found in [Beckert et al., 2014, Sect. 6]. Instead, we presented a slightly stronger property, called strong object-sensitive noninterference, which is a sufficient criterion. The main difference between (original) OSNI and strong OSNI is that strong OSNI also guarantees security against an attacker that possesses the ability to distinguish between newly created objects and objects which already existed in the prestate.

Example 6.41. Consider the program `if (h) { new C(); }`. It is similar to Listing 6.8 in that a fresh object is allocated depending on high data. The difference here is that the fresh object is not referenced in any way. According to the original definition of OSNI, the program is secure since the mere object allocation is not observable. However, it is insecure under *strong* OSNI since we additionally permit the attacker to observe allocation.

Beckert et al. [2014] further show that strong OSNI is sequentially compositional. Compositionality is considered an indispensable prerequisite for modular verification of information flow properties. For the original OSNI property, it holds only under certain conditions.

6.6 Fusing Together A Semantically Precise Information Flow Analysis for Concurrent Java

Above, in Sects. 6.3f., we have defined what secure information flow means for concurrent dWRF programs. These definitions are purposely introduced in an abstract notion, in order to instantiate them later in a uniform way. This is provided through the notion of agreement (or trace equivalence, respectively). This particularly allows to 'plug in' semantical declassification or preconditions into the security property, as explained in Sects. 6.2.3f. Furthermore, it allows to relax the indistinguishablity relation such that it is appropriate for the observation of opaque object references (Sect. 6.5).

We have mainly described information flow analysis for dWRF programs. There is no fundamental barrier to lift the results of this chapter to the full Java language. The work by Scheben and Schmitt [2012a] regards (sequential) Java programs all the way. They present a formalization of nonintereference in JavaDL. A particular challenge in their work was to consider that data in memory is structured through objects. Low-equivalence is expressed with the help of explicit location set and heap expressions, that are evaluated dynamically, as introduced by Schmitt et al. [2011]. We have

acknowledged this in the present dissertation: even though dWRF only features global variables of primitive types, we have defined its semantics in Sect. 3.4 in terms of an explicit heap representation. Also, our formalizations of noninterference feature location set expressions, thus incorporating the flexibility of a dynamic security classification.

Information Flow Through Exceptions

Another aspect to consider for an extension to full Java are exceptions. In method-modular Java, sinks can be either the memory, the return value of a non-`void` method, or the exception that is raised during execution. For instance, the program `{ L = H/H; }` terminates normally and writes the constant value 1 to `L` for all initial states where the value of `H` is not 0. In the single case in which `H` holds the 0 value a `NullPointerException` is raised. This case appears to be a termination channel, but we can obviously construct other examples that leak more than 1 bit of secret information. Although mentioned by Scheben [2014], he does not further explain how (absence of) information flow through the occurrence of exceptions can be expressed. Adding a check for exceptions in the above formalization of noninterference should not pose a major difficulty. As explained by Beckert et al. [2007b], programs can be wrapped with `catch` statements to isolate a raised exception; absence of an exception can be modeled through a `null` value.[21]

Semantically Precise Analysis

Above, we have introduced a state-based and a trace-based notion of noninterference. These notions are semantical; analysis techniques based on syntax can never be precise—by design. In this dissertation, we pursue a logic-based approach, which aims at providing ultimate precision. Ideally, it entails two components: (i) a precise formalization of the semantical property and (ii) a sound and complete deduction system for the ensuing formula.

Ad (i): according to Thm. 6.11 on page 130, the formalizations in CDTL in (6.2) and (6.3) faithfully capture (state-based) noninterference. We expect a similar result from the formalization of strong noninterference in (6.8), although we do not provide a proof here. The result of Thm. 6.11 is restricted by $\triangleright t$, though it seems that this restriction may be lifted, given that we defined schedulers as fair.

Ad (ii): We have presented a calculus for CDTL in Sects. 4.4 and 5.4. The calculus is proven sound under the condition that all thread specifications that are used are actually valid. Hence—given that we already proved thread

[21] Java does not allow `null` to be raised. An attempt to do so will raise a fresh `NullPointerException` instead.

specification validity—a closed proof of one of the formulae above means that the program under investigation is actually noninterferent. Furthermore, we believe that the calculus is complete relative to the defined interleaving semantics.[22] To this end, in the rely/guarantee approach, a guarantee condition is a complete functional description of environment steps. Under that assumption, any secure program can be proven as such.

We should note, however, that this approach is not per se suitable to detect insecure programs: a failed proof attempt can arise because the prover was not able to find a solution to the theoretically undecidable problem within a given time frame. A graphical representation of the approach overview is given in Fig. 11.1.

Tool Support

We aim at providing tool support for this analysis in the KeY system, as it will be described in detail in Chap. 7. Following the work by Scheben and Schmitt [2012a], the KeY system can formally verify noninterference in sequential Java programs. Furthermore, an extension of KeY to reason about multi-threaded Java following the rely/guarantee approach described in Chap. 5 has been implemented prototypically. In combination, this allows us to verify secure information flow in multi-threaded Java programs with KeY. Through high-level annotations in JML, the user can provide the necessary specification in a convenient way. JML and its extensions for information flow and concurrency will be introduced in Chap. 8.

[22] Actual reliable completeness proofs w.r.t. program behavior are hard—if not practically impossible—to attain.

7 A Verification System for Multi-threaded Java

In this chapter, we present an implementation of the rely/guarantee approach to compositional verification of shared memory concurrent programs presented in Chap. 5. We use an extension of the Java dynamic logic (JavaDL) and the KeY verification system. KeY is a seasoned verification system for sequential Java. At its core lies an interactive theorem prover for JavaDL. We extend the proof system of Chap. 5 to a rich subset of the Java language by incorporating the rules into JavaDL. In Sect. 7.1, we first introduce JavaDL and the KeY system briefly. In Sect. 7.2, we discuss particularities and issues with concurrent Java, that were not considered in the context of the simple language of Chaps. 4–5. Finally, in Sect. 7.3 we elaborate on design decisions and implementation details related to the rely/guarantee approach. Section 7.4 describes the proof obligations for secure information flow in concurrent Java programs.

7.1 The KeY Platform for Verification of Java Programs

The KeY system offers a wide platform of tools for verification and analysis of sequential Java programs. KeY has been developed through the time of more than one decade (cf. [Ahrendt et al., 2005]), thus having achieved a mature state. While the system evolved around an interactive theorem prover for JavaDL [Beckert, 2001], proofs can often be found without interaction—provided an appropriate specification. Specification for Java programs can be given in the Java Modeling Language (JML), which will be introduced in detail in the following chapter. By today, the KeY platform incorporates support for different input languages (such as ABS); functional, relational,

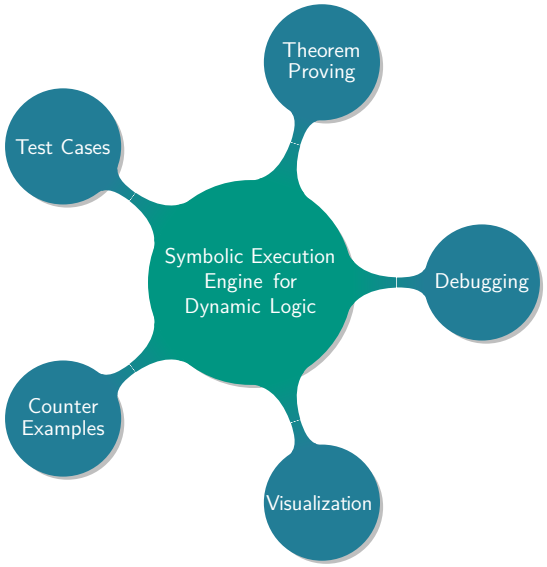


Figure 7.1: Multiple facets of analysis in the KeY framework joined together by the symbolic execution engine

and nonfunctional properties to be proven; and it combines several complementary analysis techniques. A comprehensive, but slightly dated, account on KeY is provided by Beckert, Hähnle, and Schmitt [2007a]. A follow-up [Ahrendt et al., 2016] is currently in preparation to be published in 2016. The most recent up-to-date overview over KeY is provided by Ahrendt et al. [2014].

The kinds of properties that can be proven about programs include 1. *functional* properties such as correctness w.r.t. preconditions and post-conditions or framing (i.e., write effect correctness; cf., e.g., [Bruns et al., 2015b]), 2. *relational* properties such as secure information flow [Scheben and Schmitt, 2012a] or absence of code regression [Beckert et al., 2015], and 3. *nonfunctional* properties such as memory consumption or worst case execution time [Albert et al., 2011]. The target language is a rich subset of the Java language [Gosling et al., 2014], including objects, inheritance, exceptions, static initialization, unbounded recursion [Bubel, 2007], enhanced for loops [Ulbrich, 2007], strings, etc. Floating point data types and generic types are currently not considered.[1]

KeY also supports the JavaCard language, which is not a strict subset of Java. JavaCard is covered entirely [Mostowski, 2006, 2007b], which includes features for abortable transactions. Din [2014] describes a variant of the KeY system that considers the concurrent ABS language as the target. It uses a variant of the rely/guarantee approach to reason about interleavings. KeY-ABS currently is being integrated into the master development branch

[1] See [Ulbrich, 2007] on extending JavaDL with generic types.

of KeY. The flexibility of KeY even allows components of the system to be applied to non-classical reasoning: the successful KeYmaera system by Platzer and Quesel [2008] for the verification of hybrid programs is a derivate of KeY.

As displayed in Fig. 7.1 on the preceding page, the KeY platform accommodates a range of techniques to prove or disprove that a program satisfies given properties: 1. interactive theorem proving, 2. counter example generation, 3. test case generation, and 4. visual debugging. KeY provides two graphical interfaces: 1. The stand-alone graphical user interface, that is displayed in Fig. 7.2, is intended for interactive proofs.[2] 2. The integration into the Eclipse platform is intended for code-level interactions and hides the underlying prover architecture [Hentschel et al., 2014].

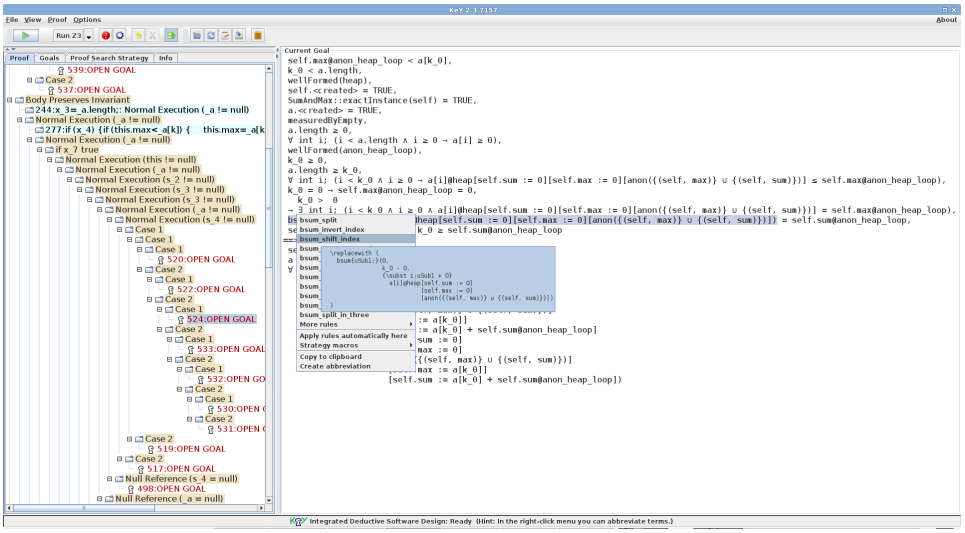


Figure 7.2: The main window of KeY's graphical user interface (GUI). On the left hand side, the proof tree shows open goals. The sequent belonging to the currently selected goal is shown on the right hand side. We selected a formula to show the applicable rules in a context menu, including a preview in the tool tip (the blueish overlay).

Secure information flow for sequential Java can be analyzed precisely with KeY as already mentioned in Sect. 2.4.3. Java programs along with a JML specification are translated to proof obligations of dynamic logic, using the approach in Sect. 6.2.5. This feature is already implemented in the stable 2.4 release of KeY. Implementation details are provided by Scheben [2014].

[2] However, proofs can often be found without interaction.

The KeY Prover

The common basis for all of this is a rule-based symbolic execution engine. In fact, symbolic execution is performed through applying proof rules for dynamic logic. Most rules are expressed using KeY's *taclet* language [Beckert et al., 2004; Rümmer, 2007]. Some rules—dealing with methods or loops in programs—are hard-coded, however. The base system uses c. 1,600 rules, of which a large part is used to reason about programs or arithmetic. In contrast to proof obligation generators, that produce usually very large FOL problems from programs/specification and try to discharge them using fully automated off-the-shelf theorem provers, all reasoning is completely integrated in the KeY prover. On the other hand, program annotations—like contracts or invariants—are not encoded as part of the proof obligation, but exist as background theories. This allows them to be entered during the proof process when needed. All rule applications (automatic or interactive) are recorded and saved as an explicit proof object. This allows the proof to be replayed for later inspection.

Dedicated strategies allow to apply rules automatically. Yet, in general, we are considering undecidable problems. Automated prover runs may time out without providing a positive or negative result. For this reason, KeY features a GUI allowing proof goals or intermediate proof steps to be investigated, rules to be applied interactively, or strategy settings to changed. See Fig. 7.2 for an interactive rule application. However, interactive application of single rules does not scale well to larger problems.

Instead, we employ *proof macros* to guide an automated strategy towards a particular goal. Macros have their own subset of rules, strategy options, and stop conditions. For instance, the macro "Finish Symbolic Execution" only applies rules as long as there is a program in the goal. The "Full Autopilot" macro performs symbolic execution first, then splits up all postconditions—something that is usually regarded bad in proofs—and finally tries to close all resulting goals within a given number of steps, but reverts rule application if a goal could not be closed automatically. This leaves the interactive user with the proof goals that are 'interesting' in some sense.

Additionally, KeY provides translation of problems to the SMT-LIB language Barrett et al. [2010]; Barrett and Tinelli [2014], allowing powerful SMT solvers to be invoked as a back end on goals without program modalities, i.e., goals of FOL plus arithmetic. SMT solvers can be used for both proving validity and constructing a counter example (in case of goals that are not valid). Supported SMT solvers are CVC3 [Barrett and Tinelli, 2007], CVC4 [Barrett et al., 2011], Simplify (which is part of ESC/Java2), Yices [Dutertre and de Moura, 2006], and Z3 [de Moura and Bjørner, 2008].

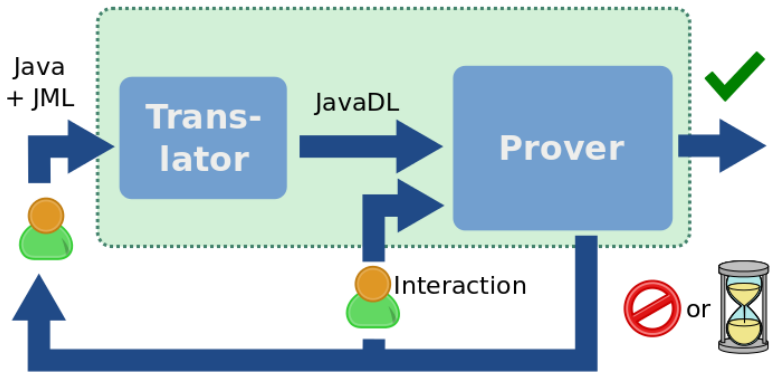


Figure 7.3: In the KeY approach, the user provides a set of Java source files that are specified with JML annotations. KeY parses these files and translates JML specification to JavaDL formulae. The prover may return with a negative result because the formula to prove is not valid or a timeout occurred. In this situation, the user can further interact with the prover or provide fresh Java/JML input. User interaction is possible at any point in time throughout the prover run.

Java Front-End

Atop the prover layer, KeY features a proof management system, that relates Java source files to proofs. The basic work flow is displayed in Fig. 7.3. The user provides a set of Java source files that are specified with JML annotations. KeY parses these files and translates JML specification to JavaDL formulae. After parsing, the system presents a list of proof obligations e.g., for functional correctness method contracts, in an intermediate representation for the user to select. These proof obligations relate to only a part of the Java input, typically to a single method.

The proof management system keeps track of the proof obligations that have been proven so far. Proofs can be 'conditionally closed': they contain rule applications of which the soundness relies on other, open, proofs. For instance, a user may add an assumption as a lemma, that can be proven separately. For the verification of non-trivial Java programs, a method call is usually abstracted through a *contract*.[3] The application of a method contract rule is only sound if the corresponding proof obligation is valid. See Sect. 8.2

[3] Alternatively, the implementation code can be inlined—but only if the concrete implementation is known.

on details regarding the concept of method contracts in modular program verification.

Deployment

KeY has been successfully applied to a large number of case studies [Bubel, 2007; Mostowski, 2007a,c; Schmitt and Tonin, 2007; Bormer et al., 2012; Filliâtre et al., 2012; Klebanov et al., 2011; Bruns et al., 2015b; de Gouw et al., 2014, 2015]. Third-party reviews of KeY and comparisons with other tools are provided by Stump [2005]; Feinerer and Salzer [2009]; Garavel and Graf [2013]; Saeed and Hamid [2015].

The KeY system is being developed by the *KeY project*, a joint effort between Karlsruhe Institute of Technology, Technical University of Darmstadt, and Chalmers University of Technology in Gothenburg, ongoing since 1999. Version 2.4 is the latest available stable release at the time of writing. KeY is free/libre/open source software (FLOSS) and can be downloaded in source or compiled form from `http://key-project.org/download/`. The Java Webstart technology allows KeY to be started directly from the browser, provided that a Java Runtime Environment (JRE) version 6 or later is installed.

7.2 Issues with Concurrent Java

In comparison to the simplified language dWRF that we have used in Chaps. 4 and 5, the actual Java language contains some additional caveats concerning concurrency. We discuss how we cater for these particularities in our KeY implementation—or why leave them unresolved.

Threads

In Java, threads are identified objects, more precisely, instances of class `Thread`. They have object identities and are subject to the usual object lifecycle. After creation, a thread first remains suspended. Each thread has an associated instance of the interface `Runnable`, called the thread's *target*,[4] that declares a method `run()`. It contains the code to be run, while the thread is identified with the instance of class `Thread`.

When the `Thread#start()` method is called, a fresh thread is created, that starts executing its target's `run()` method concurrently. In Java, the dynamic thread creation statement `fork {`π`};` from Chap. 4 corresponds to creating an instance of (an anonymous) `Runnable` and to start it as follows: `(new Thread(new Runnable(){run(){`π`}})).start();`. Threads may only be started once; an `IllegalThreadStateException` is raised if `start()` is called on an already running thread.

[4] `Thread` implements `Runnable`; so it is common that a thread's target is itself.

Thread execution may be interrupted through `Thread#interrupt()` or `Thread#yield()`. Stopping threads is still supported by Java, but considered inherently unsafe, and thus marked as deprecated in the API. For our implementation, we do not consider any of these operations.

It is a common pattern to create an anonymous subclass from the `Runnable` interface. However, anonymous classes cannot be specified with JML since there is no class declaration. This effectively prevents the possibility to formally verify such Java programs. In the following, we assume all classes to be named.

The Java Memory Model

The current Java memory model (JMM) was introduced with Java 1.5 [Manson et al., 2005]. It is a *weak* memory model. It does not assure *sequential consistency* [Lamport, 1979] of the memory per se. The sequential consistency property states that, at runtime, read and write actions appear in the order defined by the source code. The Java virtual machine (JVM) specification [Lindholm et al., 2014] allows implementations of the JVM to relax this order for efficiency reasons. In particular, this means that concurrent writes may not be immediately visible to other threads. JVM implementations are only required to ensure consistency for fields declared as `volatile` (see Java Language Specification (JLS), Sect. 8.3.1.4). Huisman and Petri [2007] provide a precise formalization of JMM taking into account so-called *happens-before* relations [Lamport, 1978] and values coming 'out of thin air.'

In general, sequential consistency is only assured if the threads do not compete in data races. Lochbihler [2014] provides a comprehensive formalization of Java, including JMM, using the Jinja formalization [Nipkow and von Oheimb, 1998] of Java in Isabelle. He provides a formal proof of sequential consistency being guaranteed under race freedom. An analysis for data race freedom can, for instance, be found in [Klebanov, 2009, Sect. 7.3].

For the more dedicated property of noninterference, there are definitions of stronger conditions, that imply noninterference for multiple classes of weak memory models [Mantel et al., 2014]. These are sufficient, but not necessary; i.e., they do not give rise a complete reasoning. Mantel et al. also present a program transformation technique that allows to construct programs that enforce a memory model independent operational semantics.

In this thesis, we are not further concerned with JMM. We assume sequential consistency without any further requirements. While this assumption is not necessarily sound, it is a sensible prerequisite for both devising denotational semantics based on interleavings as in Chap. 3 and effective reasoning. Assuming sequential consistency is the only sensible option in the context of full functional verification on the source code level without losing completeness.

Final Fields

Fields that are declared `final` may not be changed "under normal circumstances" [JLS, Sect. 17.5]. Final fields must be initialized or assigned to in the receiver's constructor. These two are the only two permitted modes of direct assignment. This effectively means that the field does never change after the receiver has been completely initialized.[5] Since only completely initialized objects are accessible to other threads, we may safely assume that final fields are never changed, effectively.

Synchronization

Synchronized blocks [JLS, Sect. 14.19] and the related concept of synchronized methods are the only means for synchronization directly supported by the language. A `synchronized` statement represents a block with a mutual exclusion lock on an object (cf. Sect. 5.6). Each object and class type is associated with a *monitor* that can be locked for exactly one thread. Only this thread may enter the block, all other are delayed until the monitor becomes unlocked again. Threads may lock monitors multiple times (given that it has not been locked by another thread); complete unlocking requires releasing the same number of locks again.

Be aware that a `synchronized` block is not the same as an atomic block in other languages. Even while a thread holds a lock on an object, other threads may write to fields of that object or call unsychronized methods on it. In fact, there is no support for atomic blocks in the Java language at all. Atomicity can only be achieved if all involved threads adhere to the mutual exclusion scheme.[6]

A thread executing `synchronized (a) {`π`}` first tries to obtain a lock on the monitor of the object represented by `a`. It is blocked while the monitor is locked by some other thread. It then executes the contents of the block π, and finally unlocks the monitor again. If the block terminates abruptly—i.e., either through a `break`, `continue`, or `return` statement; or the throw of an exception—the lock is released as well and the program execution continues as prescribed by the kind of abrupt termination. A `synchronized` method can be thought of the method body being surrounded by synchronization on `this` (for instance methods) or the class (for static methods).

[5] *Reflection* can be seen as a back door to assign final fields nonetheless, but here we ignore that possibility as usually in Java verification (cf. [Beckert et al., 2007b, Sect. 3.6.6]).

[6] The results by Abadi et al. [2006] show that even in Java's standard API implementation the synchronization regime is violated, leading to race conditions.

Nonatomic Compound Statements

The assignment statement `F = G;` where both `F` and `G` are fields is not atomic. It contains a read action first and then a write action. This common one-liner is functionally equivalent to the fragment `x = G; F = x;` using a local variable for intermediate storage, where both statements are atomic.[7] There can be an arbitrary number of read and write actions in any expression. For instance, the statement `F /= G++ * --H;` contains three read and write actions each. The final write to `F` does not occur if there is a division by zero. It can be rewritten into

```
x = F; y = G; G = y+1; z = H-1; H = z; w = x/(y*z); F = w;
```

The exact order of evaluation is crucial here. The operands are evaluated from left to right; if the evaluation has side effects, then these apply; finally the operation is applied. The division operation itself is also not atomic—even though it only operates on local variables—but it may include raising an exception, that entails further actions on the heap. The symbolic execution embedded in Java Dynamic Logic performs these kinds of transformations lazily and produces a normal form.

However, this not quite enough for reasoning about concurrent programs. A particular issue is that in Java, non-volatile fields of type `long` (64 bit integer) are not written in one atomic step, but in one for each 32 bit half (cf. JLS, Sect. 17.7). Writing to a `long` field `L` can thus be imagined of as applying two consecutive atomic writes:

```
L = x & 0xFFFFFFFF00000000L + L & 0x00000000FFFFFFFFL;
L = L & 0xFFFFFFFF00000000L + x & 0x00000000FFFFFFFFL;
```

Native Support for Atomic Operations

API implementations may provide additional methods with atomic compound operations. In the following, we investigate on the OpenJDK implementation as it is freely available under the GPL. For instance, the class `java.util.concurrent.atomic.AtomicInteger` encapsulates a 32 bit integer value that can be manipulated atomically. Listing 7.4 on the following page shows the implementation of method `getAndAdd()`, that adds an integer `delta` to the encapsulated value. It contains a loop, in which first the current encapsulated value is retrieved (Line 3). The implementation then tries to update this value atomically in Line 5. This update may fail—indicated through a returned `false` value—if the value has changed in between. In this case, the loop starts from the beginning. Otherwise, the method terminates with returning the value before the update (which may be different to the encapsulated value in the prestate).

[7] Except for the case where we have 64 bit data types, see below.

```
1 public final int getAndAdd(int delta) {
2     for (;;) {
3         int current = get();
4         int next = current + delta;
5         if (compareAndSet(current, next))
6             return current;
7     }
8 }
```

Listing 7.4: The OpenJDK implementation of the `getAndAdd()` method in class `AtomicInteger` provides lock-free atomic addition

This implementation is lock-free [Massalin and Pu, 1991], which means that it will eventually terminate successfully under the condition that the encapsulated value is not changed perpetually (and the scheduler returns control to the executing thread). The implementation relies on the *compare-and-swap (CAS)* primitive operation [Herlihy, 1991], implemented in Java as `compareAndSet()`.[8] CAS is a standard hardware operation implemented on most processors. It checks a memory address for some value and returns a failure if the address does not hold that value. Otherwise, it sets a replacement value. It constitutes the most basic atomic operation on many computer systems.

7.3 Extending KeY to Reason About Concurrent Java

The ultimate goal of constructing a usable verification system for concurrent Java involves many facets. In this thesis, we focus on the aspect of concurrency, while KeY is already a mature and usable verification system for (a substantial subset of) Java—accompanied by a considerable collection of proofs supporting this claim. Not to invalidate previous results must be the prime maxim, thus the implementation should be minimally invasive. This means in particular that the signature of JavaDL must not be changed. This is necessary to keep the implementation effectively maintainable. The development of the KeY system is a large project with several developers implementing new features in parallel. Besides adding concurrency, we do not touch KeY's coverage of Java, i.e., we still target Java 1.4 plus some extensions.

[8] In the OpenJDK implementation, `compareAndSet()` calls the native method `sun.misc.Unsafe#compareAndSwapInt()`, that provides the actual functionality. However, the class `Unsafe` is implementation-dependent, while the `compareAndSet()` method is supposed to be present in any API implementation.

Even though the original DTL calculus of [Beckert and Bruns, 2013] has been implemented on the basis of an earlier development version of KeY (see Sect 4.6), it has proved not to be very maintainable. The reason is that it introduced several changes to the core of the system. In particular, the invariant rule is hard coded in KeY. Of the invariant rules R28–R30, none is currently implemented since it is difficult to keep them in the same infrastructure as the invariant rule form standard JavaDL. Given this context, we have decided to discontinue the implementation described by Beckert and Bruns [2013] and to develop the new prototype closer to the main development branch of KeY. While this approach may increase the effort for a prototype; in the long run, it increases the chances that the change will eventually find its way into a stable release version.

In the following, we describe how trace properties can be expressed without the need to introduce the trace modality or temporal operators of DTL. We develop an experimental version of KeY on the basis of the upcoming release 2.6. A prototype implementation[9] can be downloaded from `http://formal.iti.kit.edu/~grahl/keyrg/`. Running the development version requires JRE 7 or later. It is best to use the Java Webstart version. It runs with one click from the browser and immediately opens the *example browser*, which allows to load some well-documented examples conveniently—including adaptations of the examples from Sect. 5.5 to Java. Not every feature mentioned in this dissertation is implemented at the moment of writing. Please refer to the change log on the website for an up-to-date status.

7.3.1 Trace Properties in JavaDL

Trace properties can be expressed in the 'diamond' modality of standard JavaDL with changes to the symbolic execution rules. It is well known that temporal formulae can be expressed in first order logic with arithmetic. The states need to be encoded and indexed. In general, this can result in complex formulas, that are infeasible to reason about in practice.[10] Here in this particular situation, we are only interested in the global program state induced by write actions of the thread under investigation. This is already modeled in our system using the special variable `heap`.

The proposed solution is that—in addition to `heap`, representing the current heap—we add a second special variable `heaps` of type $\mathbb{S}$. It represents the (finite) trace of heaps up to the current state of execution, starting from the initial heap. All symbolic execution rules that include changes to the heap in their premisses would have to be changed to not only update the `heap` variable, but also to append it to the trace. Currently, we are only considering assignments; thus excluding method contracts, loop invariants,

[9] Most of the actual programing has been carried out by Michael Kirsten.

[10] For instance, in the proof of Lemma 4.21 on page 77 we use such an encoding.

etc. We also introduce a variable `eStep` to distinguish environment steps from 'own' steps, as discussed in Sect. 3.4. Different to the variable `estp` from above, `eStep` is also of type $\mathbb{S}$, because we cannot represent temporal changes, but only store values in a sequence. The sequences represented by `heaps` and `eStep` are meant to have equal lengths.

We introduce a new symbolic execution rule for global assignment (i.e., heap storage). Still using the global variable syntax of Chap. 4, but the standard JavaDL $\langle\cdot\rangle$ modality ('diamond'), the new assignment rule would resemble the following. We translate this abstract notion into the concrete taclet language syntax of KeY below in Sect. 7.3.4.

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{\texttt{eStep} := \texttt{eStep} \oplus \langle \mathit{false}\rangle\}\{\texttt{heap} := \mathit{store}(\texttt{heap}, X, v)\} \quad \{\texttt{heaps} := \texttt{heaps} \oplus \langle \texttt{heap}\rangle\}\,\langle\omega\rangle\,\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\,\langle \texttt{X = v;}\ \omega\rangle\,\varphi, \Delta}$$

This is the only change to the rule base since it is the only rule that includes a local step in the trace. Assuming that the variables `heaps` and `eStep` did not appear in formulas before, this extension is also semantically *conservative*, i.e., formulas being universally valid before are still valid. Invariant rules are not changed. It is expected that invariants instead mention the variables `heaps` and `eStep`.

In place of temporal operators, we have quantifiers ranging over natural numbers and range restrictions. For example, let φ be a state formula that only contains the special variable `heap`, but no other (local) program variables. Then, the DTL formula $[\![\pi]\!]\Box\varphi$ can be (incompletely) represented by the pure JavaDL formula

$$\{\texttt{heaps} := \langle \texttt{heap}\rangle\}\ \langle\pi\rangle \qquad (7.1)$$
$$(\forall i : \mathbb{Z}.(0 \leq i < |\texttt{heaps}| \rightarrow \{\texttt{heap} := \texttt{heaps}[i]\}\varphi) \quad .$$

Likewise, the $\Diamond$ operator can be represented using existential quantification and combinations of temporal operators through iterated applications of this transformation. We assume that it is clear how to extend this schema to the general case.

Two-state formulae are similar: remember Formula 5.1, $[\![\pi]\!]\bullet\Box(\mathit{frame} \wedge \mathit{guar})$, where both *frame* and *guar* are formulae referring to the special program variables `heap` and `heap'`. A corresponding JavaDL formula that is an implicant of the above DTL formula is

$$\{\texttt{heaps} := \langle \texttt{heap}\rangle \mid\mid \texttt{eStep} := \langle \mathit{true}\rangle\}\ \langle\pi\rangle \qquad (7.2)$$
$$(\forall i : \mathbb{Z}.(0 < i < |\texttt{heaps}| \wedge \texttt{eStep}[i] \doteq \mathit{false}$$
$$\rightarrow \{\texttt{heap'} := \texttt{heaps}[i-1] \mid\mid \texttt{heap} := \texttt{heaps}[i]\}(\mathit{frame} \wedge \mathit{guar})) \quad .$$

This embedding of trace properties in JavaDL is significantly less invasive when compared to the DTL implementation, where a new modality was

introduced and all rules had to be adapted. Overall, this approach is sound, but incomplete.[11] The source for this incompleteness is that we require the program π to terminate, thus obtaining a finite trace. For instance, the DTL formula $[\![\texttt{while (true) \{...\}}]\!]\Box \mathit{true}$ is obviously valid, but it cannot be proven using the above construction. In fact, there is no sound and complete construction in this spirit in the presence of nonterminating programs.[12]

This would require to introduce complex invariant rules in the spirit of R28–R30 and to use infinite data structures (such as maps [Wallisch, 2014]) instead of sequences. This comes with a high implementation cost. From a practical perspective, it seems reasonable to restrict ourselves to terminating programs since most meaningful sequential programs terminate.

On the other hand, this construction may be not as efficient to reason about as the dedicated temporal operators of (the provable fragment of) DTL. In particular, eventualities in DTL can be checked on the fly, while here it requires quantifier instantiation. For practical considerations, it does not seem to pose a severe issue as the proof obligations always have the very special shape $\Box\varphi$ where φ is a state formula. That means that the transformed formula contains just one universal quantifier.

7.3.2 Changes to the Verifier Core

In sequential KeY, the execution context specifies the method and receiver scope of the current symbolic execution. We extend it to include 1. the method in whose scope code is executed, as identified through its name and the class in which it is implemented, 2. an object to which the `this` reference points to, 3. the thread class which executes the code, 4. and an appropriate instance of `Thread`. The syntactical form of an execution context is `source=m@C, this=o, threadClass=T, thread=t`.

The KeY system comes with a limited collection of Java API classes, that are known to exist in most Java implementations. Originally, these were the classes included in the JavaCard API. In the meantime, other widely used classes that are not part of JavaCard have been added, e.g., `String`. For a base support of Java concurrency, we have added the interface `java.lang.Runnable` and the implementing class `java.lang.Thread`. The latter always contains a public ghost field `target`, that refers to the target `Runnable` of a thread.

We introduce two more special program variables, variable `thread` of type `Thread` $\sqsubset \mathbb{O}$ (representing the current 'own' thread) and `threads` of type $\mathbb{S}$ (representing the current thread pool). We use the already implemented sequence type $\mathbb{S}$ instead of introducing another special purpose data type.

[11] More formally, given a change in assignment rules as given above, the JavaDL calculus is not a complete calculus for DTL.

[12] Replacing the 'diamond' program modality in Formula 7.2 by 'box,' i.e., dropping termination, would allow us prove the invalid DTL formula $[\![\texttt{while (true) \{...\}}]\!]\Box \mathit{false}$.

This creates another issue: sequences are not type-parametric. We have to ensure that for every reachable state, the elements of `threads` are of type `Thread`.

7.3.3 Contract Infrastructure and Proof Management

The additional artifacts that are added to KeY's contract infrastructure are thread specifications as introduced in Sect. 5.3. They are represented similar to contracts, but with the difference that contracts refer to methods and thread specifications refer to classes—more precisely, to subclasses of `Thread`. In the concrete implementation, we represent this through a class `ThreadSpecification` in package `de.uka.ilkd.key.speclang`, that implements the interface `DisplayableSpecificationElement`, which is also extended by the interface `Contract`.

These specification elements generate proof obligations. For thread specifications, we generate one proof obligation. It includes both the conditions that a thread meets its guarantee (cf. (5.3)) and that its rely condition is reflexive and transitive (cf. (5.4)). In the concrete implementation, this proof obligation is represented by the class `GuaranteePO`. The decision to combine both conditions into a single proof obligation is to be consistent with the present contract infrastructure in KeY, where any specification element yields exactly one proof obligation (cf. [Grahl and Ulbrich, 2016]).

As mentioned in Sect. 5.3, of the four conditions of a thread specification being valid, two are local to a thread while the other two are global (i.e., system-wide). Since our approach targets a modular analysis, proving global criteria makes little sense: for an entirely symbolic environment there would be no way to prove nontrivial thread specifications.

Instead, we treat these criteria similar to contracts or invariants in the established contract-based verification of sequential programs: at any point in time, we *assume* the thread specification of the system to be valid w.r.t. the currently alive threads. Only upon the dynamic creation of new threads, we prove that thread specification validity is *preserved*. In the modular verification of sequential programs, we assume the precondition at the beginning of a method and, for any method called in that context, we prove the precondition of the callee. However, it may still be that we assume an unsatisfiable precondition or thread specification. On the meta-level, this is justified because any client to the module in question must establish its specification.[13]

[13] While this approach is modular, a bootstrapping makes limited sense and it is usually not considered. If needed, one could think of the `main()` method of the Java project under investigation being verified with an assumption on a one-element thread pool. However, even this setting would not be fully appropriate in practice since the JVM usually creates several threads *before* `main()` is executed by one of them.

In Java new threads are forked through calling `Thread#start()`. Therefore, we plan to implement the 'fork' rule as a special case of the built-in method contract rule for `start()`.

KeY's proof management system [Roth, 2006] marks proofs as 'conditionally closed' if they are closed, but rely on rules which are only conditionally sound and this condition has not been proven yet. Method contracts are a typical case of that. For a proof with method contract rule applications to be considered closed, all used contracts must be proven as well. For thread specifications, we demand a similar mechanism. If a proof contains the symbolic execution rule for dispatching a new thread of type T, then the proof obligation must be proven for T. In contrast to method contracts, we explicitly allow circularities on the meta-level: if a thread of type T forks another thread of type T, then nothing is to be done.

7.3.4 Instantiating the Rules

The schematic rules as presented in Chaps. 4 and 5 are instantiated with the concrete syntax of the KeY taclet language [Beckert et al., 2004]. Depending on the target, there are multiple read or write rules. For full Java, we need to distinguish between instance fields, static fields, and array elements. The special `length` reference in arrays can be treated like a local variable since it is always read-only. This means that there is no anonymization before reading `length` in our rules. We assume the same for fields declared as `final` as discussed above. This means that we can keep the present symbolic execution read rules (i.e., for sequential Java) for `length` and all final fields. This requires a further distinction between final and nonfinal fields.

Write Action Rules as Taclets

Listing 7.5 on the following page shows a taclet rule for symbolically executing a global write action. It matches any program modality in which the active statement is an assignment of a simple expression to a nonstatic field with receiver `this`. This is expressed in the `find` clause (Lines 2f.) and the additional *variable conditions* (`varcond`, Line 4). The two dots represent an inactive program prefix, i.e., a sequence of opening braces, labels, etc. The three dots represent the remainder of the program. The rule can only be applied on the right hand side of the sequent; the sequent arrow is represented by `==>`. The `replacewith` clause in Lines 5ff. indicates that this is a rewrite rule, i.e., there are no formulae added to the premiss sequent.

In the premiss formula, the active statement in the modality is removed. There are three new updates in front of the modality: the first one (in Line 6) states that the current heap is updated such that the value `#se` is stored in field `#a`. This update appears as well in the version of the rule for sequential Java. The second one (Line 7) updates the trace variable `heaps`

such that the current heap (i.e., the heap updated as above) is appended. The third update (in parallel with the previous one) appends the constant `FALSE` to `eStep`, indicating a non-environment step. These two lines are the only ones that are different from the rule set for sequential Java. Finally, the `heuristics` clause (Line 10; named this way for historical reasons) assigns this rule to rule sets. The rule is almost identical to the rule of the same name in the calculus for sequential Java. The difference is the additional update in Lines 7f.

```
1  assignment_write_attribute_this {
2    \find (==> \modality{#allmodal}{.. #v.#a=#se; ...}
3               \endmodality(post))
4    \varcond(\not \static(#a), \isThisReference(#v))
5    \replacewith( ==>
6         {heap:=store(heap,#v,#memberPVToField(#a),#se)}
7         {heaps:=seqConcat(heaps,seqSingleton(heap))
8           || eStep:=seqConcat(eStep,seqSingleton(FALSE))}
9         \modality{#allmodal}{.. ...}\endmodality(post))
10   \heuristics(simplify_prog, simplify_prog_subset)
11   \displayname "assignmentThis"
12 };
```

Listing 7.5: Symbolic execution rule for write access in the taclet language

Rely Rules

To avoid extending the program language, environment actions are not made explicit here. Instead, the necessary anonymizations occur directly upon application of read or program termination rules.

This requires all targets of rule applications, i.e., active statements in program modalities, to be of the shape of the simple language introduced in Sect. 3.2. This means that expressions in control structures must be simple and there must only be one heap access (either read or write) per assignment statement. While Java is very rich in the variety of assignment statements (e.g., compound assignment operators like `*=` or pre/post-increment/decrement operators), the symbolic execution concept embedded in KeY performs the necessary normalization through (lazy) program transformations. The symbolic execution rules guarantee that assignments from heap locations to local variables have the strict shape that the expression on the right hand side of the assignment consists solely of the location. Fig. 7.6 on the next page shows a simplified example proof involving a complex compound statement in sequential Java. Of the approx. 1600 rules available in KeY, there are only 5 symbolic execution rules dealing with reads from the heap.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Longrightarrow \{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{\_a}, \texttt{f}, \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) \cdot \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}))\}\,[\,]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a} \parallel \texttt{x} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) \cdot \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) \parallel \texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{a}, \texttt{f}, \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) + 1)\}\,[\texttt{\_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a} \parallel \texttt{x\_2} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) \parallel \texttt{x\_3} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) \parallel \texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{a}, \texttt{f}, \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) + 1)\}\,[\texttt{x = x\_2 * x\_3; \_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a} \parallel \texttt{x\_2} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) \parallel \texttt{x\_3} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) \parallel \texttt{x\_4} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) + 1\}\,[\texttt{\_a.f = x\_4; x = x\_2 * x\_3; \_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a} \parallel \texttt{x\_2} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) \parallel \texttt{x\_3} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f})\}\,[\texttt{int x\_4 = \_a.f+1; \_a.f = x\_4; x = x\_2 * x\_3; \_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a} \parallel \texttt{x\_2} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f}) \parallel \texttt{x\_3} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f})\}\,[\texttt{\_a.f = \_a.f+1; x = x\_2 * x\_3; \_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a} \parallel \texttt{x\_2} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f})\}\,[\texttt{x\_3 = \_a.f; \_a.f = \_a.f+1; x = x\_2 * x\_3; \_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a} \parallel \texttt{x\_2} := \mathit{select}(\texttt{heap}, \texttt{a}, \texttt{f})\}\,[\texttt{int x\_3 = \_a.f++; x = x\_2 * x\_3; \_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a}\}\,[\texttt{int x\_2 = \_a.f; int x\_3 = \_a.f++; x = x\_2 * x\_3; \_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a}\}\,[\texttt{x = \_a.f * \_a.f++; \_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a}\}\,[\texttt{int x; x = \_a.f * \_a.f++; \_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a}\}\,[\texttt{int x = \_a.f * \_a.f++; \_a.f = x;}]\,\varphi}}{\Longrightarrow \{\texttt{\_a} := \texttt{a}\}\,[\texttt{\_a.f = \_a.f * \_a.f++;}]\,\varphi}}{\Longrightarrow [\texttt{A \_a = a; \_a.f = \_a.f * \_a.f++;}]\,\varphi}}{\Longrightarrow [\texttt{a.f *= a.f++;}]\,\varphi}$$

Figure 7.6: Example proof involving symbolic execution of a complex compound statement in sequential Java. For a simplified presentation, variable declarations and update simplification steps are omitted. `a` is a local variable of type `A` and `f` is a field of type `int` in `A`. The postcondition φ is not relevant here since we only show the part of the proof related to symbolic execution.

Of these rules exist different variations depending on the semantics of implicit runtime exceptions. KeY offers three possible semantics allow (i.e., strictly follow Java semantics and raise exceptions as defined in the JLS, which is sound and complete, but inefficient), ban (i.e., prove that exceptions cannot be raised, which is sound, but incomplete), and ignore (i.e., ignore the possibilities of implicit exceptions, which is neither sound nor complete, but most efficient). This is controlled through so-called *taclet options*. Neither rule is applicable when the JavaCard taclet option is active—obviously, since JavaCard does not feature concurrency. Of some of the heap-reading rules there exist variants for final fields that do not include anonymization (see Sect. 7.2 above).

All rules are valid for both 'box' and 'diamond' modalities, but only on the right hand side of the sequent, while some of the rules for sequential programs are applicable on both sides.

Generating Read Rules

For technical reasons, the read rules are not written in the taclet language directly, but are generated programmatically from the specification. As discussed above in Sect. 5.4.1, rely/guarantee specifications are not represented explicitly as formulae on the sequence, but are kept as background axioms. These background axioms cannot be imported schematically in the taclet language. Instead KeY generates taclets on the fly, inserting the concrete rely condition of the thread under investigation into a rule template. In the original calculus there are 6 different rules dealing with writing to the heap and one rule for program termination. Each of them serves as a template for a generated rely rule, resulting in 7 generated rules per thread type.

Rule Management

These changed rules effectively constitute a new calculus, while reusing many of the present calculus rules. The above added/changed rules are available through the taclet option concurrency:RG, while the original rules for sequential Java are still available through the taclet option concurrency:off.

For future versions of KeY, the concept of so-called *profiles* has been considered. The goal is to make the KeY system more modular and more adaptable to new target languages. Profiles refer to both the target language and the target property. For instance, there would be a profile for each of the four combinations functional/relational verification of sequential/concurrent Java. A profile determines a set of rules (both taclet and built-in) that form the calculus to achieve the task at hand. In the future, we plan to adopt the concept of profiles in favor over taclet options.

Soundness

The final calculus, adapted to JavaDL, ought to be sound. Platzer [2004] provides soundness proofs for a simplified dynamic logic, that is similar to the original JavaDL [Beckert, 2000]. A soundness proof for the method contract rule in JavaDL can be found in [Weiß, 2011, Appendix A.7]. As discussed by Beckert and Klebanov [2006], it is practically infeasible to *entirely* prove the soundness of a program verification calculus for a complex language like Java. A theoretical obstacle is the lack of an official formal semantics. The JLS still leaves some detail questions unanswered. While some formalizations of Java exist (cf. [Stärk et al., 2001; Nipkow and von Oheimb, 1998]), there is no guarantee that those actually represent the informal semantics of the JLS. A practical argument against a meta-verification of the approach is the labor it takes. Beckert and Klebanov argue that the effort is better spent on improving practical applicability and usability.

7.4 Reasoning About Information Flow in Concurrent Programs with KeY

Scheben and Schmitt [2012a] showed how to formalize state-based noninterference for sequential Java programs in JavaDL. In principle, their formalization is similar to the one that we have shown in Sect. 6.2, but instantiated concretely in JavaDL and implemented as a proof obligation in the KeY system. These proof obligations are referred to as "non-interference contracts." As explained in Sect. 6.3, the very same proof obligations can be used to refer to concurrent programs (at least for state-based noninterference).

We leave it as future work to extend upon this infrastructure to include proof obligations for trace-based noninterference. As explained above in Sect. 7.3.1, we are able to (incompletely) embed CDTL into JavaDL using the $\langle\cdot\rangle$ modality. Through a similar construction, this would also allow us to express strong noninterference in standard dynamic logic—restricted to programs that always terminate. The idea is to 'store' the heaps of two runs, filtered to those that are produced by 'own' steps, and to compare them component-wise.

8

Modular Specification of Concurrent Java Programs

As software systems grow bigger and get more complex, verification techniques have to account for that, too. In particular, the paradigm of object-orientation is targeted towards programing in the large, supporting reusability and extendability. For this reason, specifications and proofs thereof need to be modular and sufficiently abstract to cater for variability in the target software. Achieving complete functional verification of a complex piece of software—and the above goals in particular—still poses a grand challenge to current research (cf. [Leino, 1995; Hoare and Misra, 2005; Leavens et al., 2006a, 2007; Klebanov et al., 2011; Huisman et al., 2013, 2015]).

While this chapter is mostly concerned with *sequential* modularity, the concepts leading to modular analysis of sequential programs can well be applied to the concurrent scenario as well. Modular sequential analysis is concerned with interference that arises from method calls. It is well marked in the program code at what point in time during execution this interference occurs. Modular analysis of concurrent programs is also concerned with interference. However, this kind of interference is more difficult to assess or to harness. It may occur virtually anywhere in the program; and the severity of their effect lies within a vast range. Since the problem at hand is definitely harder, solutions for sequential modularity certainly cannot be solutions to concurrent modularity, but at least they can form a basis for further efforts.

The key to modularity is abstraction. The fundamental idea, to abstract away from concrete interference to a *contract* between a client and a provider, can be applied to both sequential and concurrent scenarios. For sequential programs, Design by Contract (DbC) [Meyer, 1992] is an established philosophy. A client is represented through a caller method and a provider through a callee method. This concept of method contracts is well supported in specification languages like the Java Modeling Language (JML). For concurrent programs, the fundamental idea manifests in the rely/guarantee methodology

(see Chap. 5). Here, the situation is more symmetrical: contracts are formed between all threads, without a particular hierarchy or order. But still, it is a game of providing a guarantee under the assumption that all others do as well.

Just as DbC is the cornerstone of programing in the large, analogously, it also enables *verification in the large*. As a result, we are able to define when a complex piece of software is not only a collection of individually verified components, but is considered entirely correct as a whole system. Given a concrete implementatation of sequential modules, we can still resort to a non-modular analysis, i.e., using method inlining. This is not so for concurrent programs. Here, abstraction is vital for an effective analysis (cf. [de Roever et al., 2001]).

It is only natural to combine techniques for the analysis of concurrent programs with those for object-oriented programs. Firstly, most object-oriented languages are also concurrent, e.g., Java. Secondly, as Jones [1996] argues, encapsulation techniques of object-oriented languages are particularly helpful to restrict the space of concurrent interleavings by design.

In modular verification, we may only have partial knowledge on changes applied to the heap. While the functional properties of contracts have been discussed before, meaning what an implementation must achieve, *framing* is roughly its dual: A method's frame specifies what implementations may *at most* do; and dependencies of state observers such as invariants or model fields define what information may be relied upon. We will come back to the explicit heap data type introduced in Chap. 4 and show how it can be used in modular verification. There is a long history of verification techniques that deal with the frame problem. A fairly new technique, called *dynamic frames* (see Sect. 8.2.3) aims at providing modular reasoning in the presence of abstraction that occurs in object oriented programs.

Chapter Overview

We introduce JML in Sect. 8.1 and provide a quick guide to basic JML specifications. A more thorough account is provided by Huisman, Ahrendt, Bruns, and Hentschel [2014], on which these sections are based. Parts of Sect. 8.2 are adapted from currently unpublished material by Grahl and Ulbrich [2016]; Grahl, Bubel, Mostowski, Ulbrich, and Weiß [2016][1] and provide a discussion on the specification and verification approach for modularized sequential Java, as part of the KeY approach [Beckert et al., 2007a]. After providing this overview of the state of the art, we present two extensions to JML. In Sect. 8.3, we introduce a new extension that caters for thread specifications following the rely/guarantee approach as introduced in Chap. 5. In Sect. 8.4, we review the extension by Scheben

[1]The material by Grahl, Bubel, Mostowski, Ulbrich, and Weiß [2016] is itself based the thesis by Weiß [2011]. Phrases that already appeared there verbatim are clearly marked.

[2014] for information flow contracts, that was developed in the Program-level Specification and Deductive Verification of Security Properties (DeduSec) project, as well as the more abstract Requirements for Information Flow Language (RIFL) [Ereth et al., 2014].

8.1 The Java Modeling Language

The Java Modeling Language (JML) is an increasingly popular and powerful specification language for Java software, that has been developed as a community effort since 1999. The main concepts are class invariants and method contracts. JML integrates seamlessly into Java as it is embedded inside comments in Java source code and JML expressions extend Java expressions in a natural way. By now, JML has become the de facto standard in formal specification of Java source code.

The nature of such a project entails that language details change, sometimes rapidly, over time and there is no ultimate reference for JML. Fortunately, for the items that we address in this introduction, the syntax and semantics are for the greatest part already settled by Leavens et al. [2013]. Basic design decisions have been described by Leavens, Baker, and Ruby [2006b],[2] who outline these three overall goals:

> - "JML must be able to document the interfaces and behavior of existing software, regardless of the analyses and design methods to create it. [...]
> - The notation used in JML should be readily understandable by Java programmers, including those with only standard mathematical training. [...]
> - The language must be capable of being given a rigorous formal semantics, and must also be amenable to tool support."

This essentially means two things to the specification language: Firstly, it needs to express properties about the special aspects of the Java language, e.g., inheritance, object initialization, or abrupt termination. Secondly, the specification language itself heavily relies on Java; its syntax extends Java's syntax and its semantics extend Java's semantics. The former makes it convenient to talk about such features in a natural way instead of defining auxiliary constructs or instrumenting the code as in other specification methodologies. The latter can also come in handy since, with a reasonable knowledge of Java, little theoretical background is needed in order to use JML. This has been one of the major aims in the design of JML. It however bears the problem that reasoning about specifications in a formal and abstract way

[2] This 2006b journal publication is a revised version of a technical report that first appeared in 1998.

becomes more difficult as even simple expressions are evaluated w.r.t. the complex semantics of Java.

Ever since, the community has worked on adopting a *single* JML language, with a *single* semantics—and this is still an ongoing process. Over the years, JML has become a very large language, containing many different specification constructs, some of which are only sensible in a single analysis technique. Because of the language being so large, not for all constructs the semantics is actually understood and agreed upon, and moreover all tools that support JML in fact only support a subset of it. There have been several suggestions of providing a formal semantics—including the author's own [Bruns, 2009]—but as of today, there is no final consensus.[3] Moreover, JML suffers from the lack of support for current Java versions; currently there are no specifications for Java 5 features, such as enums or generic types. Dedicated expressions to deal with enhanced foreach loops have been proposed by Cok [2008].

8.1.1 A Short Introduction to JML Specification

In this section, we provide a quick intuitive introduction to basic JML specifications. Huisman, Ahrendt, Bruns, and Hentschel [2014] present a longer introduction, including several examples. We essentially introduce the syntax. Semantics can only barely be touched here; we refer to Bruns [2009]; Grahl and Ulbrich [2016] for more thorough accounts. JML is designed for the specification of 1. methods, where JML specifies the effect of a single method invocation; 2. classes and interfaces, where JML specifies invariants of an object; and 3. code blocks (including loops), where JML serves as auxiliary annotation. JML specifications are written as special comments in the Java code, starting with `/*@` (block style) or `//@` (end of line style). The `@` symbol allows parsers to recognize that the comment contains a JML specification.

Expressions

Annotations are basically just Java expressions (of boolean type). This is done on purpose: JML extends Java's syntax; almost every side-effect-free Java expression (i.e., that does not modify the state and has no observable interaction with the outside world) is also a valid JML expression.

In addition, JML defines several specification-specific special constructs, to be used in expressions. These are prefixed with a backslash symbol (\) to distinguish them from regular Java expressions. The keywords `\result` and `\old` may appear in postconditions. The `\result` expression represents

[3]The Lorentz Center in Leiden hosted a workshop entitled "JML: Advancing Specification Language Methodologies" in March 2015, that was organized by M. Huisman, G. T. Leavens, W. Mostowski, and the author. The participants did agree on a joint endeavor to develop a common semantics. A follow-up workshop is anticipated for early 2016.

the value a method returns. JML allows to mark any expression e in a postcondition with `\old(`e`)`, which means that e is not evaluated in the current (post)state of the method, but in its prestate. In most cases, `\old(`e`)` is a subexpression of some bigger expression, and it is important to be aware that all parts of the expression not included in `\old(...)` construct are evaluated in the current (post)state.

The official language specification contains a few more of these. Besides Java's logical operators, such as conjunction `&`, disjunction `|`, and negation `!`, also other logical operators are allowed in JML specifications, e.g., implication `==>`, and logical equivalence `<==>`. Since expressions are not supposed to have side effects or terminate exceptionally, in JML in many cases the difference between logical operators such as `&` and `|`, and short circuit operators, such as `&&`, and `||` is not important. However, sometimes the short circuit operators have to be used to ensure an expression is welldefined [Kirsten, 2013]. For instance, `y != 0 & x/y == 5` may not be a welldefined expression, while `y != 0 && x/y == 5` is.

JML features quantified expressions of the following shapes:

- `(\forall` T x`;` a`;` b`)`
  'For all x of type T (excluding `null`) fulfilling a, b holds.'

- `(\exists` T x`;` a`;` b`)`
  'There exists an x of type T (excluding `null`) fulfilling a, such that b holds.'

Here, T is a Java (primitive or reference) type, x is any name (hereby declared to be of type T), and the range a and body b are boolean JML expressions. The range is optional, as `(\forall` T x`;` a`;` b`)` is equivalent to `(\forall` T x`;` a `==>` b`)` and `(\exists` T x`;` a`;` b`)` is equivalent to `(\exists` T x`;` a `&&` b`)`.

In addition to the boolean quantified expressions, JML offers so called *generalized quantifiers* `\sum`, `\product`, `\min`, `\max`, and `\num_of`, that can be seen as higher order functions with bound variables. The following expression represents the maximum over the elements of an `int` array:

```
(\max int i; 0 <= i && i < arr.length; arr[i]);
```

Notice that this is syntactically similar to a quantified formula: the `\max` operator binds a variable `i`, and a boolean guard expression restricts it to be within the range of the array's indices. The type of the `\max` expression is the type of its body; here it is the type of `arr[i]`. The intuitive semantics is obviously that the result is the maximum of all `arr[i]` where `i` is in the array range. However, the `\max` construct is not total, i.e., it is not always a welldefined expression. In case `arr` has zero length, for instance, there is no maximum. A similar case appears with a noncompact range, e.g., the

set of all mathematical integers (represented by the JML type `\bigint`): `(\max \bigint i; true; i)`.

Another comprehension operator is the summation operator `\sum`. Sum comprehensions in JML can have several bound variables that range over sets of values. The general pattern is `(\sum` T x`;` P`;` Q`)` where T is a type, P a boolean expression and Q an integer expression corresponds to $\sum_{x \in \{y \in T \mid P\}} Q$. Likewise the `\product` operator is used to express product comprehensions. Since addition (as multiplication) is commutative and associative, there is no particular order in which elements are summed up. Sums with empty ranges have the value 0 by definition, empty products have value 1.

Method Contracts

Contracts of methods are an agreement between the *caller* of the method and the *callee*, describing what guarantees they provide to each other; see also Sect. 8.2 below. More specifically, it describes what is expected from the code that calls the method, and it provides guarantees about what the method will actually do. While in our terminology, 'contract' refers to the complete behavioral specification, written JML specifications usually consist of *specification cases*. These specification cases are made up of several *clauses*.

The expectations on the caller are called the *preconditions* of the method. Typically, these will be conditions on the method's parameters, e.g., an argument should be a nonnull reference, but the precondition can also describe that the method should only be called when the object is in a particular state. In JML, each precondition is preceded by the keyword `requires`, and the conjunction of all requires clauses forms the method's precondition. A missing `requires` clause defaults to `true`.

The guarantees provided by the method are called the *postcondition* of the method. They describe how the object's state is changed by the method, or what the expected return value of the method is. A method only guarantees its postcondition to hold whenever it is called in a state that respects the precondition. If it is called in a state that does not satisfy the precondition, then no guarantee is made at all. In JML, every postcondition expression is preceded by the keyword `ensures`, and the conjunction of all ensures clauses forms the method's postcondition. A missing `ensures` clause defaults to `true`.

The `signals` and `signals_only` clauses specify *exceptional postconditions*. Exceptional postconditions have the form `signals (`E `e)` P, where E is a subtype of `Throwable`, and the following meaning: *if* the method terminates because of an exception that is an instance of type E, then the predicate P has to hold. The variable name `e` can be used to refer to the exception in the predicate. Note the implication direction: a `signals` clause does *not* specify under which condition an exception may occur by

itself, neither that it *must* occur. Such specification patterns can only be obtained in combination with `requires` and `ensures` clauses. A missing `signals` clause defaults to `true`. In addition, one can specify a method with `exceptional_behavior`. It expresses that a method *must* terminate with an exception (if it terminates at all).

The `signals_only` clause is optional in a method specification. Its syntax is `signals_only E1, E2, ..., En;` meaning that if the method terminates because of an exception, the dynamic type of the exception has to be a subclass of `E1`, `E2`, ..., or `En`. If `signals_only` is left out, only *unchecked exceptions*, i.e., instances of `Error` and `RuntimeException`, and the exception types declared in the method's `throws` clause are permitted.

Termination is specified with the `diverges` clause. It provides a necessary condition for nontermination, evaluated in the prestate. In contrast to the above clauses, the default for a missing `diverges` clause is `false`, meaning a method is supposed to terminate (normally or exceptionally). The semantics of a basic specification case—consisting of one of each `requires`, `ensures`, `signals`, and `diverges` clause—can be described as follows.[4] Consider the following specification case to be attached to a method:

```
/*@ requires α;
  @ ensures  β;
  @ signals (Throwable e) γ;
  @ diverges δ;
  @*/
```

It intuitively means that, under the condition that α holds in the prestate, 1. if the method terminates normally, then β holds in the poststate, 2. if it terminates through the throw of an exception, then γ holds in the poststate, and 3. if the method does not terminate, then δ holds in the prestate.

Further clauses include `assignable` and `accessible` clauses for frame specifications (for write and read effects, respectively). We will discuss framing in more detail below in Sect. 8.2.3. Finally, the `measured_by` clause provides a termination witness to recursive method implementations. Table 8.1 provides an overview over JML contract clauses.

Specification cases can be prefixed with the keyword `normal_behavior`. It states that, implicitly, the method has to terminate normally (if at all). Similarly, JML also has an `exceptional_behavior` method specification. This specifies that the method has to terminate, because of an exception. The general `behavior` specification may well contain a `signals` or `signals_only` clause, whereas a normal behavior specification may not contain these, and an exceptional behavior specification may not contain an `ensures` clause.

[4] Everything else can be considered as syntactical sugar for the basic case; see [Raghavan and Leavens, 2005; Grahl and Ulbrich, 2016].

Table 8.1: JML contract clauses

clause	meaning
`requires`	precondition
`ensures`	(normal) postcondition
`signals`	exceptional postcondition
`signals_only`	expected exceptions
`diverges`	requirement for nontermination
`assignable`	write frame
`accessible`	read frame
`measured_by`	termination witness

Class and Interface Specification

JML provides class level specifications, such as invariants, history constraints [Liskov and Wing, 1993], and initially clauses. These specify properties over the internal state of an object, and how the state can evolve during the object's lifetime. One of the most important and widely-used specification elements in object-orientation are type *invariants*. These can be seen as conditions to constrain the state an instance can be in. This means that any method, when invoked from a state in which the invariant holds, must reach a poststate (normally or exceptionally) in which the invariant holds again. In addition, any constructor has to ensure that the invariant is established. Although invariants are always specified within a class or interface, their effective scope is global. E.g., a method m in a class C is obliged to respect invariants of class D. Declaring a method `helper` avoids this obligation.

8.2 Modular Verification Using Contracts

The concept of modules in programing languages can be traced back to early examples such as Simula 67 [Nygaard and Dahl, 1981] or Modula [Wirth, 1977]; see also [Hoare, 1981]. Single modules (i.e., method implementations or classes containing them) may be added, removed, or changed with only minimal changes to their clients; programs can be reused or evolved in a reliable way. These ideas were put forth with the development of object-oriented programing: "The cornerstone of object-oriented technology is reuse." [Meyer, 1997] In object-oriented programing (OOP), methods (or procedures) consist of declarations and implementations. Declarations are visible to clients while implementations are hidden. One important addition in OOP to the base concept of modularity is that classes (i.e., modules) are meant to define types—and subclasses define subtypes. And, in particular, different classes may implement a method in different ways (*overriding*), including covariant and contravariant type refinement. A client never knows which

implementation is actually used. Any call to a (non-private) method is subject to *dynamic dispatch*, i.e., the appropriate implementation is chosen at runtime from the context. This concept is also known as *virtual* method invocation.

The concept of method contracts was first mentioned by Lamport [1983] and later popularized by Meyer [1992] under the name Design by Contract (DbC). It allows to abstract from those concrete implementations and to approximately predict module behavior statically.[5] The metaphor of a legal contract gives an intuition: A client (method caller) and a provider (method implementer) agree on a contract that states that, under given resources (preconditions), a product with certain properties (postconditions) is provided. This is a separation of duties; the provider can rely on the preconditions, otherwise they are free to do anything. Given the preconditions, they are only obliged to ensure the postconditions, no matter *how* they are established. On the other hand, the client is obliged to ensure the preconditions and can only assume a product to the given specifications. In the basic setup, a method contract just consists of such a pair of pre- and postcondition. As it has already been explained above, state of the art specification languages as JML feature contracts with several clauses (of which all can be seen as specialized, functional or non-functional pre- or postconditions).

Contracts do not only play an important role in software design, but also in verification. In verifying a method that calls another one, there are two possibilities to deal with that case. Either, the implementation can be inserted or a contract can be used. The former is intriguingly simple; this is what would happen in an actual execution. But it carries three disadvantages:

1. It transgresses the concept of information hiding.

2. The concrete implementation must be known. This is not always guaranteed in static verification techniques; in many cases there is only information on static types, but not runtime types.[6] Not even doing a case distinction over all possible types would work here, since we consider *open programs*, where there is only partial knowledge about the type hierarchy.

3. In the case of recursive implementations (with an unbounded recursion depth), inserting the same implementation again would let the proof run in circles.

This leaves contracts as the only choice to deal with method calls in most cases.

[5] Note that contracts give semantical properties about modules and are in some sense orthogonal to design documents such as class diagrams, that are mostly syntactical.

[6] A notable exception are `final` classes in Java, that may not have any subclasses. This means that an instance of a final class C also has runtime type C.

8.2.1 Behavioral Subtyping

In a completely modular context, the concrete method implementations generally are not known. Nevertheless, a client will assume that all implementations of a common public interface (i.e., a method declaration) behave in a uniform way. This concept is known as *behavioral subtyping*, *Liskov's substitution principle*, or the *Liskov-Leavens-Wing principle* [Liskov, 1988; Leavens, 1988; Liskov and Wing, 1993, 1994].[7][8] It can be formulated as follows: A type T' is a behavioral subtype of a type T if instances of T' can be used in any context where an instance of T is expected by an observer. In other words, behavioral "subtyping prevents surprising behavior" [Leavens, 1988]. Note that this notion of a 'type' is different to both types in logic (cf. Sect. 4.2) and types in Java (i.e., classes and interfaces).

Behavioral subtyping is a semantical property of implementations. Although the concept is tightly associated with design by contract, it cannot be statically enforced by programing languages. It is not uncommon to see—especially in undergraduate exercises—that subclasses in object-oriented programs are misused in a non-behavioral way. Imagine, for instance, a class `Rectangle` being implemented as a subclass of `Square` because it adds a length to `Square`'s width. This kind of data-centric reuse is a typical pattern for modular programing languages without inheritance. Not all rectangles are squares, so intuitively, this should not define a behavioral subtype. But whether it actually does, depends on the public interface (i.e., the possible observations). If the class signature of `Square` allows to set the width to a and to observe the area as a^2, then the subclass `Rectangle` is not a behavioral subtype.

For modular reasoning about programs, we may only assume contracts for a dynamically dispatched method that are associated with the receiver's static type, since the precise dynamic type depends on the context. This technique is known as *supertype abstraction* [Leavens and Weihl, 1995]. Behavioral subtyping is essential to sound supertype abstraction.[9] To (partially) enforce it, in the Java Modeling Language, method contracts are inherited to overriding implementations [Leavens and Dhara, 2000]. We can provide additional specifications in subclasses, which are conjoined with the inherited specification. This means, whatever the subclass specification states locally, it can only *refine* the superclass specification, effectively. This leads us to a slightly relaxed version of behavioral subtyping: instead of congruency w.r.t. *any* observable behavior, we restrict it to the *specified* behavior.[10] This

[7] Liskov and Wing themselves use the term "constraint rule."

[8] Despite first appearing in Leavens's thesis, it has been attributed to Liskov because of her widely influencial keynote talk at the OOPSLA conference 1988.

[9] Leavens and Naumann [2006] present a language-independent formalization of behavioral subtyping and prove that it is actually equivalent to supertype abstraction.

[10] Still, it is possible to explicitly specify all observable behavior.

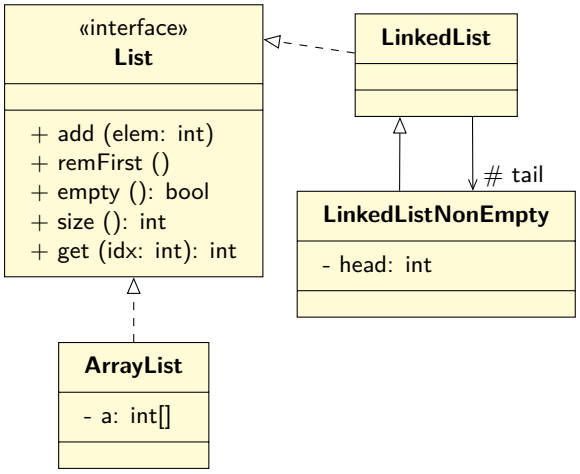


Figure 8.2: A list interface and its implementations

relaxation renders behavioral subtyping more feasible in practise, as it allows more freedom in implementing unspecified behavior, in particular regarding exceptional cases. Consider, for instance, a class that implements a collection of integers. Is a collection of non-negative integers a behavioral subtype?—The correct answer is 'maybe;' it depends on whether the operations that add members to the collection are sufficiently abstract to be implemented differently.

This notion of behavioral subtyping w.r.t. specified behavior also enables us to regard interfaces and abstract classes as behavioral supertypes of their implementations. While they do not provide a (complete) implementation themselves, they can be given a specification that is inherited to the implementing classes.

Example: Implementing a List

Consider we want to implement a list of integers in Java. It should support the following operations: (i) adding an element at the front, (ii) removing the first entry, (iii) indicating whether it is empty, (iv) returning its size, (v) retrieving an element at a given position (random access). A typical choice to implement a mutable list is using a linked list, where each entry is encapsulated in an object (usually called the *head* of the list) with a link to the remaining list (called *tail*).[11] Figure 8.2 shows a typical object-oriented design. An interface `List` provides the signature for some public methods. This interface is implemented twice: firstly simply as an `ArrayList` and secondly using the *composite* design pattern, by the classes `LinkedList` and

[11] Clearly, this is not the only possible way to implement a list. For instance, the elements could be saved in a more elaborate data structure (like a doublely linked list).

`LinkedListNonEmpty`. Note how the empty list is represented: through the so called *sentinel* pattern no `null` reference is exposed.

Before looking at an implementation, let us briefly discuss contracts in natural language. The operation 'removing the first element' only makes sense when there is at least one element—this would make a precondition. Similarly, 'retrieving an element at position n' only makes sense if n is non-negative and there at least n elements in the list. Again, implementations are free to do anything if they are called in a context where these preconditions do not hold. Listing 8.3 on the next page shows an implementation of class `LinkedList`. Here, we see two different styles of method implementations. In Lines 12ff., method `remFirst()` silently returns directly if it is called on an empty list, i.e., the precondition is violated. Alternatively, we could first check for such violations and then throw a more precise exception explicitly. This programing style is known as *defensive* implementation, where the implementing code checks for and handles abnormal situations. This is the style advocated by Meyer [1992].

In contrast, in Lines 25ff., method `get()` is implemented in an *offensive* manner. It does not check for abnormal situations, but optimistically calls the method `tail.get(idx)` where `tail` may be a `null` reference. In case the precondition is violated, an instance of `NullPointerException` will be thrown. This is the programing style advocated by Abrial [1996]. Design by contract itself does not advertise either style, but in practise the latter is usually preferred.

Another thing to notice about this implementation is the class hierarchy: most of the methods in `LinkedList` delegate to an element of the subclass `LinkedListNonEmpty`. The reason is that `LinkedListNonEmpty` is a linked list implementation that always contains at least one element, while `LinkedList` represents the supertype—a *possibly empty* list. This ensures that we have a behavioral subtype relation here. A non-empty linked list exposes at least the expected behavior of a possibly empty linked list. This allows for a maximum of reuse in class `LinkedListNonEmpty`, which is shown in Listing 8.4 on page 194; only three methods need to be overridden.[12] Note that the default constructor of `LinkedList` returns a (non-unique) empty list. Throughout the main part of this chapter, we will use the list example to explain how it can be specified in a modular fashion.

[12] An alternative common pattern for linked lists uses two classes `Nil` and `Cons`, where `Nil` is a singleton representing the empty list and `Cons` takes the same role as `LinkedListNonEmpty` in our example. This pattern can be used to implement *immutable* list objects. The disadvantage is that `Nil` and `Cons` are not (behavioral) subtypes of one or another.

```
1  public class LinkedList implements List {
2
3      protected LinkedListNonEmpty tail;
4
5      public void add (int elem) {
6          LinkedListNonEmpty tmp =
7              new LinkedListNonEmpty(elem);
8          tmp.tail = this.tail;
9          this.tail = tmp;
10     }
11
12     public void remFirst () {
13         if (empty()) return;
14         else tail = tail.tail;
15     }
16
17     public boolean empty () {
18         return tail == null;
19     }
20
21     public int size () {
22         return empty()? 0: tail.size();
23     }
24
25     public int get (int idx) {
26         return tail.get(idx);
27     }
28 }
```

Listing 8.3: An implementation to the `List` interface using a linked datastructure

```
1  class LinkedListNonEmpty extends LinkedList {
2
3      private int head;
4
5      LinkedListNonEmpty (int elem) { head = elem; }
6
7      public boolean empty () { return false; }
8
9      public int size () {
10         return 1+(tail==null? 0: tail.size());
11     }
12
13     public int get (int idx) {
14         if (idx == 0) return head;
15         else return tail.get(idx-1);
16     }
17 }
```

Listing 8.4: Non-empty lists is a behavioral subtype to lists.

8.2.2 Abstract Specification[13]

Even if programs are specified at the source code level, abstraction and modularization are indispensable for handling real world programs. In the interface `List` above, we have not yet given any contracts to the methods declared there. In fact, it is not possible to give implementation-independent specifications to all methods. Obviously, we could specify the behavior of `empty()` using `size()` as in Listing 8.5 on the next page—provided that we know that it is a pure method. Alternatively, we could give the following (recursive) postcondition to `size()`:

```
ensures \result == empty()? 0: get(0).size()+1;
```

Any way will lead us to a bootstrapping problem. In the concrete implementing classes, we can give contracts that refer to the internals, such as the private field `LinkedList#tail`, which can be declared as `spec_public` to refer to in specifications. However, this kind of implementation exposure trangresses the fundamental concepts of modularity. To solve this dilemma, we have to abstract away from the concrete Java program.

JML offers model fields [Leino and Nelson, 2002] (and the more advanced model methods) as specification-only representations of concrete implementation data. This enables implementation hiding: the requirement specification only refers to model fields while the abstraction relation is part of the (hidden) implementation details. For a detailed discussion of model fields, refer to

[13] This section is based on a preliminary and extended version of [Grahl et al., 2016].

```
1  public interface List {
2
3      //@ public invariant size() >= 0;
4
5      /*@ public normal_behavior
6        @ requires size() < Integer.MAX_VALUE;
7        @ ensures size() == \old(size()) + 1;
8        @ ensures get(size()-1) == elem;
9        @ ensures (\forall int i; 0 <= i && i < size()-1;
10       @                         get(i) == \old(get(i)));
11       @*/
12     public void add (int elem);
13
14     /*@ public normal_behavior
15       @ requires !empty();
16       @ ensures size() == \old(size()) - 1;
17       @ ensures (\forall int i; 0 <= i && i < size();
18       @                         get(i) == \old(get(i+1)));
19       @*/
20     public void remFirst ();
21
22     /*@ public normal_behavior
23       @ ensures \result == (size() == 0);
24       @*/
25     public /*@ pure @*/ boolean empty ();
26
27     /*@ public normal_behavior
28       @ ensures \result == size();
29       @*/
30     public /*@ pure @*/ int size ();
31
32     /*@ public normal_behavior
33       @ requires 0 <= idx && idx < size();
34       @ ensures \result == get(idx);
35       @*/
36     public /*@ pure @*/ int get (int idx);
37 }
```

Listing 8.5: Java interface `List` specified using pure methods

[Grahl et al., 2016]. Both concepts are deliberately still close to actual Java—not only because they syntactically resemble their respective counterparts, but also because in standard JML they can only be of a Java-defined type (either primitive or reference), but not a more structured abstract data type. Using model fields of reference type is problematic since we then conjecture that such an object exists (and is created) *in the current heap*. This means there still is a lack of abstraction.

JML additionally features abstract data types (ADTs) [Reynolds, 1975] at the language level. These include the type of finite sequences, referred to as type `\seq`, and the type of location sets, referred to as `\locSet`. These types are *primitive* in Java lingo, alike the other specification-only type `\bigint`. Reasoning about the underlying theory of finite sequences is well supported in KeY (cf. [Beckert et al., 2013b, Appendix A]). We will cover the `\seq` type below after a short general introduction to the concept of ADTs. The `\locSet` type will be introduced in Sect. 8.2.3, where we use it to express frame conditions. As already explained in Chaps. 3ff., both types play an important role in our approach to verification of *concurrent* programs.

Abstract Data Types

The category of ADTs has two interesting subcategories: the dual categories of *algebraic* and *coälgebraic* data types. In mathematics, the concept of an algebra has been known for a long time, while coälgebras appear in a lot of situations in computer science. A common example for an algebra are the natural numbers $\mathbb{N}$. The set $\mathbb{N}$ is defined as the smallest set such that $0 \in \mathbb{N}$ and, for all $n \in \mathbb{N}$, it is $s(n) \in \mathbb{N}$ where s is the successor function.[14] We say that natural numbers are *constructed* using the *constructors* $0 :\to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$, or: the data type 'natural numbers' is defined by construction. Having those constructors, we can define additional functions on $\mathbb{N}$ such as addition:[15] $(0 + n) := n$ and $(s(m) + n) := s(m + n)$.

In a similar way, we can construct the algebraic data type of (finite) sequences A^* over a set of members A. We have the constructors $nil :\to A^*$ for the empty sequence and $cons : A \times A^* \to A^*$ for appending. Again, we can define other functions on A^*, such the length of a sequence, $length(nil) := 0$ and $length(cons(a, s)) := 1 + length(s)$. In general, algebraic data types may have more than two constructors; imagine a data type of finite trees that has one constructor for each possible number of branches from a node (that makes an infinite number of constructors). However, all algebraic data types have at least a *unit constructor* (i.e., a constant) like 0 or *nil*.

[14] A well known formalization by von Neumann, based on set theory, defines $0 := \emptyset$ and $s(n) := n \cup \{n\}$, cf. [Levy, 1979].

[15] Strictly speaking, the set $\mathbb{N}$ is only the *carrier set*, of which and a category theoretical functor the actual algebra consists. For the purpose of this work, we allow us to identify algebras with their carrier sets.

We have already seen some additionally defined functions. Common to all of them is that they are defined *inductively*, i.e., their definition consists of a definition for each constructor. This kind of recursive definition is both *well-founded* and *total* due to the inductive nature of initial algebras, that entails that every element of the carrier set can be uniquely described using a finite number of constructor applications (i.e., construction is invertible).[16] As an example, in the `List` example above, we can model each state of the list using only these constructors. But this also allows to do proofs by induction. In order to prove the conjecture that for all sequences the length is non-negative, we only have to prove that the length of *nil* is non-negative and, for all $a \in A$ and all sequences s with a non-negative length, $\mathit{length}(\mathit{cons}(a, s))$ is non-negative; both of which follow directly from the definition of *length* above. Currently, KeY only supports induction over the natural numbers. However, this can be lifted to any discreet structure by indexing its members appropriately. For sequences, we perform induction over the length.

While algebras are defined through constructors, dually, *coälgebras* are defined through *observers*, also known as *destructors*. A common coälgebraic data type is the type A^∞ of infinite lists over a set of members A, that can be defined using the two observers $\mathit{head} : A^\infty \to A$ and $\mathit{tail} : A^\infty \to A^\infty$. As for algebras, we can define additional functions. For instance, a function *prepend* that adds a new element to the list has the following two definitions: $\mathit{head}(\mathit{prepend}(a, \ell)) := a$ and $\mathit{tail}(\mathit{prepend}(a, \ell)) := \ell$. This is now a *coïnductive* definition, that consists of several observations about one function value. Dual to the totality property in inductive definitions, this function is uniquely described through all observers. And, of course, there is a dual to proof by induction, that is co-induction through invariants: given that a list ℓ contains an element a, then also the list $\mathit{prepend}(b, \ell)$ contains an element a. Invariants are sometimes also called *copredicates*.

Actual coälgebraic data types mainly appear in functional programing.[17] But also in imperative programs, where all concrete data structures are finite, we can make use of coälgebraic specifications: They can describe invariants of data structures irregardless of how they are constructed. However, since these structures are finite, we need to add preconditions to when coälgebraic definitions are actually welldefined. For instance, in the `List` example, we would require the list size not to have reached the limit of Java's `int` type as a precondition for the addition operation. For further reading on coälgebras

[16] An example for an algebra that is not initial and therefore does not allow this principle of induction would be the natural numbers with addition as a third constructor, the element 5 ($= s(s(s(s(s(0)))))$) could then be represented as either $4 + 1$, $2 + 3$, $0 + 5$, etc.

[17] Coälgebraic data types are for instance used in the Dafny language [Leino and Moskal, 2013].

Table 8.6: Defined operations on the `\seq` ADT in JML

	syntax	signature
empty sequence	`\seq_empty`	$\to \mathbb{S}$
singleton sequence	`\seq_singleton(`e`)`	$T \to \mathbb{S}$
concatenation	`\seq_concat(`$s1$`, `$s2$`)`	$\mathbb{S}^2 \to \mathbb{S}$
subsequence	s`[`i`..`j`]`	$\mathbb{S} \times \mathbb{Z}^2 \to \mathbb{S}$
comprehension	`(\seq_def \bigint x; `i`; `j`; `t`)`	$\mathbb{Z}^2 \times T \to \mathbb{S}$
access	`(`T`)`s`[`i`]`	$\mathbb{S} \times \mathbb{Z} \to T$
length	s`.length`	$\mathbb{S} \to \mathbb{Z}$

and their application, we refer to, e.g., [Barwise and Moss, 1996; Jacobs and Rutten, 1997].

KeY's extension to JML introduces dedicated location set expressions. For some of them, a translation is straight-forward as they have been designed to correspond to respective predicates and functions in JavaDL and have the obvious meaning, e.g., `\intersect(s,t)`. But location set expressions also replace *reference set expressions* from standard JML. These are faithfully translated to terms in JavaDL, taking into account that JML only considers locations which belong to already *allocated* objects. Please note that the special keyword `\strictly_nothing` is not an expression in this sense, but can be used to form a nonstandard `assignable` clause.

The binary union operator is called `\set_union` in JML for technical reasons. In addition, JML also features a set comprehension operator `\infinite_union`, that binds a variable of any type and has a location set expression in the body. Optionally, a guard can be given.

The algebraic data type `\seq` of finite sequences is predefined in KeY's extension to JML, with the additional subsequence and random access operations as displayed in Tab. 8.6. These operators are directly translated to their counterparts in JavaDL. Beckert, Bruns, Klebanov, Scheben, Schmitt, and Ulbrich [2013b, Appendix A] present the underlying theory of finite sequences. In particular, we have a comprehension operator `\seq_def` where `(\seq_def \bigint x; `i`; `j`; `t`)` denotes the sequence $\langle t[x/i], \ldots, t[x/j] \rangle$. Please note that `\seq` is not a parametric type; its elements are not typed. For this reason, sequence access always needs to be preceeded by a type cast.[18]

The type `\seq` counts as a primitive type in the Java sense. This means that all operations are side effect free (i.e., corresponding to mathematical functions), instances do not need to be created, and can be compared using equality (`==`). This particularly allows us to quantify over all (infinitely many) sequences. Abstract data types therefore are to be distinguished from

[18] In JavaDL, the access function itself is type parametric. An access in JML (prefixed with a type cast) is translated to the appropriate typed access.

the concept of *model types* [Leavens et al., 2006b, Sect. 2.3] in standard JML. These model types—like `JMLObjectSequence`—still are Java reference types, that may be used in specifications.

Given these prerequisites, it is only natural to represent the concrete list implementation of Sect. 8.2.1 using the `\seq` ADT. Assume that `theList` refers to a `\seq` representation of the list. Then, we can describe the addition (prepending) of an element as a concatenation of a singleton sequence and the prestate sequence:

```
/*@ public normal_behavior
  @ ensures theList ==
  @     \seq_concat(\seq_singleton(elem),\old(theList));
  @*/
public void add (int elem);
```

Analogously, the removal method can be specified using subsequence operation. The remaining question is: what program entity exactly is `theList` in this example? One solution would be to use a ghost field, as explained in the following, or model fields, as explained by Grahl et al. [2016].

Ghost Variables and Fields

Sometimes the information needed in specifications is not provided by the source code itself. This additional knowledge can be modeled with *ghost variables* and ghost fields. A ghost variable in JML can be defined as a class/instance member or as a local variable. In both cases, it is declared as a normal Java variable, but inside a JML comment preceded by the keyword `ghost`. It is important to know that the used type might be a specification-only type, such as `\bigint` or `\seq`. The initial value of a ghost variable can be directly assigned during its declaration. Its value can be updated during method execution by a `set` statement. As explained in Sect. 5.6, ghost fields are essential to model locking and to record progress in the rely/guarantee approach to verification of concurrent programs.

Ghost variables extend the system state. In particular ghost fields live on the heap as other fields do. Verifying a program with code ghost is not strictly the same as verifying the original program. It is the task of the specifier to ensure that this extension is conservative, i.e., the concrete program state and the ghost state converge (cf. [Filliâtre et al., 2014]).

8.2.3 Dynamic Frames

For *modular* static verification, where the goal is to check the correctness of individual program parts locally—that is, without considering the program as a whole—the demands both on specifications and on the specification language itself are higher than for approaches working under a closed program assumption, e.g., for runtime checking.

The fundamental idea behind JML is to satisfy the additional demands of modular (sequential) verification, but the support for frame annotations in vanilla JML (i.e., `assignable` and `accessible` clauses) falls short of this goal. Schmitt, Ulbrich, and Weiß [2011]; Weiß [2011] present a solution to these issues with their extension to JML, based on the *dynamic frames* approach by Kassios [2006, 2011]. Dynamic frames is a flexible approach for framing in the presence of dynamic data structures and data abstraction.

As described in this section, the concept of frame annotation has been developed as an instrument for verification of sequential programs. However, in Chap. 5, we have found out that they are as well beneficial for concurrency verification. While in this dissertation, we use dynamic frames in rely/guarantee specifications, they fit into other approaches as well, as discussed in Sect. 10.3.2.

Consider our running example considering lists. In Listing 8.7, we add a simple client to the `List` class. A client object holds references to two list instances. Method `m()` adds an element to one of them. The question is how to prove the postcondition that states that the other list has not changed in size. We have to add the precondition that `a` and `b` do not *alias*, under which the postcondition could never be valid.

```
class Client {
    List a, b;

    //@ requires a != b;
    //@ ensures b.size() == \old(b.size());
    void m() { a.add(23); }
}
```

Listing 8.7: Client code using two instances of the `List` interface from Fig. 8.2

As we have seen above in Sect. 8.2.1, a correct implementation of `add()` must satisfy the postcondition that the passed element has been added to the list. This is an impartial description of the method's behavior. For our particular situation here, however, we aim for the property that `a.add()` does *not* do anything harmful to `b`—that, besides the given functional property, "nothing else changes" [Borgida et al., 1993]. Such a property is usually expressed as set of locations to which the method may write at most, called

the *frame* of the method and a set of locations on which the result of a query depends at most, called the *footprint*. The *dynamic frame* theory [Kassios, 2011] aims at solving the frame problem in the presence of data abstraction. As Weiß [2011, Sect. 3.2] explains,

> "the essence of the dynamic frames approach is to leverage the ubiquitous location sets [...] to first class citizens of the specification language: specification expressions are enabled to talk about such location sets directly. In particular, this allows us to explicitly specify that two such sets do not overlap [...]. This is an important property [for pointer-based programs], which is called the absence of *abstract aliasing* [Leino and Nelson, 2002; Kassios, 2006]."

Abstract aliasing is also known as *deep aliasing*. Its absence is the property that we needed in the specification of our example in Listing 8.7 on the preceding page. The knowledge that the location sets represented by `a.footprint` and `b.footprint` are disjoint allows us to conclude that the postcondition is actually satisfied. A *dynamic frame* is an abstract set of locations. It is 'dynamic' in the sense that the set of locations to which it evaluates depends on the current state. It can change during program execution.

Dynamic Frames in JML

Schmitt et al. [2011]; Weiß [2011] present an implementation of the dynamic frames approach in KeY, using an extension of JML, that includes high-level specification elements for location set expressions. The type `\locset` has already been briefly introduced above in Sect. 8.2.2, with the underlying theory mentioned in Sect. 3.3. Semantically, expressions of type `\locset` stand for sets of memory locations. These expressions are used to write `assignable` and `accessible` clauses. Weiß [2011, Sect. 3.3] defines the language extension in the following way:

> "The singleton set consisting of the (Java or ghost) field `f` of the reference expression o can be denoted [...] as `\singleton(`o`.f)`, and the singleton set consisting of the i-th component of the array reference a as `\singleton(`a`[`i`])`. Like in [vanilla] JML, the set consisting of a range of array components and the set consisting of all components of an array are written as a`[`i`..`j`]` and a`[*]`, and the set of all fields of an object is written as o`.*`."

In addition, the extension by Weiß features the following basic set operations on expressions of type `\locset`, with the standard mathematical meaning: the set intersection `\intersect`, the set difference `\set_minus`, the set union `\set_union`, the subset predicate `\subset`, and the disjointness predicate `\disjoint`.

8.3 Concurrent Thread Specification in JML

Given sufficient background in the previous sections, we now develop an extension to JML for rely/guarantee-style specifications, that is appropriate to specify the behavior of multi-threaded Java. Thread specifications consist of both functional specifications and frame specifications based on the dynamic frame approach. Our approach to deductive verification of concurrent programs based on the rely/guarantee approach [Jones, 1983] has been explained in detail in Chap. 5. In Chap. 7, we have sketched the fundamental ideas of an implementation in the KeY system. This includes a discussion on specific features and issues regarding multi-threaded Java (as opposed to the simpler dWRF) in Sect. 7.2. Integrating rely/guarantee specification into JML is vital to the implementation since JML constitutes the main specification frontend of KeY. Constructing the proof obligation formulae by hand would be infeasible.

As discussed in Sect. 7.2, in Java, threads are identified with instances of the class `java.lang.Thread`. The actual program to be executed by the thread is contained in an implementation of the `run()` method that is declared in the interface `java.lang.Runnable`. From this observation and our design decisions in Sect. 7.3, we draw two conclusions regarding specification: 1. It must be possible to refer to the thread that is running in the current context. To this end, we introduce the keyword `\me`, that refers to the 'own' thread—similar to `this` referring to the method receiver in the given context. 2. Thread specifications are realized as class-level specifications in subclasses of `Thread`. However, we expect thread specifications to delegate to the actual running code, can be referred to through the field `target`, while the runner can refer to the thread as `\me`. As a minor issue, this requires us to use properly named subclasses of `Thread` since anonymous classes cannot be annotated in JML.

Following the basic principles of JML, thread specifications are inherited in subclasses. This means that a thread implementation must satisfy all thread specifications imposed on superclasses. The more general question whether a 'Liskov principle for threads' based on rely/guarantee makes sense will be left to future work.

As usual in the KeY approach, we do not consider dedicated constructs for other API services. We expect a specifier to annotate API methods appropriately when needed.

Thread Specification

As mentioned above, a thread specification appears as a class-level specification element, like an invariant. It may consist of several clauses, that are preceded by the keyword `concurrent_behavior`. We use the five clauses `requires`, `relies_on`, `guarantees`, `assignable`, and `not_assigned`; that

directly correspond to the elements of a thread specification as introduced in Sect. 5.3. An example is shown in Listing 8.8, that corresponds to the example in Sect. 5.5.2.

```
class MyThread extends Thread {
    private int x;

    /*@ concurrent_behavior
      @ requires x >= 0;
      @ guarantees x >= 0 && x > \prev(x);
      @ relies_on x >= \prev(x);
      @ assignable x;
      @*/
}
```

Listing 8.8: JML thread specification

On the 'guarantee' side, we use the well known `assignable` clause and introduce a new `guarantees` clause. The `assignable` clause is standard JML; it is followed by a list of heap locations (global variables) and intuitively means that only those may be assigned *throughout* the execution, or equivalently, the value of all other locations must not be changed in any state reached throughout the execution. The described property is exactly 'strict modifies' as mentioned in Sect. 5.3. The `guarantees` clause is new. Here, `\prev` denotes the previous intermediate state on the trace. Thus, it describes a two state property on the trace. For missing clauses, the defaults are the set of all locations or *true*, respectively. This thread specification is a kind of class level specification, that may only be given to subtypes of `Thread` and is indicated by the keyword `concurrent_behavior`.

On the 'rely' side, the `relies_on` clause gives a functional rely condition through a boolean expression, that may use the new special operator `\prev`, which allows to refer to a previous state. It is similar to the `\old` operator, that refers to the prestate in postconditions of method contracts. It may use the `\prev` operator. A missing `relies_on` clause defaults to *true*. The `not_assigned` clause lists locations on which we can rely on not to be changed by the environment. A missing `not_assigned` clause defaults to the empty set.

To retain sequential modularity, thread specifications can also be given to method contracts, as displayed in the example in Listing 8.9 on the next page, which is an adaptation of the example from Sect. 5.5.1. This can be used to prove the postcondition of a method w.r.t. concurrency. Note, however, that we have not yet developed a concept for using rely/guarantee contracts in sequential composition.

Our previous work [Bruns, 2015a] contained an additional `competing` clause to specify the set of concurrently running thread types in a non-

```
/*@ normal_behavior
  @ relies_on true;
  @ not_assigned z;
  @ ensures z == \old(z)+1;
  @*/
public void inc() { this.z++; }
```

Listing 8.9: JML method contract with `relies_on` and `not_assigned` clauses

modular way. This is not necessary in the approach as any environment can be specified in its effect as described above. In particular, the empty environment satisfies `not_assigned \everything`, which is sufficient to prove that all properties that are valid for sequential executions are also valid in the concurrent setting.

Ghost Fields With Atomic Updates

As mentioned in Sect. 5.6, ghost fields play an important role in specifying synchronization and progress in concurrent programs. In general, ghost fields in JML extend the system state, but are semantically equivalent to regular Java fields (cf. Sect. 8.2.2). In Sect. 5.6, however, we have assumed that ghost fields can be accessed atomically for both reading and writing. This constitutes a novel kind of ghost fields, that we represent in our extension to JML through an additional modifier `atomic`. Since ghost fields extend the regular system state, they may introduce inconsistencies into a proof. Additional proof obligations will be necessary to ensure consistency of ghost state and 'regular' state.

8.4 Information Flow Specifiation

Scheben and Schmitt [2012a]; Scheben [2014] present an extension to JML that allows to specify confidentiality properties for Java methods, extending JML's established contract framework. This extension is already implemented in the 2.4 stable release of the KeY verification system. The approach—though unconventional—allows fine-grained, flexible, and compositional security specifications as well as a convenient way to declare multi-level security. It includes declassification and erasure policies.

These information flow specifications use a knowledge-based notion of low-equivalence, that does not explicitly involve security levels (of a security lattice; cf. Sect. 2.2). Scheben and Schmitt argue that it is unnatural for a developer to devise a security lattice a priori and to assign locations to its levels. Producing an appropriate lattice can be nontrivial. Knowledge-based

specification, on the other hand, is expected to be easier to derive from high-level requirements.

Similar to the general frame problem in object-oriented programs, security specifications need to refer to sets of locations that are determined only dynamically. This requires a flexible and expressive specification language. The present specification approach builds on the flexible approach to framing presented above in Sect. 8.2.3.

8.4.1 Actors and Views

The specification approach by Scheben and Schmitt [2012a] is motivated by the *actor* model by Poetzsch-Heffter et al. [2011]. Each actor in a system is associated with a *view* they have on the system. A view describes an upper bound for the information that an actor is permitted have. Technically, a view is an ordered set of *observations* of the system. An observation is a statement about the system, i.e., a logical term or formula. Typical atomic observations are heap locations or method parameters, but compound observations can be, e.g., '$a + b$' or '$a > 0$.' Secure information flow between observation x to y means that x is contained in every view in which y is contained. As observations can be arbitrary complex statements, as opposed to mere memory locations, this definition readily includes declassification policies.

Consider, for instance, a multi-user system with authentification. The actors are 1. an administrator and 2. an arbitrary number of regular users. Each user holds a view on their user name and password, while the administrator holds a view on all user names, but no passwords. The user names are shared between multiple views. For n users, we have the following views: $user_i = \langle name_i, pw_i \rangle$ for each $i \in [0, n)$ and $admin = \langle name_0, \ldots, name_{n-1} \rangle$. In this example, a security lattice is implicitly given through the lattice of intersections and unions of the views (that are subsets of the set of observations). However, in general, a view-based specification cannot be translated to a lattice notion directly since observations may not correspond to locations (cf. Scheben [2014, Sect. 4.2]).

8.4.2 Information Flow Specification in JML

Information flow is specified per method in a Java, extending the established concept of method contracts in JML. This allows to define views locally, without further knowledge of the complete program, finally enabling modular specification. Views are determined dynamically. Syntactically, they are represented by *observation expressions* [Scheben, 2014, Sect. 3.1]; see also the discussion in Sect. 6.5.7. This allows a flexible means of specification that caters for the special requirements of modular, object-oriented software, in particular dynamically allocated data structures. For instance, it allows

```
public class LogIn {
  private String[] user, pw;

  /*@ requires user.length == pw.length;
    @ determines \everything, \result, \exception
    @      \by \nothing
    @      \declassifies
    @        (\exists int i; 0 <= i && i < pw.length;
    @            user[i] == c_user && pw[i] == c_pw);
    @*/
  public boolean checkPw (String c_user, String c_pw)
  {...}
}
```

Listing 8.10: The class `LogIn` implements a password checker. The JML information flow specification consists of a precondition and a `determines` clause. It states that the result of the `checkPw` method (public output) does not depend on any secrets, except for—as stated in the `\declassifies` clause—the fact whether there is an entry in the password file that fits the input.

to specify security in the above example for any value of n (that represents the size of an array or the length of a linked list, etc.).

The information flow method specification by Scheben [2014] uses a novel kind of method contract clause, the `determines` clause, as the central construct.[19] Listing 8.10 displays an example specification, using the authentification example from above. Clauses basically are of the shape `determines` Obs_1 `\by` Obs_0`;` where Obs_0 and Obs_1 are observation expressions. These are simply written as a list of expressions, separated by commas; any sequence expressions are flattened. These expressions may use standard JML keywords, such as `\result` to refer to a method result. Likewise, the newly introduced keyword `\exception` refers to the exception being raised (or not) [Scheben, 2014, Sect. 8.2.3]. The empty view is represented by the keyword `\nothing`.

Observations expressions are evaluated in the poststate and prestate, respectively. Let V_i be the semantical view represented by observation expression Obs_i in some state. Intuitively, the above clause means that V_1 is completely determined by the information provided by V_0—or equivalently, that the values of the elements of V_1 must not depend on any information outside V_0.[20] See [Scheben, 2014, Def. 3] for a formal definition of noninter-

[19] The earlier [Scheben and Schmitt, 2012a] features a `separates` clause, also known as `respects`, that has slightly different semantics. However, both translate to the same kind of proof obligations.

[20] Remember that the elements of V_i are themselves syntactical expressions of the language, not values. For an even more intuitive understanding, observation expressions

ference w.r.t. observation expressions. Security policies involving multiple views—and thus multiple security levels—can be expressed through multiple `determines` clauses.

Typically, both prestate and poststate observation expressions are the same.[21] As a shorthand notation for this case, Obs_0 can be the keyword `\itself`. Scheben's specification approach features some syntical sugar to emulate traditional specification styles: `determines` clauses can be amended with subclauses headed by the keywords `\declassifies` and `\erases`, that are being followed by a list of expressions. This allows the specifier to follow the traditional lines of source/sink annotation, i.e., only to write locations in the main clause and declassify (or erase) certain expressions retroactively. The specification further includes standard `requires` clauses (precondition) to express conditional noninterference (cf. Sect. 6.2.4).

To analyze *object-oriented* programs, the `\new_objects` subclause specifies a list of expressions. These are meant to point to the freshly allocated objects in the poststate. This translates to the set N in (6.12), following the approach in Sect. 6.5.6.

To make verification more feasible, Scheben further extends other JML specification elements with `determines` clauses in the same spirit, namely loop invariants, block contracts [Wacker, 2012], and class invariants.

8.4.3 The Requirements for Information Flow Language

For many real world problems, a semantically precise information flow analysis is just too expensive in practice. These problems can be solved by more lightweight automated techniques. To facilitate coöperation between different techniques and tools, in particular within the Reliable Secure Software Systems (RS^3) project, Ereth, Mantel, and Perner [2014] proposed a common specification language called Requirements for Information Flow Language (RIFL). The main motivation is to provide an intuitively understandable, light-weight exchange format. RIFL specifies programs and possible flows on an abstract level. By design, there is no *common* formal semantics. The idea is that the intuitive security properties expressed in RIFL are translated in a language- and tool-dependent way. The specific adaptation for Java as target language has been co-developed by the author as part of the RS^3 project.

RIFL is split into language-independent and language-dependent modules. The generic, language-independent part allows to specify an information flow security policy w.r.t. abstract classes (called *categories*) of sources and sinks. Categories are assigned to hierarchical security levels (called *domains*). As of RIFL 1.0, only standard (Cohen-style) noninterference (see Sect. 6.2) is supported, but the specification may involve arbitrary lattices. Means for

can typically be thought of lists of locations, as in classical low-equivalence definitions (cf. Sect. 6.2).

[21] Their value in the prestate and poststate may still be different.

declassification or more refined notions of noninterference are to be added in a later release. Apart from secure information flow, also functional properties can be expressed in RIFL.

RIFL for Java

The language-dependent part of RIFL identifies the kinds of sources and sinks that exist in the language. For Java, sources are fields, the parameters of the method under investigation, and the return values of called methods. Conversely, sinks are fields, the return value of the method under investigation, and the parameters of called methods. It also possible to refine general policies to account for language-specific security requirements. This includes declaring default categories and domains, that are common to all programs. For Java, this could include the relaxation to object-sensitive noninterference as in Sect. 6.5 or to assign standard API methods to categories.

The upcoming 2.6 release of KeY provides experimental support for RIFL specifications, implemented by the author. In our approach, we do not generate proof obligations directly from RIFL specifications, instead we translate them to the JML extension for information flow, that has been introduced in Sect. 8.4. This intermediate step bears two advantages. Firstly, the resulting JML specification can be shared with other tools that understand JML (at least the functional part). Secondly, proofs typically require additional user input, such as loop invariants, that have to be added manually. As loop invariants are tool-specific specifications, they are not part of RIFL specifications.

In the current state, there are some limitations: 1. The parser does not accept the final syntax of RIFL 1.0, but only the preliminary syntax of RIFL alpha, defined in a previous proposal. 2. We assume that programs provide proper encapsulation, i.e., methods only access fields of `this`. 3. Only a two-point security lattice (per method) is supported. Since the security lattice is only implicit in our notion, categories are ignored in the translation.

9 Verification of an Electronic Voting System

Electronic voting (e-voting) systems that are used in public elections need to fulfil a broad range of strong requirements concerning both safety and security. Among these requirements are reliability, robustness, privacy of votes, coercion resistance and universal verifiability. Bugs in or manipulations of an e-voting system may have considerable influence on society—and thus the life of the humans living in a country—where such a system is used. Hence, e-voting systems are an obvious target for software verification.

In this chapter, we report on an implementation of such a system in Java and the formal verification of functional properties thereof on the source level. We prove these properties using the KeY verification system (see Chap. 7). Even though the actual components are clearly modularized, the challenge lies in the fact that we need to prove a highly nonlocal property: After all voters have cast their ballots, the server calculates the correct votes for each candidate w.r.t. the original ballots. This kind of trace property is difficult to prove with static techniques like verification and typically yields a large specification overhead. In the approach that we follow, we cater for that by first verifying a basic implementation of the system. The basic system is refined on later iterations, such that previously obtained proof artifacts are still valid.

9.1 Electronic Voting

Elections form a part of everyday life that has not (yet) been fully conquered by computerized systems. This is partly due to the relatively high effort—elections do not occur often—and partly due to little public trust in security. The public discussion of this issue—in Germany at least [Deutscher Bundestag, p. 101]—has revealed a high demand for secure systems and in turn

a projection of high costs to construct those, that lead to the introduction of electronic voting being suspended. Systems for electronic casting and tallying of votes that are in the field in other countries (e.g., the Netherlands (cf. [Jacobs and Pieters, 2009]), the USA) are known to expose severe vulnerabilities. Apart from vote casting, computers are actually used in other activities related to elections such as voter registration or seat allocation.

A general goal is that electronic voting is at least as secure as voting on paper ballots. This includes *confidentiality* of the individual ballots/ votes. In particular they must not be attributed to a voter. But there is also an *integrity* issue: the final election result must reproduce the original voter intention; no vote must be lost, none must be manipulated. In paper-based elections, this mostly depends on trust in the election authorities. In electronic voting, the idea is to issue a receipt to the voter, a so-called *audit trail*, for casting their vote. After the votes have been tallied, the voters can then check on a public bulletin board whether their vote has actually been counted. This is called *verifiability* of the vote. To achieve verifiability and confidentiality of individual votes at the same time appears to be contradictory. The proposed solution is cryptography—that allows trails to be readable only to the voter. Some electronic voting systems also try to rule out voter *coercion* (by threatening or bribe). The idea is that trail and bulletin board are of a shape such that an attacker cannot distinguish the vote even under the circumstance that the coerced voter is trying to reveal it. This way, electronic voting may be even more secure than voting using paper ballots.

Due to the nature of requiring highest security guarantees, electronic voting has been frequently designated as a natural target for verification, e.g., by Clarkson, Chong, and Myers [2008]; Cortier [2014].

Yet, *proving* security is only a necessary step in establishing electronic voting. Systems must gain the *trust* of the public in order to be a democratic instrument. An important practical aspect of elections is *fairness*. As argued by Bruns [2008], fairness requires a profound understanding of verifiability and confidentiality not only to security experts, but to any eligible voter—who may be a layperson to formal security analysis. This issue is usually not considered with the present, complex systems. It is known that this very complexity is a prime reason why e-voting is not accepted by the public [Deutscher Bundestag, Sect. 2.4].

9.2 System Setup

We consider parts of the electronic voting system sElect described by Küsters, Truderung, and Vogt [2011]; Küsters and Truderung [2014]; Bruns et al. [2015a], that was developed in the RS^3 priority program. In this system, a remote voter can cast one single vote for some candidate. This vote is sent

through a secure channel to a tallying server. The secure channel is used to guarantee that voter clients are properly identified and cannot cast their vote twice. The server only publishes a result—the sum of all votes for each candidate—once all voters have cast their vote.

9.2.1 Verification Approach

As described by Beckert, Bruns, Küsters, Scheben, Schmitt, and Truderung [2012b], the distant goal is to show that no confidential information (i.e., votes) are leaked to the public. Obviously, the result—a public information—does depend on confidential information. This is a desired situation. In order to allow this, the strong information flow property needs to be weakened, or parts of the confidential information need to be *declassified*. Beckert et al. describe how such a property can be formalized using Java Dynamic Logic and proven in the KeY verification system.

Declassification —in the sense that parts of the secret information are purposely released[1]—is essentially a functional property.[2] In an election, the public result is the sum of votes, that result from secret ballots. In general, this cannot be dealt with using lightweight static analyses, such as type systems or program dependency graphs, that are still predominant in the information flow analysis world. Instead, the problem demands semantically precise information flow analyses as provided by the direct formalization of noninterference in dynamic logic [Scheben and Schmitt, 2012a]. In fact, this functional verification can be decoupled from the verification of information flow properties. Here, we report on functional verification only.

There are two approaches to verify information flow properties in this system. The first one, described in [Scheben, 2014, Chap. 9], is based on dynamic logic formalization of noninterference and theorem proving in KeY as layed out by Scheben and Schmitt. Scheben's proof heavily relies on functional correctness established by the proofs described in Sect. 9.3.2 below. The accounts given in this chapter and by Scheben [2014, Chap. 9] are companions to each other as they both report on different facets of the same problem.

Another approach combines functional correctness proofs in KeY with lightweight static information flow analysis as proposed by [Küsters, Truderung, Beckert, Bruns, Graf, and Scheben, 2013]. The target program is transformed in such a way that there is no declassification of information. We then prove that this transformation preserves the original functional behavior. This is discussed in Sect. 9.3.3. It allows the static analyzer JOANA

[1] This understanding is opposed to other uses of the term 'declassification' that denote the release of *any* information under certain constraints.

[2] We restrict ourselves to *functional* declassification here. As mentioned in Sect. 6.4.4, the *temporal* dimension of declassification would also be of interest in the election scenario.

[Hammer, 2009; Graf et al., 2013]—which is sound, but incomplete—to report the absence of information flow.

Verification of Cryptographic Implementations

The system uses cryptography and other security mechanisms. From a functional point of view, cryptography is extremely complex and it seems largely infeasible to reason about it formally. In particular, the usual assumption in cryptography that an attacker's deductive power is polynomially bounded—this is called a Dolev/Yao attacker [Dolev and Yao, 1983]—can not be reasonably formalized. As a matter of fact, even encrypted transmission does leak information and therefore *strong secrecy* of votes—which can be expressed as noninterference—is not fulfilled: the messages sent over the network depend on the votes and could theoretically be decrypted by an adversary with unbounded computational power. As a consequence, information flow analysis techniques—like the ones presented in Sect. 2.4—would classify the sElect system insecure, although it is secure from a cryptographic point of view.

Küsters et al. [2011]; Küsters et al. [2012] proposed a solution to this problem: the authors showed that the real encryption of the system can be replaced by an implementation of *ideal encryption*. Ideal encryption completely decouples the sent message from the secret. Even an adversary with unbounded computational power cannot decrypt the message. The receiver can decrypt the message through some extra information sent over a secret channel which is not observable by adversaries. Küsters et al. showed that if—in the system with ideal encryption—votes do not interfere with the output to the public channel, then the system with real encryption guarantees privacy of votes. Hence, it is sufficient to analyze the system with ideal encryption.

9.2.2 System Overview

The implementation is not build on distributed software systems, but is rather a simulation of several components involved. The basic protocol works as follows: First, voters register their respective client (represented by a class `Voter` here) to the server, obtaining a unique identifier. Then, they can send their vote along with their identifier (once). Meanwhile, the server waits for a call to either receive one message (containing a voter identifier and a vote) or to close the election and post the result. In the former case, it fetches a message from the network. If the identifier is invalid (i.e., it does not belong to a registered voter) or the (uniquely identified) voter has already voted, it silently aborts the call. In any other case, the vote is counted for the respective candidate. In the latter case, the server first checks whether a

sufficient condition to close the election holds[3], and only then a result (i.e., the number of votes per candidate) is presented. This is illustrated in the sequence diagram in Fig. 9.1 on the next page.

This simplified representation hides many aspects essential to real systems. We assume both a working identification and that identities cannot be forged. We assume that the network does not leak any information about the ballot (i.e., voter identifier and vote). This is meant to be assured through means of cryptography. The network may leak—and probably will in practice—other information such as networking credentials. We do not need to assume that the network communication is lossless or must not produce spurious messages.

9.2.3 Verification of a Nonmodular Software System

The particular challenge in this case study is that we prove a highly nonlocal (both spatial and temporal) property: after the election is closed, the *original* votes of all voters who are marked as voted *in the server* are counted to the result. This property is spatially nonlocal since it refers to the server and all voters simultaneously. It is temporally nonlocal since it refers to a particular state. This is very much countering the idea of Design by Contract (DbC) [Meyer, 1992], where properties are local to method call (and return) events. Instead, we have a kind of a trace property, that needs to be proven for every run of the protocol.

To verify this in the implementation, runs of the protocol are simulated through Java code again. Then we can annotate the synthetic main method with the desired property. As we will see, a simulation in Java brings with it the whole 'clutter' of a real-world language, such as object identities, createdness, heap separation, etc. Many of the specification items intended for the main method need to 'tracked' through the program stack trace. This approach comes with some major disadvantages. Firstly, the resulting specifications are strongly specialized and probably cannot be reused. Secondly, it produces a high specification overhead and thus also a verification overhead. Finally, reasoning about Java programs is far more expensive than reasoning on an abstract level. For KeY, though performing symbolic execution is not a bottleneck, reasoning about heap allocated data definitely is.

Example 9.1. Consider the following problem: The entries of two integer vectors (of fixed length) are nonnegative, prove that the vector resulting from pairwise addition again contains only nonnegative entries. This is more or less obvious; and a formalization in first order logic can be proven in KeY in less than 100 rule applications, taking 0.1 s time on a standard desktop computer. Now we implement this in Java, using arrays as vector representation, as

[3] In the present implementation, this is when all voters have cast their votes.

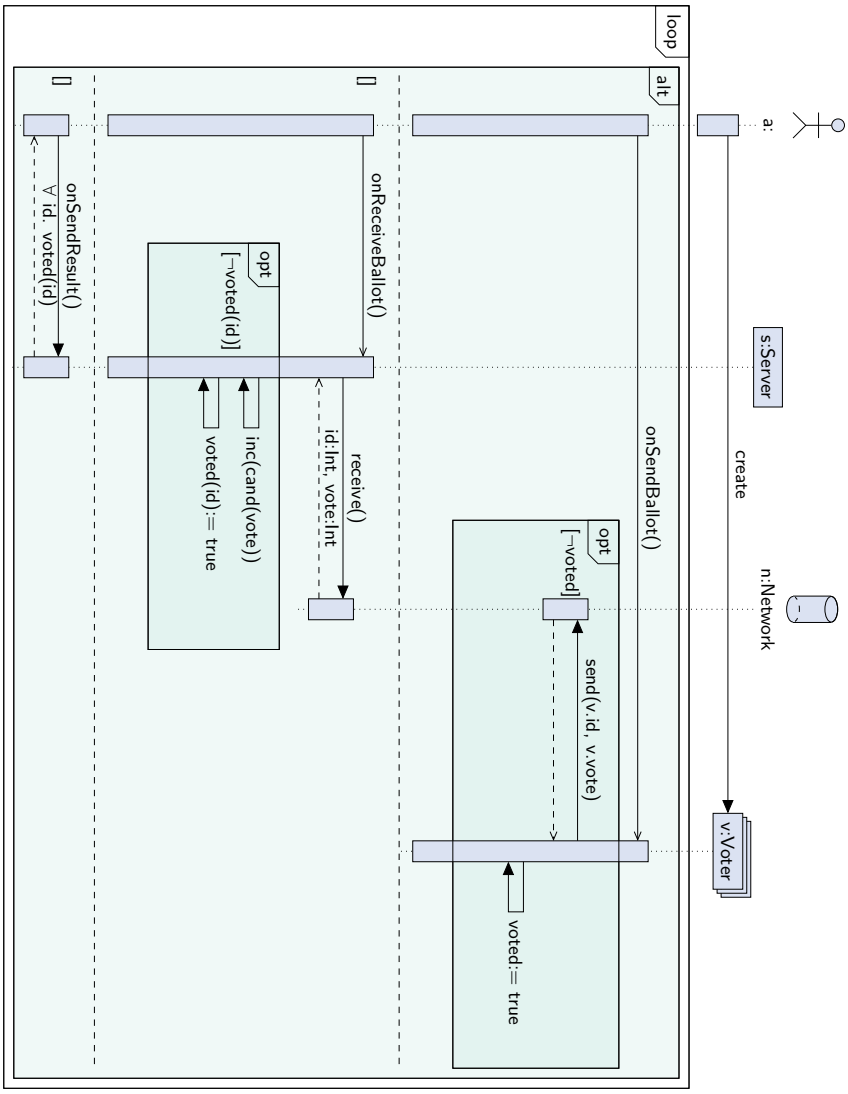


Figure 9.1: The overall protocol of the e-voting system. Here, the actor represent the indeterministic choice of events.

shown in Listing 9.2.[4] The first thing to notice is the outright specification overhead, including a loop invariant and frame annotations. The shown method can be proven correct w.r.t. the given specification automatically in KeY. But the proof size is considerably larger than for the FOL version. It now takes over 6500 rule applications and 11.5 s time.

```
1  class VectorAdd {
2      int[] a, b;
3
4      /*@ requires a.length == b.length;
5        @ requires (\forall int j; 0 <= j && j < a.length;
6        @                a[j] >= 0 && b[j] >= 0);
7        @ ensures (\forall int j; 0 <= j && j < a.length;
8        @              \result[j] >= 0);
9        @ ensures \fresh(\result);
10       @ pure
11       @*/
12     int[] add() {
13         int[] c = new int[a.length];
14         /*@ maintaining 0 <= i && i <= a.length;
15           @ maintaining (\forall int j; 0 <= j && j < i;
16           @                  c[j] == a[j] + b[j]);
17           @ maintaining \fresh(c);
18           @ assignable c[*];
19           @ decreasing a.length-i;
20           @*/
21         for (int i=0; i<a.length; i++)
22             c[i] = a[i]+b[i];
23         return c;
24     }
25 }
```

Listing 9.2: A simple Java program implementing vector addition

Our approach to tackle the complexity of the system is to verify a heavily reduced version first, then to refine it stepwise. This way, only smaller components are changed and the modular verification paradigm enshrined in the KeY system allows us to reuse many of the already obtained proofs while we only verify the changed components. The versions of the system produced in this way were developed by ourselves, in contrast to the actual sElect system implemented by Küsters et al.

[4] We added (weak) purity and freshness of the result to the postcondition to make the method 'more functional.'

9.3 Implementations and Verification

As described above, the goal is to verify a simple implementation of a distributed e-voting system. The design is based on the sElect system developed by Küsters et al., but reduced to its essential functionality. We start with a very basic version and incrementally add functionality or modeling aspects. Each step includes formal specification in JML and a full functional verification using a development version of KeY (pre-2.2).

For some of the proofs, we have given figures on the number of proof steps and the computation time for automated rule application. Automated rule application in KeY is supposed to be deterministic. Therefore, given the same version of KeY and the same options, the figure for proof steps should be verifiable in new experiments. Time measurements have been taken on a standard laptop computer (1 processor core, 1.5 GHz, 4 GiB of RAM, Debian/Linux). Another version of KeY, in particular the 2.2 release, may yield other figures. Please note that it is difficult to give figures for manual proofs. Firstly, the human interaction is necessary and therefore cannot be compared against computation time. Secondly, the time for the remaining automated rule application is not reliable as it may include time for rules applied automatically, but reverted by the user.

In any implementation, there is a class `Setup` that contains the main loop, that contains all global actions. The overall functional property to prove is that—after all votes have been cast (and collected by the server)—the server posts the correct number of votes per candidate. More precisely, the 'correct number' corresponds to the sum of votes for each candidate as on the ballots filled in by the voters.

9.3.1 Basic System

In the basic implementation, there are classes `Server` and `Voter` (clients) as well as a `Message` encapsulation class. Voters cast their votes in the order in which they are defined (and exactly once). Messages are passed directly to the server (through a method call).

The main method along with JML specifications is shown in Listing 9.3 on the next page. In the preconditions, we assume that no voter has cast their vote yet (or more precisely, the server has not yet marked the vote as cast) and all candidates have zero votes (in the server). The postcondition states that the number of votes for each candidate is exactly the number of voters who voted for them. This is expressed using the generalized quantifier `\num_of` (see Sect. 8.1.1). The explicit `diverges` clause allows this method to not terminate.[5]

[5]The actual sElect implementation does terminate, but since the implementations shown below do not terminate, we only require partial correctness for the sake of consistent properties.

```
1  /*@ normal_behavior
2    @ requires (\forall int j;
3    @              0 <= j && j < numberOfVoters;
4    @              !server.ballotCast[j]);
5    @ requires (\forall int i;
6    @              0 <= i && i < numberOfCandidates;
7    @              server.votesForCandidates[i]==0);
8    @ ensures (\forall int i;
9    @              0 <= i && i < numberOfCandidates;
10   @              server.votesForCandidates[i] ==
11   @                  (\num_of int j;
12   @                      0 <= j && j < voters.length;
13   @                      \old(voters[j].vote) == i));
14   @ diverges true;
15   @*/
16 public void main () {
17     /*@ maintaining \invariant_for(this);
18       @ maintaining 0 <= k && k <= numberOfVoters;
19       @ maintaining (\forall int j;
20       @                  0 <= j && j < numberOfVoters;
21       @                  j < k <==> server.ballotCast[j]);
22       @ maintaining (\forall int i;
23       @                  0 <= i && i < numberOfCandidates;
24       @                  server.votesForCandidates[i] ==
25       @                  (\num_of int j;
26       @                      0 <= j && j < k;
27       @                      voters[j].vote == i));
28       @ assignable server.ballotCast[*],
29       @            server.votesForCandidates[*];
30       @*/
31     for (int k= 0; k < voters.length; k++) {
32         server.onCollectBallot(voters[k].onSendBallot());
33     }
34     server.onSendResult();
35 }
```

Listing 9.3: The main loop in the basic setup

Since the loop is based on simple linear traversal of an array, the invariant is essentially an abstraction from the contract. The server entries for ballots cast and votes for candidates are the only changed locations here.

In this version, there are 4 methods to be verified with a total of 18 single lines of (executable) code and approximately 80 lines of specification.[6] The specification includes class invariants, method contracts, and loop invariants. Given our overall experience in formal specification, a 1:4 ratio of code against specification seems reasonable. As a positive result, the implementation can be verified without further user interaction. The downside is the proof size for the main method. It contains over 27,000 proof steps and took 210 s of computation time.

9.3.2 Adding a Network Component

To model a more realistic system, in the second implementation, we allow the adversary to decide on the order of events (i.e., voter submits a ballot, server collects a ballot, election ends). We now have an explicit modeling of a network component, through which messages are sent. However, the implementation is based on synchronous communication as the server immediately fetches a message that has been sent. This is the version on which Scheben [2014] reports.

The main loop is changed such that the order in that voters cast their votes is decided by the environment (low input). We have an additional class `Environment` that models all global sources and sinks. The untrusted input from the environment needs to be sanitized, but still the main loop may not terminate and voters are requested to cast their votes for an arbitrary number of times. The classes `NetworkClient` and `SMT` (for 'secure message transfer') model the network component. In the implementation, they mainly encapsulate a single message. Except for these additional classes, the size of the system is about the same as above in Sect. 9.3.1.

```
public void main () {
    while ( !server.resultReady() ) { // possibly infinite loop
        // let adversary decide send order
        final int k = Environment.untrustedInput(voters.length);
        final Voter v = voters[k];
        v.onSendBallot(server);
    }
    publishResult();
}
```

Listing 9.4: Implementation of the `main` method.

[6] Since there is no canonical representation, specification cannot be quantified objectively.

```
public interface Environment {
    //@ public static ghost \seq envState;

    //@ public static model \locset rep;
    //@ public static represents rep = \locset(envState);
    //@ accessible rep : \locset(envState);

    /*@ normal_behavior
      @ ensures     true;
      @ assignable  rep;
      @ determines  Environment.envState, \result
      @             \by Environment.envState;
      @*/
    //@ helper
    public static int untrustedInput();

    /*@ normal_behavior
      @ ensures     true;
      @ assignable  rep;
      @ determines  Environment.envState
      @             \by Environment.envState, x;
      @*/
    //@ helper
    public static void untrustedOutput(int x);

    /*@ normal_behavior
      @ ensures     0 <= \result && \result < x;
      @ assignable  rep;
      @ determines  Environment.envState, \result
      @             \by Environment.envState, x;
      @*/
    //@ helper
    public static int untrustedInput(int x);

}
```

Listing 9.5: Declaration of the interface `Environment`.

Listing 9.4 shows the implementation of `Setup#main()`. The adversary decides in the loop which client should send its vote next, until the server signals that the result of the election is ready. More precisely, the adversary is modeled through a call to the method `Environment.untrustedInput()`, that decides which client should send its vote. When subsequently the method `onSendBallot()` is called on the corresponding `Voter` object, the client sends its secret vote (stored in the attribute `vote`) to the server (synchronously), with the help of an implementation of ideal encryption. In its `onCollectBallot()` method, the server immediately counts the vote—provided that the voter did not vote before. Finally, the server is asked by a call to the method `resultReady()` whether the result of the election is ready. If so, the loop terminates and the result is published by a call to the method `publishResult()`.

We have a class `Environment` that models all global sources and sinks (see Listing 9.5 on the preceding page). The state of the environment is abstracted by a (ghost) field of type sequence. Because any computable information can be encoded into a sequence of integers, this is a valid abstraction. Each method of the `Environment` has a contract which, in essence, guarantees that the environment cannot access any other part of the evoting system. More precisely, each method is required to meet the following restrictions: (1) the final state of the environment depends at most on its initial state and the parameters of the method, (2) if the method has a result value, then also the result depends at most on the initial state of the environment and the parameters of the method, and (3) at most the state of the environment is modified.

The untrusted input from the environment needs to be sanitized, but still the main loop may not terminate and voters are requested to cast their votes for an arbitrary number of times. The classes `NetworkClient` and `SMT` (for 'secure message transfer') model the network component. In the implementation, they mainly encapsulate a single message.

The functional contract for the main method in this version is identical to the one shown in Listing 9.3. In the preconditions, we assume that no voter has cast their vote yet (or more precisely, the server has not yet marked the vote as cast) and all candidates have zero votes (in the server). The postcondition states that the number of votes for each candidate is exactly the number of voters who voted for them. This is expressed using the generalized quantifier `\num_of` in Line 11. The explicit `diverges` clause allows this method to not terminate.

The method `Voter#onSendBallot()` generates a new message containing the vote of the voter and sends it over the network, see Listing 9.6 on the next page. `Setup#onSendBallot()` has two contracts. Both require that the invariant of the server holds and ensure that the final state of the environment depends at most on its initial value. They differ in the functional part: the first contract requires additionally that the voter has not voted yet. In

```
/*@ normal_behavior
  @ requires     ! server.ballotCast[id];
  @ requires     \invariant_for(server);
  @ ensures          server.votesForCandidates[vote]
  @               == \old(server.votesForCandidates[vote])+1;
  @ ensures      server.ballotCast[id];
  @ assignable   server.votesForCandidates[vote],
  @              server.ballotCast[id], Environment.rep;
  @ determines   Environment.envState \by \itself;
  @ also normal_behavior
  @ requires     server.ballotCast[id];
  @ requires     \invariant_for(server);
  @ ensures           \old(server.votesForCandidates[vote])
  @                == server.votesForCandidates[vote];
  @ ensures      \old(server.ballotCast[id])
  @                == server.ballotCast[id];
  @ assignable   Environment.rep;
  @ determines   Environment.envState \by \itself;
  @*/
public void onSendBallot(Server server) {
    Message message = new Message(id, vote);
    //@ set message.source = this;
    SMT.send(message, id, server);
}
```

Listing 9.6: Contract of `Voter#onSendBallot()`

this case the contract ensures that the server counted the vote correctly by incrementing the value of `server#votesForCandidates[vote]`. The second contract requires that the voter did already vote and guarantees in this case that the server does not count the vote again.

The counting takes place in the method `Server#onCollectBallot()`. It is called indirectly by `SMT.send()`. Because `onCollectBallot()` has a purely functional contract, it will not be considered in detail here. The same holds for the method `resultReady()` which simply returns `true` if all voters voted. In total, this implementation consists of 11 methods with 150 SLoC.

For the specification effort, this means that we need refined contracts that take into account the situation that voters have already cast votes. In the loop invariant—which is still the one displayed in Listing 9.3—we talk about a bounded sum (indicated by the keyword `\num_of` in Line 11) that is defined through a nontrivial induction scheme: the elements are not added linearly, but only under stuttering and permutation. This makes it—at least with the current machinery—impossible to prove the invariant automatically. To prove equality of sums, we had to apply the 'split sum'

rule several times interactively. This rule rewrites a sum comprehension into two comprehensions over split ranges. In addition, we have added some rules representing lemmas dealing with bounded sums to the rule base of KeY and we have proven their soundness. The proof for `main` finally took about 63,000 proof steps, of which only 10 were applied by hand. The computation time for the automated parts of the proofs was 580 s.

The specification of the `Voter#onSendBallot()` method has changed in comparison to Sect. 9.3.1. Its proof is slightly larger—from 600 proof steps and 1.3 s in the basic version to 2400 proof steps and 6 s—but the KeY prover was still able to find the proof without further user interaction. All other methods were not touched; and their respective proof is still valid.

9.3.3 Hybrid Approach Setup

Küsters, Truderung, Beckert, Bruns, Graf, and Scheben [2013]; Küsters et al. [2015]; Bruns et al. [2015a] describe a hybrid approach[7] that combines functional verification in KeY with lightweight information flow analysis based on program dependency graphs [Hammer, 2009]. In order to leverage the JOANA tool [Graf et al., 2013] to accept declassification, the original program is transformed such that it does not have any illegal information flow by construction.

```
1  private Setup () {
2      final int n = numberOfVoters;
3      final int m = numberOfCandidates;
4
5      // let adversary create fake voters
6      Voter[] v1 = createFakeVoters();
7      Voter[] v2 = createFakeVoters();
8      int[] r1 = computeResult(v1);
9      int[] r2 = computeResult(v2);
10     if (equalResult(r1,r2)) {
11         // store correct result
12         out = r1;
13         // select voters according to secret
14         voters = secret? v1: v2;
15         server = new Server(n, m);
16     } else
17         // abort if not equal
18         throw new Throwable();
19 }
```

Listing 9.7: The ‘hybrid approach’ setup

[7] This is a broader notion of ‘hybrid’ than in [Russo and Sabelfeld, 2010], where it refers to combinations of static and dynamic information flow analyses.

This technique is based on a simulation of noninterference in the Java code. The secret here is only a single bit (stored in the static field `Setup.secret`). In the setup, two arrays of voter objects are created according to the environment to simulate two possible high inputs. The program aborts in case they yield nonequivalent results. At this point in the program execution, both high inputs are incomparable modulo the declassified property (i.e., the result of the election). Then one array is chosen, depending on the secret, to be used in the main loop. This setup can be seen in Listing 9.7 on the facing page.

Since the functional property and the actual implementation did not change in comparison to Sect. 9.3.2, there are only new verification targets, namely 1. the `Setup()` constructor, that establishes the above described setup and 2. the so-called "conservative extension" method shown in Listing 9.8, that is called after the election has terminated. The extension effectively eliminates the declassification through overwriting the result, as computed by the actual implementation, with a precomputed correct result. The central goal was to prove that this extension is really ineffective (which is an even stronger property than conservatism).

Both required significant interaction in proving, while having the automated prover apply several thousands of rules in between each interactive step. Interestingly, this is mainly due to the sheer size of the code under investigation, but not to any particularly pattern that is hard to prove. By 'size,' we do not only understand single lines of code, as often in software analysis, but rather the lack of proper modularization. After all, the proof for `main` consists of over 200,000 proof steps, of which some 100 were applied by hand. The labor invested in verifying it approximately amounts to three weeks full time.

```
/*@ requires out == computeResult(voters);
  @ requires
  @     (\forall int i; 0 <= i && i < numberOfCandidates;
  @         server.votesForCandidates[i] ==
  @         (\num_of int j; 0 <= j && j < numberOfVoters;
  @             voters[j].vote == i));
  @ ensures equalResult(out,\old(out));
  @*/
private void conservativeExtension () {
    out = server.votesForCandidates;
}
```

Listing 9.8: "Conservative extension" in the hybrid approach setup

9.4 Discussion

In the course of this chapter, we presented an approach to functionally verify a Java implementation of an electronic voting system, that is to be deployed in the field. The work by Scheben [2014, Chap. 9] serves as a companion text, describing verification of secure information flow in this system with KeY. An important element of this information flow analysis is the refined definition of noninterference in the presence of objects, as introduced in Sect. 6.5.

Scheben [2014] claims that his work marks the "first time that preservation of privacy of votes could be shown *on the code level*." In fact, there are other implementations of electronic voting, that are much more elaborate [Adida, 2008; Clarkson et al., 2008; Bohli et al., 2009; Chaum et al., 2009]. But for these systems only design level properties have been proven, in particular, fine-grained information flow properties on the source level have not been considered yet.

Furthermore, the modifications in Sect. 9.3.3 lay the foundations of a hybrid approach combining KeY and the automatic analyzer JOANA. The results by Küsters et al. [2013, 2015]; Bruns et al. [2015a] show that effort can be considerably reduced through the use of automatic tools while maintaining the ultimate precision of logic-based approaches.

Lessons Learned

As Scheben [2014] states, analyses of such systems mostly target the design or the system level. Even a system like the one presented here—which can be considered small if measured in lines of code—poses a major challenge to formal verification at code level. Therefore, it is not surprising that the proofs were laborious.

Actually, far more effort than in conducting the interactive proofs needed to be put into understanding the system and developing an appropriate specification. Apart from representing the high level design, an appropriate specification needs to be correct w.r.t. the program. This in turn requires early proof attempts with prototype implementations. Our approach to first verify a very basic version and to refine it later has turned out to be helpful in this regard. It provided clear, reachable milestones.

An interesting point is that the main complexity resides in the synthetic setup that is used to model a deployed system and not in the components that are actually used. It is well-known that tools intended for code verification do not perform well at system level verification. As already noted by Woodcock, Stepney, Cooper, Clark, and Jacob [2008], verifying software that was not originally produced for the purpose of verification almost always constitutes an ill-fated endeavor. While not the size of system described by Woodcock et al., we experienced this phenomenon in the system by Küsters et al. The starting point of our verification was a final piece of software, incorporating

a considerable number of wellknown software engineering antipatterns. The program was produced without any formal development process behind it. In particular, specifications had to be conceived by ourselves, using only the present source code and informal descriptions of the components' behavior. Although there are no guidelines to produce well-verifiable programs, we believe that adherence to common software engineering guidelines would render formal specification and verification more feasible.

Nevertheless, this case study serves as a benchmark and has pushed forward several performance improvements in the KeY system. This includes both improvements in the strategy (i.e., moving to a more tractable complexity class) and practical implementation changes. In particular, some proofs forced KeY to consume a lot of memory. In the past, memory has never been the limiting force in proofs, but here KeY used up to 40 GiB of RAM. Later improvements in the implementation found by the author reduced memory consumption by 30–40% on proofs of this size. These improvements have played a large part in the development of the previous milestone release KeY 2.2 in April 2014.

Outlook

This case study clarified the boundaries to which verification scales with the KeY prover. Going even further, we performed first experiments with replacing synchronous by asynchronous message transfer. Again, the client and server components can be verified with reasonable effort, but the setup is barely tractable. One reason for that is that the votes are counted in an unorthodox way as there may be any order in which voters cast their votes. The proof thus strongly relies on reasoning about permutions. Increased support for abstraction seems to be necessary for further results.

The presented version of the system works strictly sequential. A natural next step would be to parallelize parts of the system and to verify it using the techniques developed in this dissertation. Through the client/server design, the system is obviously amenable to parallelization. In particular, there is an unbounded number of clients that each perform the same task (i.e., sending the ballot). The version with asynchronous communication would be good starting point for a parallel version.

10 Related Work

In this chapter, we review some of the work by others that is related to the topics of this thesis. For both of the two main themes—verification of concurrent programs and information flow analysis—there exist extensive bodies of research. Not everything of that can be discussed here in detail.

Overview

Temporal reasoning as well as verification of concurrent programs has traditionally been the domain of model checkers or other fully automated analysis systems. Analysis based on these systems is never complete, and sometimes not even sound. For concurrent Java, the tools Java PathFinder [Havelund and Pressburger, 2000] or Bogor [Robby et al., 2006] are available. There are several others for C and derived languages; see [Rinard, 2001; D'Silva et al., 2008; Beyer, 2015] for recent overviews. Like all model checking approaches, these suffer from the state explosion problem. Many analyses are specialized towards particular properties about concurrent programs, such as data race freedom [Abadi et al., 2006] or lock freedom (cf. [Gotsman et al., 2009]).

Noninterference is undecidable. This means that there cannot be a sound, precise, and automated analysis—at least one of them needs to be sacrificed. Most state of the art approaches are based on a preference of automation over precision. In particular, analyses based on type systems have been the predominant information flow analysis technique; cf. [Volpano and Smith, 1997].

This chapter is structured into several sections addressing particular topics. Section 10.1 discusses work on modal logics that is related to our logic CDTL and its calculus from Chap. 4. Concepts for modular specification and analysis of sequential programs are very much related to the established concepts for *concurrent* programs, where modularity is essential. This includes work on dynamic frames and separation logic. We discuss work on

modularity in Sect. 10.2. In Sect. 10.3, we discuss work that is concerned with formal analysis of concurrent programs—and as such related to our approach in Chap. 5. In particular, work on the rely/guarantee approach is covered in Sect. 10.3.1. Other approaches are based on separation logic (Sect. 10.3.2) or a combination of both approaches (Sect. 10.3.3).

The second half of this chapter is dedicated to work that is related to our analysis of secure information flow. Section 10.4 discusses other *semantical* approaches. Information flow in concurrent programs is a well-studied area of research. However, the approaches in this area are usually incomplete. Therefore, we treat it in a separate section (Sect. 10.5). A similar argument holds for object-sensitive analyses (Sect. 10.6). Section 10.7 covers specification of information flow policies. Section 10.8 contains work that is related to the electronic voting case study in Chap. 9, including approaches to verify systems that use cryptography.

10.1 Modal Logics

Henriksen and Thiagarajan [1999] describe a dynamic logic where programs are ω-regular languages, thus traces are infinite words. This logic is not branching, but instead, implicitly, traces are existentially quantified. The authors remark that evaluation over traces defined by programs is a strengthening of the until operator (the only temporal operator here). One major difference to our work is, that there is a trace which is part of the model and all traces in the specification are just subtraces of this; thus it is not possible to talk about multiple traces. On the other hand this logic is more expressive than propositional LTL in that temporal properties such as 'in every even state' can be expressed.

Beckert and Schlager [2001] extended Dynamic Logic with a modality also written $[\![\cdot]\!]$, where $[\![\pi]\!]\varphi$ stands for 'φ holds throughout the execution of π.' This can be seen as a special case of DTL because the same property can be expressed in DTL as $[\![\pi]\!]\Box\varphi$. That is, in our earlier work, the temporal formula was restricted to the form $\Box\varphi$ with φ not containing further temporal operators. Platzer [2007] introduced Temporal Dynamic Logic (dTL) where programs are *hybrid programs*; in particular, they are indeterministic, and therefore, traces are branching. It features formulae of the shapes $[\![\pi]\!]\Box\varphi$ ('for all traces, φ always holds') and $\langle\!\langle\pi\rangle\!\rangle\Diamond\varphi$ ('there is a trace such that eventually φ holds') where φ is a state formula. There is no further combination of temporal operators. Similar to our setting in Chap. 4 of this dissertation, traces can be of finite or infinite length.

Platzer presents a sequent calculus for dTL, which, however, is incomplete, much due to the continuous state space of hybrid programs. In a follow up work, Jeannin and Platzer [2014] present dTL^2 that allows a restricted set of combinations of temporal operators such as $\Box\Diamond\varphi$, but no 'until' operator.

They also present a calculus that is relatively complete when only repetition-free programs are allowed in modalities.

Dynamic logic with temporal assertions is related to Alternating Time Logic (ATL) [Alur et al., 2002], that extends CTL by featuring explicit *agents* that can choose paths. Thus, a program can be seen as an explicit representation of an actor.

Hybrid logics [Blackburn, 2000] are a family of multi-modal logics that can explicitly refer to worlds of the underlying Kripke structure. Hybrid logics feature a 'satisfaction operator' $@_i$ where i denotes a set of worlds W. A formula $@_i\varphi$ is valid if and only if there is a $w \in W$ such that $w \vDash \varphi$. Note that validity of the modal formula in this case is independent of a world. In the special case, that the Kripke structure is linear, we may write $@_n$ where $n \in \mathbb{N}$ identifies a state. In combination with quantification over the integers, this allows to embed LTL—plus stronger temporal operators like $\bullet^n$ introduced in Sect. 6.4.2. For this particular application, the use of hybrid logic is thus similar to our embedding of CDTL into JavaDL in Sect. 7.3.1.

10.1.1 Deductive Verification of Temporal Properties

Reasoning about temporal properties is traditionally the domain of model checking. Nevertheless, there is some work on deductive techniques (tableaux, sequent calculi, resolution etc.) applied to temporal logics. Good sources on the topic of theorem proving for propositional linear time logics are the articles by Wolper [1985]; Goré et al. [2011] and the textbook chapters by Goré [1999] and Reynolds and Dixon [2005].

The work by Wolper introduces a tableau method for propositional LTL. A similar approach can be found in work by Abadi and Manna [1985], which is then extended to a first order version of LTL [Abadi and Manna, 1990]. It is known that, although LTL is decidable, there does not always exist a finite proof tree. The proof graph may contain cycles in the presence of eventualities (i.e., formulas with a positive occurrence of U). There are different techniques to deal with this. In the calculus presented in this paper, we use program invariants. The above approaches are two-pass systems: first, a (cyclic) proof graph is constructed, then, strongly connected components are identified. Schwendimann [1998] uses additional bookkeeping and on the fly checks for a one-pass calculus. The sequent calculus for LTL presented by Brünnler and Lange [2008] uses *history annotations* on formulae to ensure a finite proof tree.

Thums, Schellhorn, Ortmeier, and Reif [2004] present a sequent caculus for Interval Temporal Logic (ITL) [Cau et al., 2002] in order to verify state charts [Harel et al., 1990] against ITL specifications.

10.1.2 Temporal Behavior of Programs

Bandera [Corbett et al., 2000] was one of the first projects to aim at software model checking. It is of particular interest that it employs an implementation-aware temporal specification language called Bandera specification language (BSL). The major goal of BSL was to avoid formalisms such as LTL, which are deemed to be not comprehesible to software developers. Therefore, a set of particular specification patterns [Dwyer et al., 1999] was selected to form the essential syntactical entities. In contrast to other specification languages, it is both state and event based. The language is two-tiered: First the linear and infinite timeline is divided into *scopes* which are indicated through either states or events: *global*, *before* Q, *after* Q, *between* Q *and* R, and *after* Q *until* R (i.e., weak until). Within scopes, state- or event-based *occurence patterns* describe temporal properties: absence, (bounded) existence, universality, precedence, response. For instance, to express that between any occurence of Q and R, B occurs always preceeded by A, it can be stated as: '*after* Q *until* R A *precedes* B,' which could be expressed in LTL with weak until as $\Box\,(Q \rightarrow ((\neg B \,\mathsf{W}\, A) \,\mathsf{W}\, R))$.[1] The authors argue that for most properties to be specified, LTL formulae were to complicated. However, Dwyer et al. own experimental results show that over 80 % of specifications use the 'global' scope, thus giving a property which can be easily expressed through LTL's $\Box$ operator. Bandera specifications form a subset of expressions from the intersection of LTL and CTL (if we added bounded operators). There are some LTL properties which are not expressible in BSL, such as $\Box\Diamond Q$, meaning 'Q occurs infinitely often'.

Inspired by the Bandera specification language, Trentelman and Huisman [2002] define an extension to JML with events and temporal properties. The set of permitted expressions is reduced compared to Bandera; particularily, scopes can only be triggered by events. Statements not expressible include, for instance, 'if φ holds, then eventually `m` is called', or $\varphi \rightarrow \Diamond \mathit{call}_{\mathtt{m}}$ in LTL, where φ is a state property. State properties are regular JML expressions enriched by the 'enabled' statement providing whether a method invoked in that state would terminate normally. Events in this context are calls to methods and returns from calls (either normal or exceptional). The semantics for an event to 'hold' in a state s_i of a sequence $\bar{s}$ is that it represents a transition from s_{i-1} to s_i. It is however not clarified in the paper, what is exactly meant by a 'state,' in particular, whether they only consider visible or observed states. There is a runtime checker implementation for this language called temporaljmlc [Hussain and Leavens, 2010]. Wagner [2013] provides a translation from temporalJML style specifications to DTL.

The Competition on Software Verification (SV-COMP) defines a common input language for a wide range of tools [Beyer, 2015]. It uses 'globally' and 'eventually' operators to express simple patterns, e.g., `G !call(foo)`

[1] There's actually an error in the original translation provided by Dwyer et al. [1999].

expresses that a method `foo` is never called throughout a run. These specification patterns are restricted to some nonfunctional safety properties as well as termination.

Gotsman, Cook, Parkinson, and Vafeiadis [2009] use LTL-like temporal operators to describe the temporal behavior of programs in a C-like language. Traces can be finite or infinite. This specification is used in an extension to the rely/guarantee approach for verification of concurrent programs (see Sect. 10.3.3 below).

Programs of concrete programing languages like Java are usually reasoned about in a state based manner. There are a few runtime checking approaches that check for trace properties using LTL-like specification [Bartetzko et al., 2001; Stolz and Bodden, 2006]. Hussain and Leavens also check assertions at runtime, but in addition, they use temporalJML as an extension to the JML specification language, that allows to write high level temporal properties, but is not as expressive as LTL.

10.2 Modular Specification and Verification

Together with the annotation language of the ESC/Java tool [Flanagan et al., 2002], the JML language has been a pioneer in the area of *annotation based specification languages* dedicated to a single programing language. JML has a number of similarities to the Object Constraint Language (OCL) [Warmer and Kleppe, 1999], a language for annotating Unified Modeling Language (UML) class diagrams with constraints on object states. It is used for both meta modeling and application modeling. In the latter case, annotations are added to the fine design of the implementation, much like class and method specifications in JML. But unlike JML, OCL does not subscribe to any programing language, and therefore does not address language-specific concerns (like, e.g., exceptions).

JML has been an inspiration for many other program annotation languages that have emerged over the last years, such as the ANSI/ISO C Specification Language (ACSL) [Baudin et al., 2010], and the language of the VCC tool [Cohen et al., 2009], Spec# for C# [Barnett et al., 2005], as well as Dafny [Leino, 2009, 2010] and Abstract Behavioral Specification (ABS) [Clarke et al., 2010; Johnsen et al., 2010; Hähnle, 2012], that are integrated annotation *and* programing languages.

Cohen, Moskal, Schulte, and Tobies [2010] consider global invariants and history constraints in concurrent programs. They present sufficient conditions for local checking of these invariants being sound, leading to additional proof obligations. This technique is implemented in VCC [Schulte et al., 2008; Cohen et al., 2009]. It is not complete, as invariants that are not *admissible* according to their definition cannot be proven.

Numerous case studies covering object-based data structures, such as single linked lists, can be found in the literature. See, for instance, [Zee et al., 2008; Gladisch and Tyszberowicz, 2013]. In [Bruns, 2011], the author presents a specification of maps using model fields and dynamic frames together with an implementation of maps based on red/black trees, which is one of the challenges named by Leino and Moskal [2010]. Bormer [2014, Chap. 8] compares the usage of ADTs in VCC and KeY.

10.2.1 Separation Logic and Region Logic

Separation logic [Reynolds, 2002; O'Hearn et al., 2001, 2009] is a nonclassical extension to Hoare logic. Similar to the dynamic frames approach, it allows explicit reasoning about the heap, which makes it suitable for reasoning about pointer programs and about concurrent programs. In contrast to the dynamic frames approach, separation logic deviates from classical logic. While reasoning about dynamic frames is based on classical set theory, separation properties are not made explicit. Instead, separation logic introduces the operator $*$ ('separating and').[2] It simultaneously acts as a classical conjunction and mandates the existence of a heap separation between both operands. Separation logic is nonclassical in the sense that formulae can be 'used,' which invalidates them when used twice. Like Hoare logic, separation logic is not closed under FOL operators.

As pointed out by Smans, Vanoverberghe, Devriese, Jacobs, and Piessens [2014], plain separation logic does not provide complete reasoning w.r.t. concurrent programs. They present the example program shown in Listing 10.1 on the facing page (adapted to Java by the author) where one thread infinitely increases a shared variable and an arbitrary number of threads observe that this variable is monotonically increasing. The assertion in Line 17 cannot be proven since no thread can claim exclusive ownership over the shared variable.

There are a few tools for separation logic verification, such those by Tuerk [2009]; Jacobs et al. [2011]; Amighi et al. [2012]. The VeriFast system [Jacobs et al., 2011] for verification of Java and C programs proves separation logic properties. It performs symbolic execution like KeY, but does not offer user interaction. It employs a combination of separation logic and rely/guarantee to reason about concurrency (see Sect. 10.3.2 below).

Region logic [Banerjee et al., 2013] is a kind of Hoare logic with concepts from separation logic. It is comparable to JavaDL in a sense that it features a notion of location sets (named 'regions' here) as first-class citizens of the

[2] There is a second operator $-\!\!*$ ('separating implication' or 'magic wand'), that can be intuitively percieved as the dual to $*$.

```
1  class Example {
2      static int C = 0;
3
4      void test() {
5          (new Inc()).start();
6          while (true) (new Test()).start();
7      }
8
9      class Inc extends Thread() {
10         public void run() { C++; }
11     }
12
13     class Test extends Thread() {
14         public void run() {
15             int m = C;
16             int n = C;
17             assert m <= n;
18         }
19     }
20 }
```

Listing 10.1: Example by Smans et al. [2014], that cannot be verified with separation logic

logic. Banerjee et al. [2008a]; Rosenberg et al. [2012] present systems for automated reasoning about region logic.

10.3 Deductive Reasoning About Concurrent Programs

Abrahamson [1979] presents one of the first works on the issues of dynamic logic, combining program analysis with temporal properties, and concurrency. Here an unstructured programing language with parallel composition and explicit labels gives rise to a branching time temporal structure. Trace formulae are implicitly evaluated over all possible traces. They resemble LTL formulae, but modalities may contain path conditions (typically sequences over labels). The paper does not contain formal semantics or a calculus.

Peleg [1987] introduces *Concurrent Dynamic Logic (CDL)*—based on Harel's original notion—where program modalities contain a parallel composition operator $\cap$. The programs here are linear programs; there is no shared memory. As Peleg himself acknowledges "processes of CDL are totally independent and mutually ignorant." For this reason, the formula $\langle \pi_1 \cap \pi_2 \rangle \varphi$ with π_1 and π_2 executed in parallel is just equivalent to $\langle \pi_1 \rangle \varphi \wedge \langle \pi_2 \rangle \varphi$.

The book by de Roever et al. [2001] provides a good overview over early (both compositional and noncompositional) approaches to verification of shared memory concurrent programs.

Another approach using a dynamic logic—named *multi-threaded object-oriented dynamic logic (MODL)*—is taken by Klebanov [2009]; Beckert and Klebanov [2013], that uses a realistic Java-like programing language and explicitly constructs interleaved programs. Concurrent programs are composed sequentially into a single program with multiple program pointers, that represent the state of all threads simultaneously. During symbolic execution of threads, these pointers are moved in the (unmodified) program code. This is different to our dynamic logic where we consider sequential programs executed by some thread; and program statements are deleted and the program pointer is implicitly at the beginning of the remainder. The (deterministic) scheduler is explicitly axiomatized in MODL. They use a Java-like language, but impose the rather strong requirement that all loops are atomic. It also includes atomic blocks that are symbolically executed in another kind of DL modality. Like our calculus, theirs is implemented in the KeY system, too. Through the vast possibilities of interleaved executions—it can be seen as an instance of what de Roever et al. [2001] call the *global method* for concurrency verification—this approach suffers from a high complexity (i.e., the number of possible program states grows exponentially with the number of program statements). The authors mention that the complexity can be reduced practically by making stronger assumptions about the scheduling process, which has not been constrained so far.

Ábrahám et al. [2005, 2008]; de Boer [2007] describe a deductive verification system for a subset of Java based on the Owicki and Gries approach. This subset particularly includes Java-style synchronization and dynamic thread creation. It does not feature inheritance, which is an essential feature of object-oriented languages and thus a central concern of the KeY verification approach. Programs are specified in the classical style of Hoare [1969]. Ábrahám et al. use the Prototype Verification System (PVS) to discharge the resulting higher-order proof obligations. They describe a tool called Verger, which is "the only implemented deductive verification system for multi-threaded Java," according to Klebanov [2009]. However, it does not seem to be available anymore at the time of writing.

10.3.1 Rely/Guarantee

The rely/guarantee approach was introduced by Jones [1981, 1983] and later rephrased by many authors, including Xu et al. [1997] and de Roever et al. [2001]. Proofs of soundness appear in the works by Jones [1981]; Stirling [1988]; Stølen [1990]; Prensa Nieto [2002]; Coleman and Jones [2007]. These use different definitions of rely/guarantee; a comparison is provided by

Coleman and Jones [2007]. Id. and Collette and Jones [2000] present some practical improvements to the rules of the original rely/guarantee calculus.

Prensa Nieto [2002] presents the first thorough formalization of rely/guarantee that can be machine-checked (in Isabelle/HOL [Paulson, 1994]), and probably the first formalization of any shared-variable verification technique. She assumes a single master thread that dispatches all other threads, which must not further fork. It is proven that rely/guarantee is complete w.r.t. the concurrency semantics by Owicki and Gries [1976]. This approach requires a considerable amount of interaction as it uses a general purpose HOL theorem prover.

Flanagan, Freund, Qadeer, and Seshia [2005] present a static analysis system for modular programs with indeterministic parallel composition. The discerning point of this approach is that it is modular in both the sequential and concurrent dimension of the program, meaning that method-modular proofs can be reused—as usual in sequential program verification. Although they present only a minimal target language, the authors claim that this technique can be extended to Java. However—as they acknowledge themselves—their approach is unsound.

Miné [2014] combines the rely/guarantee approach with *abstract interpretation*.[3] While rely/guarantee heavily relies on specifications, the idea is to derive them automatically because of the more simple domains. This proof technique is implemented in the automatic static analyzer Astrée for concurrent C [Blanchet et al., 2003]. Astrée checks for a predefined class of programing errors and does not consider dynamic memory allocation nor recursion. The analysis is sound, but highly incomplete. Miné [2014] introduces several dedicated abstractions tailored to verify specific programs more precisely. This includes a lock state analysis.

The work by Schellhorn et al. [2011]; Schellhorn, Tofan, Ernst, Pfähler, and Reif [2014] is closely related to ours. They extend ITL with interleaved programs and HOL. 'Programs' in this setting are represented by higher order temporal formulae that describe a trace of symbolic states. Their concurrency semantics is similar to ours in that traces contain alternations of steps of the own thread and the environment. A difference is that they consider indeterministic parallel decomposition instead of multi-threading. Building on the earlier ideas of Thums et al. [2004], Schellhorn et al. [2014] present a calculus based on symbolic execution and rely/guarantee, that is implemented in the KIV interactive theorem prover. The symbolic execution rules are similar to ours: a program is sequentially decomposed to produce a 'step form' formula, i.e., a formula on which the rule for a temporal step (cf.

[3] In the abstract interpretation approach [Cousot and Cousot, 1977], concrete value domains, e.g., the integers, are overapproximated with *abstract domains*. For instance, the domain $\{<, 0, >\}$ intuitively represents any negative number, zero, and any positive number; that can be refined to the integers. Typically, abstract domains can be described through linear inequations.

our rule R25) can be applied. Like in our approach, environment steps are not visible locally, but collapsed into a macrostep; they also assume fairness. However, rely/guarantee is not fundamental in their approach, but rather an advanced proof technique. They have encoded the complete proof system by Xu et al. [1997], but do not indicate to what extend reasoning can be automated. It appears that considerable interactive effort is necessary to conduct a proof, in particular since it regards higher order logic and the calculus does not feature an invariant rule.

Ahrendt and Dylla [2009, 2012] describe a verification system for concurrent programs written in the Creol language. Creol [Johnsen et al., 2006] is an experimental object-oriented language that features different concurrency paradigms. On an outer layer, it features *distributed objects* (i.e., distributed components that are class instances), which execute truly in parallel. Distributed objects communicate through asynchronous message passing. Conversely, intra-object execution is multi-threaded with a shared memory.

Ahrendt and Dylla apply a rely/guarantee approach to reason about this kind of concurrency. An important difference to our work is that Creol follows a *coöperative scheduling* philosophy [Dovland et al., 2005], in which sequential executions are not arbitrarily interleaved, but threads actively release control at explicit *release points* programmatically. Creol features unconditional release statements `release`, as we do, as well as conditional releases through the `await` statement. Their semantics is based on the technique by Zwiers [1989] to construct histories of interactions by nondeterministically 'guessing' environment actions. These interactions include 'yield' and 'resume' events that capture the memory state upon a release, thus bearing a similarity to our state traces. Since shared memory is strictly internal to an object, there is no need for complex (dynamic) framing annotations.

Ahrendt and Dylla present a symbolic execution calculus for a dynamic logic for Creol with an implementation in an experimental version of KeY. The calculus rule dealing with release uses a special kind of update that anonymizes the state, through a (deterministic) ε assignment [Hilbert and Bernays, 1939], such that the rely condition holds in this state. Since they make no fairness assumptions regarding the scheduler, there cannot be a sound rule for `release` appearing in a 'diamond' modality. This leaves the calculus incomplete. They neither provide a proof of soundness.

The ABS language [Johnsen et al., 2010; Hähnle et al., 2011] borrows many concepts, in particular regarding concurrency, from Creol. Din et al. [2012] and Din [2014] describe a verification system, similar to the one of Ahrendt and Dylla [2009], and an implementation in KeY. It uses a variant of the rely/guarantee approach with a high degree of automation to reason about interleavings. For the same reasons as above, their logic does not feature a 'diamond' modality, which makes the calculus inherently incomplete. KeY-ABS currently is being integrated into the master development branch

of KeY. Threads in both Creol and ABS cannot be created dynamically. The scope of their shared memory is restricted to class boundaries.

Mansky and Gunter [2012] devise a propositional temporal logic with agents, which essentially is a reformulation of ATL*. It features a modal operator $\Rrightarrow$, where $\varphi \Rrightarrow^A \psi$ intuitively means 'if agent A guarantees φ, then all other agents guarantee ψ.' As mentioned above in Sect. 10.1, programs can be interpreted as representations of agents. Thus, this logic can be applied to program verification, theoretically. However, Mansky and Gunter do not present concrete programs. It is unclear how the approach can be applied in practice. In particular, all proof obligations of the rely/guarantee approach are packed into a single formula. Furthermore, the logic is not very expressive: it is strictly propositional and it cannot express relations between two states.

10.3.2 Concurrent Separation Logic and Related Methods

While separation logic has been conceived for modular analysis of sequential programs (see Sect. 10.2.1 above), it is rather obvious that it can be applied to concurrent programs as well. The *concurrent separation logic* by O'Hearn [2007] features a parallel decomposition rule and explicit shared resources (i.e., heap partitions). But it is too restrictive to be used effectively as it requires the heap to be completely separated between parallel executions. The basic idea is similar to the original Owicki and Gries approach: to verify a thread in isolation—as if it were a purely sequential program—and to provide an additional proof of interference-freedom (that does not result in a higher proof complexity).

Most approaches mentioned in this section work under the premiss that the heap can be partitioned into globally shared and thread-local partitions. A correct partitioning is usually expressed through *permissions*. This allows to view access to the thread-local part as in a sequential program. In contrast to our approach, where we consider any concurrent interference, approaches based on concurrent separation logic assume that there is no (effective) interference at all, except where permissions are explicitly transferred.

Fractional permissions [Boyland, 2003; Heule et al., 2013] is a technique that allows that either a single thread holds a write permission on a location (i.e., it holds the full permission 1) or an unbounded number of threads hold read permissions (i.e., they hold a fraction $p < 1$). To gain write access, all fractional permissions must be transferred to the potential writer to gain full permission. Still, programs with benign data races cannot be verified using this approach since more than one thread may write to the heap within a given time scope. Bornat et al. [2005]; Jacobs and Piessens [2011] extend concurrent separation logic with permission accounting. In order to avoid the need to specify concrete permissions, Huisman and Mostowski [2015]

introduce *symbolic* permissions. Here, permissions are described qualitatively through specifications, rather than quantitatively. They introduce a dedicated data type to model permissions.

Concurrent Separation Logic for Java

The aforementioned approaches only target toy languages, while the following claim 'Java-like' languages as their goal. Both Haack and Hurlin [2008] and Amighi, Blom, Darabi, Huisman, Mostowski, and Zaharieva-Stojanovski [2014] present calculi for fork/join concurrency, based on concurrent separation logic, that allow for concurrent read actions. The goal is to model the multi-threading architecture of Java. In their methodology, there is no notion of a 'shared' memory; all memory accesses (read or write) must be explicitly permitted. Changes to the heap can be specified upon permission transfer. A complete soundness proof is provided by Hurlin [2009].

Building on the work of Haack and Hurlin [2008]; Haack et al. [2015], the VerCors system by Amighi et al. further features a high-level specification language, inspired by JML. It features separating conjunctions and 'magic wands' as in separation logic, a built-in permission predicate, and abstract specification predicates [Parkinson and Bierman, 2005] (which are similar to boolean model methods). Programs and specifications are translated to the Chalice tool [Leino et al., 2009] for verification. VerCors further provides an intermediate layer where programs and specification are expressed in an intermediate language, called Common Object Language (COL). This is meant to enable VerCors to be applied to other input languages and to support other back-ends in the future. The VeriFast system by Jacobs, Smans, Philippaerts, Vogels, Penninckx, and Piessens [2011] is based on concurrent separation logic and provides verification of concurrent C and Java, but allows shared resources (see Sect. 10.3.3 below). Both the specification styles of Chalice and VeriFast, which are inherently incompatible, can be emulated in the VerCors language [Amighi et al., 2014].

Other Approaches Based on Permissions

Mostowski [2015] combines symbolic permissions with explicit dynamic frames. In principle, this approach is similar to approaches based on separation logic, but it makes frames explicit using the framing approach described in Sect. 8.2.3. This allows to extend the verification approach of the KeY system in a natural way. To model permissions, Mostowski extends program semantics by introducing a second heap (cf. [Mostowski, 2013]), that maps each location to a permission ('none,' 'read,' 'write'). Permissions can be transferred upon synchronization points. Specifications must be self-framing, i.e., only those assertions are well-defined that state at least a read permission on the locations on which the assertion depends.

The approach is implemented in the KeY system. A high-level specification interface is provided through an extension to JML, that is intriguingly simple. It only introduces predicates to refer to the state of permissions to be used in method contracts. No additional thread specifications or intra-code annotations on permission transfer are required.

Other common techniques to analyze the behavior of concurrent programs are permission systems and ownership annotations, that are checked at runtime. Notable approaches are described by Zaharieva-Stojanovski and Huisman [2014]; Zaharieva-Stojanovski [2015]; Dinsdale-Young et al. [2010]. The experimental programing languages Spec# [Barnett et al., 2005] and Dafny [Leino and Müller, 2009] also feature ownership annotations.

Discussion

Dynamic or static verification systems based on permissions are easy to implement and local checking of permissions is efficient, because concurrent actions can be ignored safely. Yet, they require annotations inside the program code, thus breaking modularity. Also, they rarely provide complete reasoning, which makes the approach less general as the rely/guarantee approach. In particular, permissions require that programs are synchronized to work well. Threads require exclusive access to resources. While a working synchronization is definitely a best practice in the development of functionally correct software, for a *security* assessment, we are forced to consider unsynchronized programs, in general.

10.3.3 Combining Rely/Guarantee and Separation Logic

Vafeiadis and Parkinson [2007] introduce RGSep, an extension to separation logic that includes rely/guarantee reasoning. Rely/guarantee does not only provide functional specifications for threads, but also separates the memory on which threads work. The central idea is similar to our approach to frame possible write effects and thus to reduce the overhead for functional specification. The main difference is that we make our frame annotations explicit, while in separation logic it is enshrined in the 'star' operator. There is no notion of local variables in RGSep, but annotations are used to dynamically separate the heap into a shared partition and local partitions for each thread. Rely/guarantee specifications only apply to the shared partition. Feng et al. [2007] describe a similar approach and provide a comparison between concurrent separation logic and rely/guarantee.

Gotsman et al. [2009] present a verification system for RGSep, which they use to verify lock freedom in implementations. In their work, rely and guarantee conditions are LTL-like temporal formulae (cf. Sect. 10.1.2). A rely condition describes the subtrace produced by executing the environment steps. This allows reasoning about global liveness properties under possibly unfair

schedulers. Using the traditional two-state properties would be unsound in this case, since guarantees specify atomic steps, but not progress [Abadi and Lamport, 1995]. This issue does not apply to CDTL, as we only consider liveness properties that are *local* to a thread while assuming a fair scheduler.

Dodds, Feng, Parkinson, and Vafeiadis [2009] discuss the inability of rely/guarantee to specify the behavior of programs that use forking and joining of threads, as opposed to parallel composition (on which rely/guarantee is traditionally defined). Forking and synchronization (of which joining is a special case) are the mechanisms used in real world programing languages. With those, the lifetime of a thread is controlled dynamically. To deal with programs of this shape, they propose an approach called "deny/guarantee," building on the work by Vafeiadis and Parkinson, in which nonbehavior of environments is specified using separation logic. These specifications are also dynamic, unlike the statically defined invariants of rely/guarantee.

The VCC system [Cohen et al., 2009, 2010] presents a practical approach to functional verification of concurrent C programs. Since the C language is even less tractable than Java, their approach makes several assumptions about the underlying hardware and operation system layers, such as the accessibility of non-volatile memory.

VCC heavily relies on permission annotations to clearly separate the heap into global and thread-local partitions. This enables reasoning about thread-local memory to resemble reasoning about sequential programs; concurrent changes need only be considered at certain synchronization points, namely when entering an atomic block [Cohen and Schirmer, 2010]. Keeping in mind that there is no language support for atomic blocks in Java, their result is similar to our argument on observability of environment actions in Sect. 3.6, where we postulate that `release` may only appear before reading from global memory. The VCC methodology uses a kind of rely/guarantee specification where environment changes are effective: before entering an atomic block, a two-state invariant can be assumed, otherwise the shared memory is havocked. This invariant must be established after the atomic block. Here, rely conditions and guarantees collapse into a single invariant.

As a product by Microsoft Research, VCC targets applicability in industry-sized software projects by software engineers. Programs and user-provided annotations are translated to large monolithic FOL formulae through weakest precondition computation, which can then be proven by an automated SMT solver. The overall approach is carefully engineered—it has been successfully applied to very large case studies—but lacks documentation on the core theory. In particular, no formal soundness proof is available.

The VeriFast system [Jacobs et al., 2011] follows a similar approach to VCC. It provides verification of concurrent C and Java programs, based on automated theorem proving for concurrent separation logic, whereas VCC relies on classical logic. Smans, Vanoverberghe, Devriese, Jacobs, and Piessens [2014] extend concurrent separation logic by introducing *shared*

boxes, that encapsulate shared variables with a two-state invariant (i.e., a rely condition). This invariant must hold whenever a thread accesses the shared variable, similar to the invariant semantics for atomic blocks above. Box annotations tend to be verbose. In addition to the declaration of a box, it requires a considerable number of annotations in the running code.

The "Concurrent Views Framework" by Dinsdale-Young et al. [2013] is a metatheory of reasoning principles for shared-memory concurrent programs. In the framework, both rely/guarantee, the Owicki and Gries method, concurrent separation logic, and other approaches can be expressed.

10.4 Semantic Information Flow Analysis

Joshi and Leino [2000]; Amtoft and Banerjee [2004] were among the first to encode noninterference in a program logic and therefore to lay the foundation for sound and complete reasoning about information flows. Joshi and Leino use classical (i.e., non-relational) Hoare logic with explicit havoc statements in programs to express the low value indpendence.

This is a different approach to formalization that we take in Sect. 6.2.5: we use a relational property, viz. the equivalence of two runs starting in low-equivalent initial states. Darvas, Hähnle, and Sands [2005] continue the idea by Joshi and Leino and apply it to Java, also considering leakage through nontermination and abnormal termination. They use JavaDL instead of Hoare logic, which allows to express this independence property through existential quantification. The resulting formulae can be proven in KeY with some interaction. This approach has still some drawbacks: Firstly, the formalization using existential quantification is not amenable to automation. Secondly, the formalization uses the 'old' memory modeling of KeY (cf. [Beckert et al., 2007b]), that considers each memory location as a single nonrigid 0-ary function. As discussed in Sect. 3.3, this modeling is unhandy when applied to dynamic data structures and does not cater for modular specification.

Scheben and Schmitt [2012a]; Scheben [2014] present several practical improvements to the approach by Darvas et al. [2005] and an implementation in KeY. In particular, they formalize the usual property that two runs starting in low-equivalent states must produce low-equivalent outputs (cf. Sect. 6.2). Secure information flow can be formalized in HOL, and higher order theorem provers like Coq can be used for checking secure information flow [Nanevski et al., 2013]. This approach seems to be very expressive, but comes at the price of more and more complex interactions with the proof system. Reasoning about HOL formulae is harder than FOL reasoning: a sound and complete calculus is even theoretically not possible.

Since classical Hoare logic cannot express noninterference, Amtoft and Banerjee [2004] employed a dedicated relational extension of region logic

[Amtoft and Banerjee, 2007]. Self-composition [Barthe et al., 2004] works with an off-the-shelf program logic but uses program transformations to express relational properties. Product programs [Barthe et al., 2011, 2013a] are similar to self-composed programs, but again use a dedicated program logic. There is unpublished work by Scheben and Schmitt [2012b] that sketches an adaptation of dynamic logic to product programs.

Balliu, Dam, and Le Guernic [2011] explicitly model knowledge of attackers (and trusted agents) through epistemic temporal logic. Epistemic logic is a multi-modal logic (cf. [Harel, 1984]) that features a modal operator K_A for each agent A, where $\mathsf{K}_A\varphi$ intuitively means 'A knows φ.' To reason about epistemic properties of programs, Balliu et al. first construct a model of the program through symbolic exploration using Java Pathfinder. The result is a "symbolic output trace," that consists of path conditions and concrete execution states. An epistemic formula can then be checked against this model with the model checker MCMS. Like all model checking approaches, it suffers from state explosion. Moore, Askarov, and Chong [2015] extend this logic by an 'effect' operator E_A to model integrity, where $\mathsf{E}_A\varphi$ intuitively means 'A can change φ.'

10.4.1 Semantic Declassification

Declassification is not included right away in many definitions of noninterference from the literature, cf. [Sabelfeld and Myers, 2003a]. In these definitions, input and output agreement are the same relation. Declassification is considered inducing a different security policy, frequently named *relaxed noninterference* [Li and Zdancewic, 2005]. Mantel [2001] explicitly distinguishes between security policies and declassification relations. Sabelfeld and Sands [2009] provide a good overview over declassification policies. A considerable body of work considers *partial equivalence relations (PERs)* [Abadi et al., 1999; Sabelfeld and Sands, 2001] to model declassification. Another significant direction of research is concerned with *quantitative* information flow [Clark et al., 2001]. Provided a finite state space, the worst case entropy of released expressions can be quantified using information theory. Klebanov [2014] presents a tool chain for automated quantitative information flow analysis, that uses KeY to compute a precise denotational semantics from a program.

Scheben and Schmitt [2012a] present a unified notation of low-equivalence and declassification of terms. They use higher order terms called *views*, later renamed to *observation expressions* [Beckert et al., 2014, Sect. 3], which semantics is a sequence of terms again. Low-equivalence of states is defined as equality of the values of the component terms. The common notion of low security locations is subsumed by this. However, the authors do not provide a formal semantics of observation expressions. Greiner, Birnstill, and Krempel [2013] present a case study that uses this approach and the

implementation by Scheben [2014]. Whether information is considered secret or public depends on the internal state of the system; i.e., it is classified dynamically. Therefore, the information flow specifications of Greiner et al. make use of conditional observation expressions. Nevertheless, the modeled attacker of Scheben [2014] is equally capable as the one considered in this work.

10.4.2 Combining Precise and Other Analyses

While semantical approaches are ultimately precise, there are frequent patterns in which this precision is not required to prove the presence or absence of information flows, as this is clear from syntax alone. One would like to reason about these patterns in an approximate way, i.e., being able to deduce information flow security of the whole program from the premiss that both branches are secure and the branch condition does not have an effect on security. In this way, this kind of reasoning is similar to flow sensitive type systems and be performed efficiently. Similar approximation calculi based on dynamic logic have presented by Ruch [2013] and Scheben [2014, Chap. 7], both as additions to the calculus of Scheben and Schmitt [2012a]. While Ruch uses a relational dynamic logic dedicated to detecting the absence of information flows, Scheben [2014] introduces relational predicates on the formula level.

In both approaches, the price for this is, of course, losing completeness. For instance, the secure Program 3 from Listing 6.1 on page 120 could not be dealt with since the branch condition does actually depend on a high location, and there is a direct assignment from high to low in one branch. Still, in both approaches, it is possible to resort to the original, complete calculus.

Hähnle, Pan, Rümmer, and Walter [2007],2008 and Popescu, Hölzl, and Nipkow [2012, 2013a] embed traditional type systems into first/higher order logic. Even though reasoning is done using a theorem prover (KeY or Isabelle, respectively), it still suffers from the inherent incompleteness of type systems and only checks for sufficient conditions for secure information flow. For instance, it is not able to verify program 3 in Listing 6.1 on page 120.

Static information flow analysis can be combined with *abstract interpretation* [Cousot and Cousot, 1977] to obtain an automated approach based on theorem proving. The obvious drawback is losing precision due to the abstraction. Approaches based on this idea have been published by Jacobs, Pieters, and Warnier [2005] and Giacobazzi and Mastroeni [2010]. Furthermore, Bubel, Do, Hähnle, and Wasser [2014]; Wasser [2015] suggested to combine information flow analysis in KeY with abstract interpretation.

10.5 Information Flow Analysis for Shared-Memory Concurrent Programs

In one of the first works on noninterference for concurrent programs, Smith and Volpano [1998] show that classical (i.e., Cohen-style) noninterference can be proven using type systems for multi-threaded programs with a purely indeterministic scheduler if noninterference holds for sequential programs and there are no loops which execution depend on high input. This is a very strong restriction and seems to be induced by both the complexity of indeterministic program semantics and the inability of type systems to recognize a secure program. For a general overview over information flow policies for concurrent programs, cf. [Giffhorn and Snelting, 2015, Sect. 8].

JOANA is a static analysis tool [Graf et al., 2013] based on PDGs [Hammer, 2009; Hammer and Snelting, 2009], that targets concurrent Java programs. It supports program analysis for either sequential noninterference or *relaxed* low-security observational determinism (RLSOD), a weaker version of LSOD that permits indeterminism on low variables [Giffhorn and Snelting, 2015].[4] Despite being a static analysis, the PDG approach works on non-abstract system states, in particular, it includes book-keeping of all running threads. A generalization to a thread-modular analysis is not considered so far. While providing a practically better precision than approaches based on type systems, the PDG approach does not consider values and thus it is not complete.

Garg, Franklin, Kaynar, and Datta [2010] describe an approach to compositional formal information flow analysis of multi-threaded programs. Unlike most approaches, that consider parallel composition, but alike ourselves, they consider threads in an underspecified environment. A difference, however, is that Garg et al. do not consider communication—and thus information flow—through shared memory, but only through prescribed interfaces. Attackers are confined to those. Thread and environment behaviors are specified in a temporal logic fragment of hybrid logic (see Sect. 10.1), that can express temporal intervals. Atop hybrid logic, they develop a program logic, that is similar to dynamic logic. It features two program modalities: one for termination within a time frame and one for exceeding a time frame. To reason about concurrent effects, Garg et al. employ a variant of the rely/guarantee approach, where rely conditions and guarantees describe traces of actions. Overall, their framework is very abstract and does not feature concrete program semantics. It is unclear how complete their approach is and how it can be automated.

[4] As there is no formal definition of RLSOD yet—only an algorithm to check it—we cannot compare it to other properties.

10.5.1 The Role of Scheduling

The effect of schedulers on information flow security is discussed controversially in the literature. Many security policies are only valid for a particular scheduler. For instance, the approaches by Smith and Volpano [1998]; Sabelfeld [2001]; Barthe et al. [2004]; Mantel et al. [2006] assume a completely indeterministic scheduling, while Volpano and Smith [1999]; Smith [2003] use uniform indeterministic scheduling, and round-robin scheduling (i.e., deterministic uniform scheduling) is assumed by Russo et al. [2006]; Russo and Sabelfeld [2006]. In contrast, the *strong security* policy by Sabelfeld and Sands [2000] is scheduler independent for a "natural" class of schedulers. As the name suggests, strong security is a very restrictive property. On the other hand, this comes not with a surprise: Sabelfeld [2003] proves that this is the weakest possible information flow policy in the presence of this class of schedulers, that is thread-compositional. Mantel and Sudbrock [2010] present a weaker, yet compositional, policy called flexible scheduler-independent (FSI) security. Concurrent programs composed from FSI-secure threads are secure w.r.t. a probabilistic extension of noninterference for a restricted class of probabilistic schedulers, the *robust schedulers*.

Huisman, Worah, and Sunesen [2006] consider information flow in finite state programs with parallel composition. They present two faithful formalizations of LSOD in CTL* and the polyadic μ calculus [Andersen, 1994]. This allows automated analysis using model checking. While the security property itself is formalized in logic, the program under investigation forms a complicated model—based on product programs—against which the formula is checked. Instead of the commonly used definition of LSOD by Zdancewic and Myers [2003], that relaxes equivalence to finite prefixes, Huisman et al. present their own property that considers the complete (location) trace, while still tolerating stuttering. They argue that LSOD under prefixing does not consider certain indirect information flows. Given the immense state space of concurrent programs, this approach does not scale well.

Noninterfering Schedulers

As mentioned above, LSOD has been criticized for being overly strict by disallowing 'benign indeterminism.' Example 6.17 on page 135 displays an example of a program that is considered insecure under LSOD, even though no high variable does syntactically occur. This is due to the possibility that the scheduler itself may leak secret information encoded in the schedule. An interesting direction of research is thus to analyze (and prevent) this kind of leakage. Surprisingly, there is little literature on this topic.

Popescu, Hölzl, and Nipkow [2013b] introduce a notion of "noninterfering schedulers." However, they deviate from the language-based scenario in that they do not consider confidential information in the memory, but classify the

threads into 'visible' and 'invisible.' According to their definition, a scheduler is noninterfering if the schedule when projected to visible threads is the same as a schedule for visible threads alone. As a result, possibilistic [McLean, 1996] and probabilistic noninterference [Sabelfeld and Sands, 2000] collapse for this class of schedulers. However, this result only applies to threads that operate on strictly separated memory; e.g., it is not applicable to Ex. 6.17.

Russo and Sabelfeld [2006] explicitly construct a round-robin scheduler that is aware of security levels. The thread pool is partitioned into 'low,' 'high,' and 'temporarily high' threads, which are treated differently, depending on the current system state. This dynamic security classification of threads has to be provided in the code. The resulting scheduler does not leak high information. Under Russo and Sabelfeld's scheduler-specific notion of noninterference, the program from Ex. 6.17 is secure.

10.5.2 Compositionality

By 'compositionality' we understand that two concurrent systems that each are secure for themselves can be composed into a secure system again. While most security properties are closed under sequential composition, *parallel* compositionality requires severe restrictions on intermediate execution states.

Mantel, Sands, and Sudbrock [2011] introduce a weaker notion of compositionality, which they claim to be effective in practice. The goal is to develop a thread-compositional analysis for Cohen-style noninterference (i.e., regarding only final states as output, in contrast to LSOD). Mantel et al. target a smaller completeness gap compared to the type-checking approaches by Volpano and Smith [1999]; Sabelfeld and Sands [2000]; etc. They present a framework that considers for each thread the sets of variables that are (i) assumed not to be read by the environment, (ii) assumed not to be written by the environment, (iii) guaranteed not to be read by us, and (iv) guaranteed not to be written by us. Membership in these sets can change throughout program execution. Mantel et al. [2011] develop a variant of noninterference, called *secure information flow using modes (SIFUM)*, in which the low-equivalence relation is weaked by considering only those low-security locations that are possibly read by the environment. The idea is to allow intermediate information flow to low locations as long as they are not read during that period. In the final state, the values stored in low locations must not be secret, however.

Mantel et al. present a type system to check for security violations. The analysis relies on permission accounting in code annotations (cf. Sect. 10.3.2 above). While the type system ensures that guarantees are ideed fulfilled, it is not clear how assumptions on the environment are backed by guarantees. On the semantical level, the approach by Mantel et al. is similar to our rely/guarantee framework in Chap. 5, where sets are specified that are assumed not to be written by the environment and at most written by us. However,

only to specify location sets is too coarse-grained, in general. It may be appropriate for an analysis based on a type system since that is incomplete anyway. For a complete analysis, that considers actual variable values, a full functional rely/guarantee specification is necessary.

10.5.3 Analysis of Timing Channels

Sabelfeld and Myers [2003a] introduce a notion of *timing-sensitive* noninterference: in addition to the final states of two terminating runs being low-equivalent, the number of execution states must be the same (or both runs do not terminate). There are several type system approaches that give sufficient conditions for the absence of timing leaks, e.g., by Volpano and Smith [1999], as well as program transformation algorithms that remove timing channels, e.g., by Agat [2000].

Huisman and Ngô [2012] introduce scheduler-specific observational determinism (SSOD). It extends the weaker property that is proven by the type system of Zdancewic and Myers [2003] (that implies LSOD), named SSOD-1 here, with a second property (SSOD-2) that requires that there *exists* a scheduler for which traces are stutter-equivalent. They provide a partial formalization in CTL, based on self-composition of programs. that can be model checked efficiently. Ngô [2014]; Ngô, Stoelinga, and Huisman [2014] present an approach to verify programs against SSOD, based on a dedicated model checking algorithm. In this system, the security policy is directly encoded in the model to check.

All the above approaches do not consider temporal declassification. We are not aware of any (semantically) precise analyses of timing channels. Dimitrova, Finkbeiner, Kovács, Rabe, and Seidl [2012] introduce a relational extension to LTL, called SecLTL, that includes explicit temporal operators for information flow. Their goal is to reason about deterministic interactive programs [Clark and Hunt, 2009]. There are operators H ('hide until') and its dual L ('leak while'), which are indexed with location sets. The intuition behind the formula $\mathsf{H}_{H,L}\varphi$ is that no information (evaluated in the current state) flows from the locations in H to the locations in L (i.e., H is hidden from L) as long as φ does not hold (i.e., until φ becomes true, but this does not necessarily happen). The formula φ can be seen as a release condition that includes the timing of declassification. The dual formula $\mathsf{L}_{H,L}\varphi$ means that a flow from H to L releases φ. In this context, the location sets H and L are static, but generalizations with expressions of type $\mathbb{L}$ are also possible. Dimitrova et al. [2012] employ a model checking algorithm to prove relational properties of SecLTL.

The author of this dissertation shows [Bruns, 2014b], using the techniques from Sect. 6.4.2, that any property of the form $\mathsf{H}_{H,L}\varphi$ can be expressed in the temporal logic fragment of DTL. This technique is complete, i.e., it does not report false positives or unknown results, but not very efficient. The

author presents additional rules that are applicable to a certain subset of formulae, local reasoning in the style of Ruch [2013] can be applied to the H operator.

10.6 Object-sensitive Secure Information Flow

The work closest to ours in Sect. 6.5 is the one by Amtoft, Bandhakavi, and Banerjee [2006]. The authors build on region logic (see Sect. 10.2.1 above) and use a similar definition of object-sensitive secure information flow. However, instead of providing verification conditions which can be discharged with an established calculus, as we do with KeY, they introduce a more specialized, but incomplete, calculus to show object-sensitive secure information flow. This specialized calculus uses approximate rules which avoid an explicit modeling of isomorphisms, but comes at the price of imprecision.

Precision and a mature tool integration further distinguishes our approach from other approaches. This particularly includes JIF, which already presented an incomplete treatment of object-sensitive secure information flow for Java by Myers [1999]. JIF is a practical approach to the analysis of secure information flow, that covers a broad range of language features, but it has not been formally proven to enforce noninterference. JIF itself is a dialect of Java, that includes features for information flow specification. Nikolov [2014] presents a translation from the JIF specification to JML extension introduced in Sect. 8.4.

Similar to JIF, Barthe, Pichardie, and Rezk [2013c]; Banerjee and Naumann [2002] use type systems for the verification of object-oriented secure information flow. They treat a smaller set of language features, but prove that their type systems indeed enforce noninterference. A closely related approach is presented by Beringer and Hofmann [2007]. Here, only the information flow analysis is based on type systems; the verification task is separated from the analysis and based on program logics.

The static analyzer JOANA (see Sect. 10.5 above) targets Java and Android programs. This approach has a notion of heap and objects, but there is no dedicated information flow property implemented, that considers leaks through object references. The analysis carried out by JOANA works on compiled bytecode. Application of the approach to Android requires another analysis [Mohr et al., 2015], despite the common source code language. The approach by Barthe et al., already mentioned above, and the approaches by Hansen and Probst [2006] and Hedin and Sands [2005], too, target Java Bytecode in contrast to source code, as the other approaches do. The latter is based on type systems, whereas Hansen and Probst use abstract interpretation in combination with classical static analysis for the verification of secure information flow.

To the best of our knowledge, the only approach which models object isomorphisms explicitly is by Naumann [2006]. He uses self-composition and program instrumentation to analyze the program in the static checker ESC/Java2 [Cok and Kiniry, 2005].[5] The central drawback of Naumann's approach is that the specifier needs to track the isomorphism manually with the help of additional ghost code annotations. This increases the burden on the specifier, whereas our approach detects the isomorphism automatically.

10.7 Information Flow Property Specification

In Sect. 8.4, we have used an extension to the Java Modeling Language (JML) by Scheben and Schmitt [2012a]; Scheben [2014]. to specify noninterference and declassification. Dörre and Klebanov [2015] further extend their specification to express the *presence* of information flows, rather than the absence. Their goal is to prove that Android's implementation of a pseudo-random number generator does not lose any initial entropy.

Dufay, Felty, and Matwin [2005] devise their own JML extension with new keywords which directly define relations between the program variables of two self-composed executions. In particular, two keywords to distinguish the variables of the two runs are defined. The approach uses ghost code to store the return value and the values of parameters of the first run in order to use those values during the application of non-interference contracts in the second run. The approach is limited to primitive types. In contrast to dedicated extensions, Warnier [2006] and Haack, Poll, and Schubert [2008] describe techniques to specify properties in vanilla JML.

In our approach, security is specified on the code level. Other approaches are model-based. The UMLSec language by Jürjens [2005] adds high level security properties to UML design models. Both static and dynamic UML diagrams can be annotated with c. 30 predefined stereotypes, such as «secret» on a class, «encrypted» on a message, which denote some inherent property. Requirements are also denoted by steoretypes (on whole subsystems). Jürjens outlines how a secure Java program can be refined from the design model. The behavior of both systems and attackers is modeled by means of "UML-machines," which are introduced for this purpose. The given stereotypes are translated to constraints on those machines, that can be model-checked.

Kramer, Hergenröder, Hecker, Greiner, and Bao [2014] extend the Palladio component model [Becker et al., 2009] with security annotations. In contrast to the work by Jürjens [2005], the goal is derive verification conditions for software artifacts. The approach by Kramer et al. contains a complete refinement chain, down to information flow proof obligations for KeY and JOANA.

[5] ESC/Java2 offers neither sound nor complete analysis, which makes the overall approach unsound, unfortunately.

10.8 Implementation-Level Analysis of Electronic Voting Systems

Bär [2008] specifies functional properties of a Java implementation of the Bingo Voting system by Bohli et al. [2009] with the Java Modeling Language. These specifications have been partially checked with the (unsound and incomplete) ESC/Java2 tool by Beck [2010]. Kiniry, Morkan, Cochran, Fairmichael, Chalin, Oostdijk, and Hubbers [2006] report on the Dutch Kiezen op Afstand (KOA) remote voting system, that has been used in the European Parliament election in 2004 for a small group of voters. In order to specify the (offline) vote counting module with JML and subsequently analyze it with ESC/Java2, they reïmplemented the KOA system in Java.

Clarkson, Chong, and Myers [2008] mention that their Civitas system has been checked for information flows with JIF [Myers, 1999]. For the system described in Sect. 9.3.2 of this thesis, Scheben [2014] formally proved noninterference and declassification in the KeY system.

10.8.1 Verification of Cryptographic Implementations

Symbolic approaches, using ideal cryptographic functionality, can be seen as state of the art. There are other approaches that include formal reasoning about cryptographic guarantees [Stern, 2003] in the code analysis. This is usually named the *computational* approach. Barthe, Grégoire, and Zanella Béguelin [2009], von Gleissenthall et al. [2014] present a framework in which adversaries can be modeled as probabilistic polynomially bounded `while` programs. A probabilistic relational Hoare logic—extending Benton's logic [2004]—allows to formally reason about these adversaries, that is implemented in the EasyCrypt system [Barthe et al., 2013b].

Fournet et al. [2011]; Barthe et al. [2014] present a functional programing language with dependent types based on relational assertions. Information flow analysis for this language is provided through typing, where the functional/relational analysis required to evaluate dependent types is delegated to an SMT solver through verification condition generation (VCG). This allows for automated proofs, while not reaching the expressivity of EasyCrypt, that relies on interactive proofs.

Our analysis strictly targets the code level. In contrast, there is a large area of research dedicated to the (formal) analysis of security *protocols*. These analyses work on an abstract level and do not consider the particular issues of implementations. See, for instance, the textbook by Anderson [2008] or the overview articles by Cortier et al. [2010]; Cortier and Kremer [2014].

11 Conclusion

In this dissertation, we have introduced a logic-based approach to the functional and relational verification of concurrent Java programs. It includes two major parts: 1. We have developed a deterministic interleaving semantics for shared-memory multi-threaded programs, on which we have build a trace-based dynamic logic based on symbolic execution and rely/guarantee. 2. We have developed a semantically precise analysis technique for noninterference in concurrent programs. In this concluding chapter, we shortly summarize the contributions that we have made and draw conclusions from them. We finish with an outlook on future work in Sect. 11.3.

11.1 Summary

Concurrent programing has been state of the art for some time. Still, formal verification of programs is yet in an early stage. Chapters 3–5, 7, and 8 are the result of the author's contribution to this challenge. We have introduced a simple, but multithreaded, programing language in Chap. 3 with a deterministic semantics, that is scheduler-parametric. Concurrent programs, according to our definition, are collections of sequential programs, that may have provisions for dynamic thread creation. We first define semantics of sequential programs in Sect. 3.4 and then develop a semantics for interleavings on top of this in Sect. 3.5.

We have demonstrated how to define a concurrent language as a *conservative* extension to a sequential language. This allows to reuse several important results. In particular, a central goal of this work is to enable the KeY verification to reason about concurrent Java programs. In Sects. 3.6 and 7.2, we argue that our model of concurrency can, in principle, be lifted to concurrent Java. There is a remaining gap, that is mostly induced by our decision to assume sequential consistency and to not model the JMM. We expect this gap to be negligible in practice.

To formally reason about concurrent programs, we adapted the rely/guarantee approach in Chap. 5. In order to make this approach entirely thread-modular, we leave out global proof obligations. Instead, we introduce a proof correctness framework in which we assume initial validity of all thread specifications and prove its preservation upon thread creation. A further extension of the classical rely/guarantee approach, that is based on functional specification, is the introduction of frame clauses. They are essential to specify nonbehavior and thus reduce specification verbosity.

The rely/guarantee framework is embedded in a sequent calculus for a novel dynamic logic, CDTL. CDTL is a conservative extension of the previously defined DTL, which introduced a marriage of a program logic and a temporal logic. By extending the sequential language that DTL targets, we also extend the logic. Additionally, we include some first-order definable theories, which do not raise the expressive power, but they are convenient to represent program effects. The extension paradigm also applies to the calculus: In Sect. 4.4, we introduced a sequent calculus, based on symbolic execution of sequential programs, for the base DTL. We proved this calculus sound and complete in Sect. 4.5.

The extension to a calculus for interleaved dWRF programs, and thus to full CDTL, is minimal. It only consists of symbolic execution rules for thread creation and for interleavings, which forms the core of the rely/guarantee approach. Proving soundness of this rule is far from trivial (see Thm. 5.13 and Thm. 5.15). However, the central soundness result in Thm. 5.22 for the overall calculus is not difficult to prove, given that it only depends on the soundness of two additional rules.

Secure information flow for concurrent programs has been studied intensively. This includes both the development of dedicated security policies, such as LSOD or SSOD, and of analysis systems thereof. However, syntax-directed static analyses like type systems—to which the vast majority of analyses belong—or PDG analysis are incomplete w.r.t. such semantically defined policies, by design. These approaches are fundamentally restricted to overapproximate and possibly yield false-positive results.

In this dissertation, we presented the first semantically precise approach to analyze secure information flow in shared-memory concurrent programs. This analysis is thread-modular and appropriate to be applied to Java programs. Furthermore, it includes semantical declassification. This allows to precisely state what *information* is at most revealed. For instance, in an election, the final result is public information, even though it depends on the otherwise secret information about individual ballots. Overall, our framework is flexible enough to support extension like declassification in a natural way.

Our approach builds on a faithful formalization of state-based (see Sect. 6.2) or trace-based notions (see Sect. 6.4) of indistinguishability and

the respective notion of noninterference arising from them. These proof obligations can then be discharged with the KeY prover. Given that we use a sound and complete calculus, such a proof represents a sound and complete information flow analysis. The overall work flow of an analysis of secure information in Java programs is depicted in Fig. 11.1.

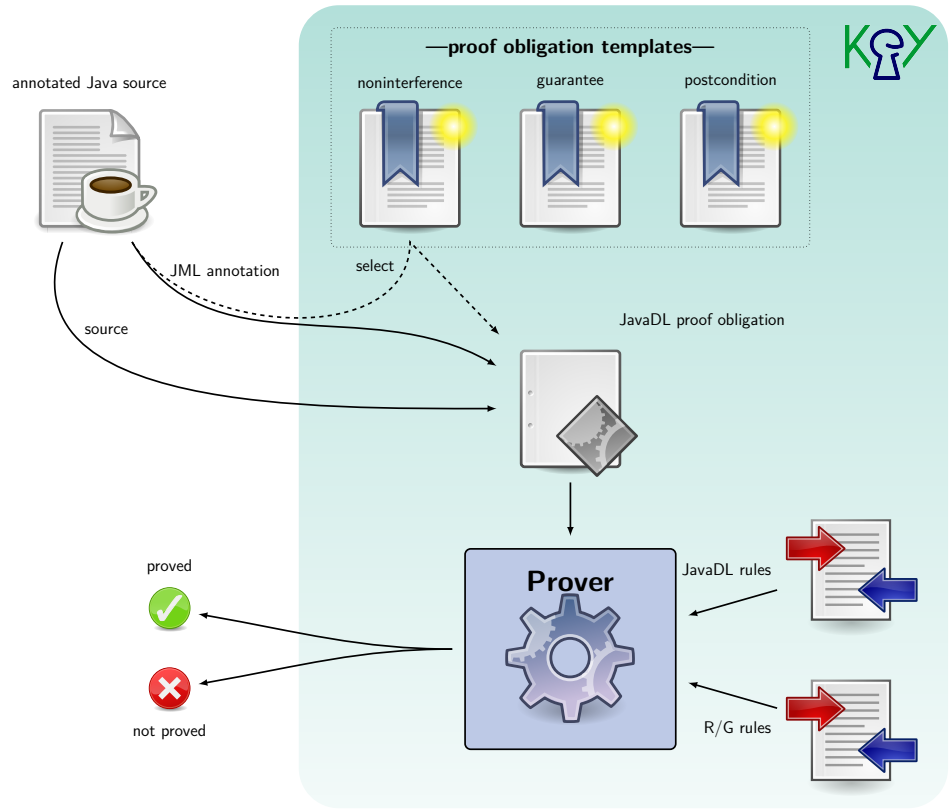


Figure 11.1: Work flow for information flow analysis

Considering a concrete real-world target language like concurrent Java imports further complications to the scene. Firstly, Java is a very feature-rich language in general. However, most of these features can be seen as syntactic sugars or as instances of more general principles that can be mapped to more tractable languages like dWRF. Many of these issues are discussed in Sects. 6.6 and 7.2. Our overall approach benefits from the already high coverage of Java in the KeY verification system.

Secondly, not all dependencies on the semantical level actually relate to an insecure information flow on the language level. We have to take into account security that is established by the language definition itself. A particular instance of this situation is the pointer opacity in Java. As a result, the scenarios in which information is actually leaked through object identities are limited. We describe an approach to represent this fact in

the semantical information flow properties and the analysis technique in Sect. 6.5.

Specification on a higher level of abstraction is a significant relief for logic-based analysis of complex systems. The development of modular analysis techniques even for sequential programs is still a major challenge (cf. [Hoare and Misra, 2005; Leavens et al., 2007]). The features for modularization in JML and its extensions are a key to this as described in Chap. 8. We build on these features to develop thread specifications based on the rely/guarantee methodology that are appropriate for a modular analysis of concurrent Java. The results have been presented in Sect. 8.3. Likewise, specification of information flow properties shares many elements (see Sect. 8.4).

We apply our extension to JML to the specification of an electronic voting system in Chap. 9. The system has been developed in the RS^3 project, with the concrete implementation under investigation produced by the author. The functional verification of this case study is part of a hybrid approach involving JOANA as a tool for information flow analysis [Bruns et al., 2015a; Küsters et al., 2015].

11.2 Concluding Remarks

In this dissertation, we have described important steps to develop a usable verification system for multi-threaded Java programs. The extension to concurrency is a paradigm shift for a verification system like KeY, that has been originally developed for sequential programs. This change is fundamental and the quest for concurrency verification methods is definitely not concluded yet. Jones [2003] "[. . .] recognises that there are also quite general problems to be faced before a satisfactory compositional approach to the development of concurrent programs can be claimed." The author regards this work—and the resulting implementation in KeY—as a step towards this. As mentioned before, the implementation is yet in an early stage. However, we have at least sketched the central ideas in Chap. 7. Our implementation described there is not just meant as a proof of concept, but it is planned to find its way into a future stable version of KeY. We expect that it reaches maturity eventually with more case studies to be completed and the approach being evaluated critically against other approaches, such as permissions [Mostowski, 2015] (cf. Sect. 10.3.2).

Although it may seem as a detour, the introduction of the toy language dWRF proved to be beneficial for developing a sound and complete analysis. The structural simplicity of the language enables us to provide formal meta-level proofs about the approach. Instead, including actual Java syntax and semantics already at this stage would render definitions and proofs much more complicated, while it would not reveal any significant new insight. Attaining

an entirely complete analysis technique for Java is arguably not realistic. This is already indicated by our long-standing experience in sequential verification with KeY.

The rely/guarantee methodology allows for sound and complete modular analysis of concurrent programs. In contrast to classical definitions, that are based on closed programs with parallel composition, our variant of rely/guarantee targets the modular analysis of *threads*. We prove thread properties independently of the actual runtime environment. This enables truly modular analysis, in the sense that we may consider thread executions w.r.t. any number or type of already concurrently running threads.

Rely/guarantee seems to be an appropriate technique for possibly unsynchronized programs, i.e., threads that may interfere with each other. In the rely/guarantee approach, we precisely describe the interference on our 'own' thread by the environment. On the semantical level, we make this interleaved interference explicit. In contrast, the approaches by Owicki and Gries [1976] as well as those based on concurrent separation logic (see Sect. 10.3.2) do not consider interleavings, but treat concurrent programs like sequential.

Our approach is in a sense similar to the combination of rely/guarantee and separation logic by Vafeiadis and Parkinson [2007]. Both combine the advantages of classical, i.e., purely functional, rely/guarantee with those of techniques for memory separation. Both allow to dynamically determine the parts of the heap that are specified to be separated. In place of the implicit separation properties of concurrent separation logic and related approaches, we use explicit frame annotations based on the dynamic frame approach. This offers more flexibility and verbosity (cf. [Weiß, 2011; Mostowski, 2015]).

Information flow analysis based on theorem proving is still a heavyweight approach and can be time-consuming, especially for the human effort provided by a proof engineer, since they describe theoretically undecidable problems. This effectively limits the ability to apply the approach to larger programs or programs that are hard to specify on the code level. The latter category particularly includes programs that use cryptography. Analyzing such programs can only be made feasible through a combination of complementary techniques, as discussed in Chap. 9. A symbolic approach to cryptography, using ideal functionalities, can be used to prove cryptographic indistinguishability.

In developing an information flow analysis for concurrent programs, the biggest challenge turned out to be the definition of an appropriate concurrency model and a programing language supporting it. Surprisingly, the definition of security in sequential programs can be leveraged to concurrent programs in a natural way. The only difference is that the security of concurrent programs can depend on the scheduler. This forces us to classify some programs as

insecure even though high variables do not even occur in the program context. The reason for this is that we cannot exclude the case in which a scheduler encodes secrets in a schedule. This issue is not particular to our approach, but also appears in LSOD as mentioned by Snelting [2015]. One solution to this would be to specify that the program is only executed under schedulers that do not depend on secrets.

Quantitative analyses based on our framework can be developed. The tool chain described by Klebanov [2014] combines a qualitative information flow analyzer with model generation and model counting. A precise analysis is necessary to generate models. Incomplete analyzers cannot be used to produce reliable results.

Certificates allow the results of security analyses to be archived persistently. The Common Criteria (CC) is an established scheme for software and hardware security certification [CC]. Software certification is commonly seen as a powerful means to achieve high quality software—particularly if it is employed in a security critical context. Improved availability of comprehensive product certification in software engineering would have considerable advantages. In particular, software engineers have been promoting a paradigm shift from *process* to *product* oriented certification for a long time (cf. [Voas, 1998; Meyer, 2003]). However, the CC are not directly concerned with software verification. Whether the program code actually conforms to the specified security functionality described in the documentation is never proven formally. Beckert, Bruns, and Grebing [2012a] showed that it is possible in principle to use code level specifications as development models in the CC. Since they are not required, adequacy in real world certification will depend on the evaluator. The standard itself tends to be very vague on *how* formal methods are used.

Formal verification is no "silver bullet" in quality software development [Brooks, 1987]. Firstly, verification only proves correctness w.r.t. a given specification, which may or may not reflect the actual intention. Verification must be considered as just one piece of the puzzle. Testing—as the dual to verification—is still necessary (cf. Knuth's famous quote from 1977). Secondly, software itself needs to comply with analysis techniques, too. Software needs to meet at least the quality standards to be accessible at all. In this regard, high-level languages with strict structuring and dedicated features are more amenable to verification.

Much work in proving secure information flow is dedicated to establishing fundamental *global* invariants in order to verify *local* information flow properties. Dynamic analysis techniques, such as runtime checking or testing, can be added to the setup. Gladisch [2010] has demonstrated how unit test

cases can be derived from open proof goals. Complete analysis techniques allow for counter example generation.

We have experienced that concrete proof obligations (for both thread specification validity and secure information flow) tend to be large and possibly incomprehensible to a user of the verification system. It is thus highly important to provide a specification interface as we did with the JML extensions, that were presented in Chap. 8. JML has the advantage that it is already tuned towards specifying sequential Java. Since it does not yet contain any provisions for neither concurrency nor secure information flow, our proposals are candidates for inclusion in the official reference.

11.3 Outlook on Future Work

Completeness of reasoning about concurrent programs is hard to attain. While the original rely/guarantee approach has been proven complete by Stølen [1990], we did not provide a completeness proof. Even worse, our calculus definitely is incomplete w.r.t. the semantics defined in Sect. 3.5: it is not possible to assert that all threads have terminated. Given this situation, there are two possible future directions: 1. reïnforcing our verification framework to make it complete or 2. dropping the fairness assumption in Sect. 3.1.2 and relaxing proof obligations appropriately. The latter option seems more promising. On the one hand, as already discussed in Sect. 5.4.4, formally attaining completeness is a futile endeavour. On the other hand, we have introduced fairness into concurrent program semantics only as an auxiliary feature, that ensures that thread executions terminate if the corresponding sequential program terminates locally. While this premiss helps to keep definitions relatively simple, it is not entirely realistic to assume it. Further investigations into this area should be driven by practical applications, e.g., the example in Sect. 5.6 should be provable w.r.t. a stronger postcondition that prescribes the termination of all involved threads.

Synchronization of concurrent threads is an essential means to develop programs that make use of concurrency in a sensible way. In Sect. 7.2, we discuss the instruments that are specific to Java. As explained in Sect. 5.6, incorporating synchronization means would require some modifications both to the theoretical definitions in this dissertation and to the implementation in KeY. Stølen [1991] extends the basic rely/guarantee approach by introducing 'wait conditions' to specify program behavior at synchronization points. These conditional wait statements with atomic evaluation are similar to loop invariants since they describe what is maintained while waiting. A related issue are *liveness* properties of a concurrent systems as a whole. Since classical rely/guarantee lacks a notion of progress, Gotsman et al. [2009] replace the two-state assertions with temporal formulae in guarantees and

rely conditions. By incorporating a way to reason about progress, we could also relax the fairness assumption imposed on schedulers.

Effective concurrency verification can be established in a combination of the rely/guarantee approach with approaches that establish a stricter separation. The approach in Chap. 5 is appropriate for a general case, where fine-grained separation properties are needed. In Java, there is no language support for atomic blocks. If we could identify such blocks that may be treated as if they were atomic, then we could reduce the proof complexity. More effective approaches rely on a broader separation of memory, such as the approaches based on separation logic (cf. Sect. 10.3.2) or the approach by Mostowski [2015]. This allows that concurrent changes only need to be considered at certain places, in which we can apply rely/guarantee. Since Mostowski's approach is based on KeY, too, it appears natural to combine it with ours.

The implementation in KeY of the rely/guarantee approach for multi-threaded Java is still in a prototype state. It seems very promising to bring it forward. So far, proof obligations for guarantees as well as rules for interleavings have been implemented. The next steps include: 1. support for dynamic thread creation; 2. support for synchronization; 3. reïntegrating loop invariant and method contract rules; 4. adapting the rules and proof obligations to Java-specific issues, e.g., exceptions; and 5. generating proof obligations for secure information flow. We expect that more case studies will be available while the implementation develops. In particular, we aim to verify a version of the e-voting system described in Chap. 9 that includes parallel vote tallying. Once we have come to a more stable system, we may also consider the practical improvements to information flow analysis described by Scheben [2014]. Finally, we also need to consider practical aspects like scalability and usability.

Combining different techniques for secure information flow is vital in order to analyze more realistic programs. Formally proving secure information flow has a high specification overhead and a slow performance in practice. On the other hand, the success of imprecise static or dynamic analysis techniques (for both functional and relational properties) in practice reveals that there are many 'typical' situations in which these properties are efficiently provable. For this reason, it is promising to combine approaches from both worlds. A first idea has been described by Küsters et al. [2015], that combines functional verification with less precise static information flow analysis.

The collaboration of different techniques and tools needs to be placed upon a sound foundation. A remaining goal will be to develop a framework to combine the KeY and JOANA tools for a more tractable information flow

analysis. The basic idea is to analyze information flow with KeY only in modules that cannot be analyzed precisely with JOANA. We are aiming at a meta-level theorem that establishes the overall soundness and completeness of the approach. A further technical challenge lies in defining the exact interface between both tools. The development of RIFL and the collaboration so far within the RS^3 project have pointed out some starting points.

Going in the opposite direction of the tool chain, JOANA provides static *program slicing*. The knowledge obtained about a target program could be used to simplify the proof obligation by (virtually) slicing away unrelated program parts. While this approach breaks modularity of the analysis, the slicing criteria guarantee that it is sound. As a variation, static slicing could be replaced by *semantic* slicing [Liu et al., 2015], which combines the strengths of static and dynamic slicing.

The influence of schedulers on security is still an open issue. Our own security property from Sect. 6.3 considers programs insecure in which the 'low' output depends on the scheduler. This can be seen as restrictive (cf. [Snelting, 2015]). Yet, there is no satisfactory modular analysis approach that is tolerant to this situation. Explicit modeling of a scheduler (as by Russo and Sabelfeld [2006]) is not an acceptable solution since it breaks modularity. A possible direction of future research could include a relaxation of the rely/guarantee framework to include *free* rely conditions, that can be assumed, but need not to be proven. We would then encode scheduler properties in those. The overall soundness of such an approach could be established on the meta level, possibly in a hybrid approach.

Assuming noninterfering schedulers would also be an important step towards a parallel *compositional* notion of secure information flow. Even with strong requirements on intermediate data flows, we need to ensure that the composition is secure. We expect that the analysis of compositional properties can be easier than the current approach, in which we consider the entire functional effect of interleavings. This may not be necessary in every case, as the approach by Mantel et al. [2011] shows.

Relational rely/guarantee specifications are a possibility to achieve a precise *and* compositional analysis technique for secure information flow in concurrent programs. The basic idea is to consider interleavings as we did in Chap. 5, but instead of functional (and frame) specification, we specify thread behavior in terms of information flow. These specification statements are of the shape 'on every atomic step, no information flows from `H` to `L`.'

Bibliography

Martín Abadi. Protection in programming-language translations. In *25th International Colloquium on Automata, Languages and Programming (ICALP '98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 868–883, Aalborg, Denmark, July 1998. Springer-Verlag, Berlin Germany. (Cited on page 18.)

Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995. URL `http://www.acm.org/pubs/toc/Abstracts/0164-0925/201069.html`. (Cited on pages 102 and 240.)

Martín Abadi and Zohar Manna. Nonclausal temporal deduction. In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 1–15, Brooklyn, NY, June 1985. Springer. ISBN 3-540-15648-8. (Cited on page 229.)

Martín Abadi and Zohar Manna. Nonclausal deduction in first-order temporal logic. *Journal of the ACM*, 37(2):279–317, April 1990. (Cited on page 229.)

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 147–160. ACM Press, 1999. (Cited on page 242.)

Martín Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst*, 28(2):207–255, 2006. (Cited on pages 168 and 227.)

Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. An assertion-based proof system for multithreaded Java. *Theor. Com-*

put. Sci., 331(2-3):251–290, 2005. URL http://dx.doi.org/10.1016/j.tcs.2004.09.019. (Cited on page 234.)

Erika Ábrahám, Frank S. de Boer, Willem P. de Roever, and Martin Steffen. A deductive proof system for multithreaded Java with exceptions. *Fundamenta Informaticae*, 82(4):391–463, 2008. URL http://content.iospress.com/articles/fundamenta-informaticae/fi82-4-05. (Cited on page 234.)

Karl R. Abrahamson. Modal logic of concurrent nondeterministic programs. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 21–33. Springer, 1979. ISBN 3-540-09511-X. URL http://dx.doi.org/10.1007/BFb0022461. (Cited on page 233.)

Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996. (Cited on page 192.)

Ben Adida. Helios: Web-based open-audit voting. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 335–348. USENIX Association, 2008. ISBN 978-1-931971-60-7. (Cited on page 224.)

Johan Agat. Transforming out timing leaks. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, Boston, Massachusetts, January 19–21, 2000. (Cited on page 247.)

Wolfgang Ahrendt and Maximilian Dylla. A verification system for distributed objects with asynchronous method calls. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, volume 5885 of *Lecture Notes in Computer Science*, pages 387–406. Springer, 2009. ISBN 978-3-642-10372-8. (Cited on page 236.)

Wolfgang Ahrendt and Maximilian Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programing*, 77(12): 1289–1309, 2012. (Cited on page 236.)

Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005. (Cited on page 161.)

Wolfgang Ahrendt, Frank S. de Boer, and Immo Grabe. Abstract object creation in dynamic logic. In Ana Cavalcanti and Dennis Dams, editors,

FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2–6, 2009. Proceedings, volume 5850 of *Lecture Notes in Computer Science*, pages 612–627. Springer, 2009. ISBN 978-3-642-05088-6. URL http://dx.doi.org/10.1007/978-3-642-05089-3. (Cited on page 44.)

Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, number 8471 in Lecture Notes in Computer Science, pages 1–17. Springer-Verlag, 2014. ISBN 978-3-642-54107-0. doi: 10.1007/978-3-319-12154-3_4. URL http://link.springer.com/chapter/10.1007/978-3-319-12154-3_4. (Cited on pages 9, 15, 32, and 162.)

Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Vladimir Klebanov, and Peter H. Schmitt, editors. *The KeY Book: Deductive Software Verification in Practice*. Lecture Notes in Computer Science. Springer, 2016. In preparation. (Cited on pages iii, 162, 274, and 285.)

Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román-Díez. Verified resource guarantees using COSTA and KeY. In *Proc. ACM SIGPLAN 2011 Workshop on Partial Evaluation and Program Manipulation (PEPM'11), Austin, Texas, USA*. ACM Press, 2011. (Cited on page 162.)

Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002. URL http://doi.acm.org/10.1145/585265.585270. (Cited on pages 56 and 229.)

Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. The VerCors project. Setting up basecamp. In Koen Claessen and Nikhil Swamy, editors, *Programming Languages meets Program Verification (PLPV 2012)*, 2012. URL http://wwwhome.ewi.utwente.nl/~marieke/vercors-basecamp.pdf. (Cited on page 232.)

Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Verification of concurrent systems with VerCors. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer, editors, *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, volume 8483 of *Lecture Notes in Computer Science*, pages 172–216. Springer, 2014.

ISBN 978-3-319-07316-3. URL http://dx.doi.org/10.1007/978-3-319-07317-0. (Cited on pages 9 and 238.)

Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2004. (Cited on pages 29, 118, and 241.)

Torben Amtoft and Anindya Banerjee. Verification condition generation for conditional information flow. In Peng Ning, Vijay Atluri, Virgil D. Gligor, and Heiko Mantel, editors, *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, pages 2–11, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-887-9. (Cited on page 242.)

Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Proceedings POPL*, pages 91–102. ACM, 2006. (Cited on pages 25 and 248.)

Henrik Reif Andersen. A polyadic modal μ-calculus. Technical Report 145, Danmarks Tekniske Universitet, 19 February 1994. (Cited on page 245.)

Ross Anderson. *Security engineering: a guide to building dependable distributed systems*. John Wiley and Sons, Inc., second edition, 2008. ISBN 0-470-06852-3. URL http://www.cl.cam.ac.uk/~rja14/book.html. (Cited on pages 17 and 250.)

Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998. URL http://www.aw.com/cp/javaseries.html. (Cited on page 3.)

Edward A. Ashcroft. Proving assertions about parallel programs. *J. Comp. Sys. Sci.*, 10:110–135, 1975. (Cited on page 88.)

Edward A. Ashcroft and Zohar Manna. Formalization of properties of parallel programms. *Machine Intelligence*, 6:17–41, 1971. (Cited on page 88.)

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In Sushil Jajodia and Javier López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2008. ISBN 978-3-540-88312-8. URL http://dx.doi.org/10.1007/978-3-540-88313-5_22. (Cited on page 23.)

Philippe Balbani and Tinko Tinchev. Definability and computability for PRSPDL. In Rajeev Goré and Agi Kurucz, editors, *Advances in Modal Logic*. College Publications, 2014. (Cited on page 57.)

Musard Balliu, Mads Dam, and Gurvan Le Guernic. Epistemic temporal logic for information flow security. In Aslan Askarov and Joshua D. Guttman, editors, *Proceedings of the 2011 Workshop on Programming Languages and Analysis for Security, PLAS 2011, San Jose, CA, USA, 5 June, 2011*, page 6. ACM, 2011. ISBN 978-1-4503-0830-4. URL `http://dl.acm.org/citation.cfm?id=2166956`. (Cited on page 242.)

Michael F. Banahan, Declan Brady, and Mark Doran. *The C book – featuring the ANSI C standard*. Addison-Wesley, 2 edition, 1991. ISBN 978-0-201-54433-6. (Cited on page 151.)

Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proceedings CSFW*, 2002. (Cited on pages 148, 154, and 248.)

Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, 2005. (Cited on page 28.)

Anindya Banerjee, Mike Barnett, and David A. Naumann. Boogie meets regions: A verification experience report. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 177–191, Berlin Heidelberg, 2008a. Springer. ISBN 978-3-540-87872-8. doi: 10.1007/978-3-540-87873-5_16. URL `http://dx.doi.org/10.1007/978-3-540-87873-5_16`. (Cited on page 233.)

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 339–353. IEEE, 2008b. (Cited on page 29.)

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Local reasoning for global invariants, part I: Region logic. *Journal of the ACM*, 60(3):18, 2013. (Cited on page 232.)

Michael Bär. Analyse und Vergleich verifizierbarer Wahlverfahren. Diplomarbeit, Fakultät für Informatik, Karlsruhe Institute of Technology, 2008. (Cited on page 250.)

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference*

Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005. URL `http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3362&spage=151`. (Cited on pages 38, 231, and 239.)

Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007. ISBN 978-3-540-73367-6. (Cited on page 164.)

Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund Clarke, Tom Henzinger, and Helmut Veith, editors, *Handbook of Model Checking*. Springer, 2014. (to appear). (Cited on pages 8 and 164.)

Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010. (Cited on page 164.)

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. ISBN 978-3-642-22109-5. (Cited on page 164.)

Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass – Java with assertions. *Electr. Notes Theor. Comput. Sci*, 55(2):103–117, 2001. URL `http://dx.doi.org/10.1016/S1571-0661(04)00247-6`. (Cited on page 231.)

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, CSFW-17, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004. (Cited on pages 24, 29, 118, 123, 128, 138, 242, and 245.)

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *ACM SIGPLAN Notices*, 44(1):90–101, January 2009. ISSN 0362-1340. doi: 10.1145/1480881.1480894. (Cited on page 250.)

Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors,

FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011. (Cited on pages 29, 138, and 242.)

Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*, volume 7734 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2013a. ISBN 978-3-642-35721-3; 978-3-642-35722-0. URL `http://dx.doi.org/10.1007/978-3-642-35722-0`. (Cited on page 242.)

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst*, 35(3):9, 2013b. (Cited on page 250.)

Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference Java bytecode verifier. *Mathematical Structures in Computer Science*, 23(5):1032–1081, 2013c. ISSN 1469-8072. doi: 10.1017/S0960129512000850. URL `http://journals.cambridge.org/article_S0960129512000850`. (Cited on pages 28 and 248.)

Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. Probabilistic relational verification for cryptographic implementations. *SIGPLAN Not.*, 49(1):193–205, January 2014. ISSN 0362-1340. doi: 10.1145/2578855.2535847. URL `http://doi.acm.org/10.1145/2578855.2535847`. (Cited on page 250.)

Jon Barwise and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-wellfounded Phenomena*. Cambridge University Press, 1996. (Cited on page 198.)

Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2010. URL `http://frama-c.cea.fr/acsl.html`. Version 1.5. (Cited on page 231.)

Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput*, 20(3):651–674, 2010. (Cited on page 70.)

Tobias Beck. Verifizierbar korrekte Implementierung von Bingo Voting. Studienarbeit, Fakultät für Informatik, Karlsruhe Institute of Technology, March 2010. (Cited on page 250.)

Steffen Becker, Heiko Koziolek, and Ralf H. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009. URL http://dx.doi.org/10.1016/j.jss.2008.03.066. (Cited on page 249.)

Bernhard Beckert. A dynamic logic for Java Card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, pages 111–119, 2000. (Cited on pages 57 and 179.)

Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *Lecture Notes in Computer Science*, pages 6–24. Springer, 2001. (Cited on pages 5, 38, 40, 53, 57, 85, and 161.)

Bernhard Beckert and Daniel Bruns. Formal semantics of model fields inspired by a generalization of Hilbert's ε terms. In Wolfgang Ahrendt and Richard Bubel, editors, *10th KeY Symposium*, Nijmegen, the Netherlands, 26–27 August 2011. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000024829. Extended Abstract. (Cited on page 15.)

Bernhard Beckert and Daniel Bruns. Formal semantics of model fields in annotation-based specifications. In Birte Glimm and Antonio Krüger, editors, *KI 2012: Advances in Artificial Intelligence*, volume 7526 of *Lecture Notes in Computer Science*, pages 13–24. Springer-Verlag, 2012a. ISBN 978-3-642-33346-0. URL http://link.springer.com/chapter/10.1007/978-3-642-33347-7_2. (Cited on page 14.)

Bernhard Beckert and Daniel Bruns. Dynamic trace logic: Definition and proofs. Technical Report 2012-10, Department of Informatics, Karlsruhe Institute of Technology, 2012b. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028184. A revised version replacing an unsound rule is available at http://formal.iti.kit.edu/~bruns/papers/trace-tr.pdf. (Cited on pages iii, 12, 13, 59, 70, 77, 78, and 129.)

Bernhard Beckert and Daniel Bruns. Dynamic logic with trace semantics. In Maria Paola Bonacina, editor, *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 315–329. Springer-Verlag, 2013. ISBN 978-3-642-38573-5. doi: 10.1007/978-3-642-38574-2_22. URL http://link.springer.com/chapter/10.1007/978-3-642-38574-2_22. (Cited on pages iii, 12, 13, 32, 33, 34, 35, 36, 37, 42, 43, 44, 53, 54, 58, 59, 62, 66, 69, 72, 77, 86, 100, and 171.)

Bernhard Beckert and Reiner Hähnle. Reasoning and verification. *IEEE Intelligent Systems*, 2014. to appear. (Cited on page 8.)

Bernhard Beckert and Vladimir Klebanov. Must program verification systems and calculi be verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, 2006. (Cited on page 179.)

Bernhard Beckert and Vladimir Klebanov. A dynamic logic for deductive verification of multi-threaded programs. *Formal Aspects of Computing*, 25 (3):405–437, 2013. ISSN 0934-5043. (Cited on pages 34, 35, 63, and 234.)

Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, volume 2083 of *Lecture Notes in Computer Science*, pages 626–641. Springer, 2001. (Cited on pages 78 and 228.)

Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence. (Cited on pages 164 and 175.)

Bernhard Beckert, Reiner Hähnle, C. A. R. Hoare, Douglas R. Smith, Cordell Green, Silvio Ranise, Cesare Tinelli, Thomas Ball, and Sriram K. Rajamani. Intelligent systems and formal methods in software engineering. *IEEE Intelligent Systems*, 21(6):71–81, November/December 2006. URL `http://doi.ieeecomputersociety.org/10.1109/MIS.2006.117`. (Cited on page 8.)

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2007a. (Cited on pages 5, 92, 95, 101, 147, 162, 182, 269, 301, and 307.)

Bernhard Beckert, Vladimir Klebanov, and Steffen Schlager. Dynamic logic. In Beckert et al. [2007a], chapter 3, pages 69–178. (Cited on pages 38, 42, 44, 51, 53, 57, 85, 108, 149, 152, 159, 168, and 241.)

Bernhard Beckert, Daniel Bruns, and Sarah Grebing. Mind the gap: Formal verification and the Common Criteria. In Markus Aderhold, Serge Autexier, and Heiko Mantel, editors, *6th International Verification Workshop, VERIFY-2010*, volume 3 of *EPiC Series*, pages 4–12. EasyChair, 2012a. URL `http://easychair.org/publications/?page=1489979161`. (Cited on pages 15 and 256.)

Bernhard Beckert, Daniel Bruns, Ralf Küsters, Christoph Scheben, Peter H. Schmitt, and Tomasz Truderung. The KeY approach for the

cryptographic verification of Java programs: A case study. Technical Report 2012-8, Department of Informatics, Karlsruhe Institute of Technology, 2012b. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000027497. (Cited on pages 15, 128, 130, and 211.)

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Information flow in object-oriented software – extended version –. Technical Report 2013-14, Department of Informatics, Karlsruhe Institute of Technology, 2013a. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000037606. (Cited on pages iii and 14.)

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Secure information flow for Java – a dynamic logic approach. Technical Report 2013-10, Department of Informatics, Karlsruhe Institute of Technology, 2013b. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000036786. (Cited on pages iii, 14, 59, 63, 64, 138, 139, 196, and 198.)

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Information flow in object-oriented software. In Gopal Gupta and Ricardo Peña, editors, *Logic-Based Program Synthesis and Transformation, LOPSTR 2013*, number 8901 in Lecture Notes in Computer Science, pages 19–37. Springer, 2014. ISBN 978-3-319-14125-1. doi: 10.1007/978-3-319-14125-1_2. (Cited on pages iii, 14, 26, 122, 147, 148, 153, 154, 155, 156, 157, 158, and 242.)

Bernhard Beckert, Vladimir Klebanov, and Mattias Ulbrich. Regression verification for Java using a secure information flow calculus. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*, FTfJP '15, pages 6:1–6:6. ACM, 2015. doi: 10.1145/2786536.2786544. URL http://doi.acm.org/10.1145/2786536.2786544. (Cited on page 162.)

D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, U. S. Air Force Electronic Systems Division, November 1973. (Cited on page 27.)

Marco Benedetti and Alessandro Cimatti. Bounded model checking for past LTL. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2003. ISBN 3-540-00898-5. URL http://dx.doi.org/10.1007/3-540-36577-X_3. (Cited on page 56.)

Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 14–25. ACM, 2004. ISBN 1-58113-729-X. URL `http://dl.acm.org/citation.cfm?id=964001`. (Cited on pages 29 and 250.)

Lennart Beringer and Martin Hofmann. Secure information flow and program logics. In *Computer Security Foundations*, pages 233–248, 2007. (Cited on page 248.)

Marc Beuter, Benedict Hauck, Manuel Kohnen, Florian Pohl, Fedor Scholz, and Jürgen Schuck. Fährmann – Automatisches Prüfen von Programmeigenschaften, 2013. URL `https://github.com/bananen/faehrmann`. (Cited on page 29.)

Dirk Beyer. Software verification and verifiable witnesses – (report on SV-COMP 2015). In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 2015. ISBN 978-3-662-46680-3. URL `http://dx.doi.org/10.1007/978-3-662-46681-0`. (Cited on pages 227 and 230.)

K. J. Biba. Integrity considerations for secure computer systems. Technical Report 3153, Mitre, Bedford, MA, April 1977. (Cited on page 18.)

Leyla Bilge and Tudor Dumitras. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 833–844, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382284. URL `http://doi.acm.org/10.1145/2382196.2382284`. (Cited on page 7.)

Patrick Blackburn. Representation, reasoning, and relational structures: a hybrid logic manifesto. *Logic Journal of IGPL*, 8:336–365, 2000. (Cited on page 229.)

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207. ACM, 2003. URL `http://doi.acm.org/10.1145/781131.781153`. (Cited on page 235.)

Mike Bland. Finding more than one worm in the apple. *Communications of the ACM*, 57(7):58–64, July 2014. ISSN 0001-0782 (print), 1557-7317 (electronic). doi: http://dx.doi.org/10.1145/2622630. (Cited on page 2.)

Jens-Matthias Bohli, Christian Henrich, Carmen Kempka, Jörn Müller-Quade, and Stefan Röhrich. Enhancing electronic voting machines on the example of Bingo voting. *IEEE Transactions on Information Forensics and Security*, 4(4):745–750, 2009. (Cited on pages 224 and 250.)

Alexander Borgida, John Mylopoulos, and Raymond Reiter. "... And nothing else changes": The frame problem in procedure specifications. In Victor R. Basili, Richard A. DeMillo, and Takuya Katayama, editors, *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993*, pages 303–314. IEEE Computer Society / ACM Press, 1993. ISBN 0-89791-588-7. URL `http://dl.acm.org/citation.cfm?id=257572`. (Cited on pages 89 and 200.)

Thorsten Bormer. *Advancing Deductive Program-Level Verification for Real-World Application – Lessons Learned from an Industrial Case Study*. PhD thesis, Karlsruhe Institute of Technology, 23 October 2014. (Cited on pages 9 and 232.)

Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen, and Mattias Ulbrich. The COST IC0701 verification competition 2011. Technical report, COST IC0701, 2012. URL `http://docs.google.com/viewer?a=v&pid=sites&srcid=Y29zdC1pYzA3MDEub3JnfGZvdmVvb3MyMDExfGd4Ojc1YzdjOWE1Yjk2Yjc4YjM`. (Cited on page 166.)

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 259–270. ACM, 2005. ISBN 1-58113-830-X. URL `http://dl.acm.org/citation.cfm?id=1040305`. (Cited on page 237.)

Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002. (Cited on page 28.)

John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis (SAS)*, volume 2694 of *Lecture Notes*

in Computer Science, pages 55–72, Berlin, 2003. Springer-Verlag. URL http://dx.doi.org/10.1007/3-540-44898-5_4. (Cited on page 237.)

Frederick P. Brooks, Jr. No silver bullet: essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987. (Cited on page 256.)

Kai Brünnler and Martin Lange. Cut-free sequent systems for temporal logic. *J. Log. Algebr. Program*, 76(2):216–225, 2008. (Cited on page 229.)

Daniel Bruns. Elektronische Wahlen: Theoretisch möglich, praktisch undemokratisch. *FIfF-Kommunikation*, 25(3):33–35, September 2008. ISSN 0938-3476. (Cited on page 210.)

Daniel Bruns. Formal semantics for the Java Modeling Language. Diplomarbeit, Universität Karlsruhe, June 2009. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000012399. (Cited on page 184.)

Daniel Bruns. Specification of red-black trees: Showcasing dynamic frames, model fields and sequences. In Wolfgang Ahrendt and Richard Bubel, editors, *10th KeY Symposium*, Nijmegen, the Netherlands, 26–27 August 2011. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000024828. Extended Abstract. (Cited on pages 15 and 232.)

Daniel Bruns. Eine formale Semantik für die Java Modeling Language. *Informatik-Spektrum*, 35(1):45–49, 2012. doi: 10.1007/s00287-011-0532-0. URL http://www.springerlink.com/content/b503j663353x482w/. (Cited on page 15.)

Daniel Bruns. Formal verification of an electronic voting system. Technical Report 2014-11, Department of Informatics, Karlsruhe Institute of Technology, August 2014a. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000042284. (Cited on pages iii and 15.)

Daniel Bruns. Towards specification and verification of information flow in concurrent Java-like programs. Technical Report 2014-5, Department of Informatics, Karlsruhe Institute of Technology, March 2014b. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000039446. (Cited on pages iii, 14, 86, 138, 143, and 247.)

Daniel Bruns. Deductive verification of concurrent programs. Technical Report 2015-3, Department of Informatics, Karlsruhe Institute of Technology, February 2015a. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000045641. (Cited on pages iii, 12, 13, and 203.)

Daniel Bruns. A theorem proving approach to secure information flow in concurrent programs (extended abstract). In Deepak Garg and Boris Köpf, editors, *Workshop on Foundations of Computer Security (FCS 2015)*, July

2015b. URL http://software.imdea.org/~bkoepf/FCS15/paper3.pdf. (Cited on page 14.)

Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. Verification of software product lines with delta-oriented slicing. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 2011. ISBN 978-3-642-18069-9. doi: 10.1007/978-3-642-18070-5_5. URL http://www.springerlink.com/content/441476732611n21t/. (Cited on page 15.)

Daniel Bruns, Huy Quoc Do, Simon Greiner, Mihai Herda, Martin Mohr, Enrico Scapin, Tomasz Truderung, Bernhard Beckert, Ralf Küsters, Heiko Mantel, and Richard Gay. Poster: Security in e-voting. In Sophie Engle, editor, *36th IEEE Symposium on Security and Privacy, Poster Session*, 18 May 2015a. (Cited on pages 15, 210, 222, 224, and 254.)

Daniel Bruns, Wojciech Mostowski, and Mattias Ulbrich. Implementation-level verification of algorithms with KeY. *Software Tools for Technology Transfer*, 17(6):729–744, November 2015b. ISSN 1433-2779. doi: 10.1007/s10009-013-0293-y. URL http://link.springer.com/article/10.1007/s10009-013-0293-y. (Cited on pages iii, 15, 162, and 166.)

Richard Bubel. *Formal Verification of Recursive Predicates*. PhD thesis, Universität Karlsruhe, 2007. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000008366. (Cited on pages 162 and 166.)

Richard Bubel and Peter H. Schmitt. *Theories*, chapter 5. In Ahrendt et al. [2016], 2016. In preparation. (Cited on pages 66 and 76.)

Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract interpretation of symbolic execution with explicit state updates. In Frank de Boer, Marcello M. Bonsangue, and Eric Madelaine, editors, *Post Conf. Proc. 6th International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 5751 of *Lecture Notes in Computer Science*, pages 247–277. Springer-Verlag, 2009. (Cited on page 28.)

Richard Bubel, Huy Quoc Do, Reiner Hähnle, and Nathan Wasser. Automated deductive information flow analysis. Presentation at RS^3 annual meeting, 8 October 2014. (Cited on page 243.)

J. Richard Büchi. On a decision method in restricted second-order arithmetic. In *Proc. Int. Congr. for Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford Univ. Press, 1962. (Cited on page 56.)

Marco Carvalho, Jared DeMott, Richard Ford, and David A. Wheeler. Heartbleed 101. *IEEE Security & Privacy*, 12(4):63–67, 2014. URL http://dx.doi.org/10.1109/MSP.2014.66. (Cited on page 1.)

Antonio Cau, Ben Moszkowski, and Hussein Zedan. Interval temporal logic, 23 September 2002. URL http://www.cse.dmu.ac.uk/~cau/papers/itl homepage.pdf. (Cited on pages 56, 83, and 229.)

CC. *Common Criteria for Information Technology Security Evaluation*, version 3.1 release 4 edition, September 2012. (Cited on page 256.)

David Chaum, Richard T. Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L. Rivest, Peter Y. A. Ryan, Emily (Emily Huei-Yi) Shen, Alan T. Sherman, and Poorvi L. Vora. Scantegrity II: End-to-end verifiability by voters of optical scan elections through confirmation codes. *IEEE Transactions on Information Forensics and Security*, 4(4), October 2009. ISSN 1556-6013. (Cited on page 224.)

Stephen Chong and Andrew C. Myers. Security policies for downgrading. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick Drew McDaniel, editors, *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, pages 198–209. ACM, 2004. ISBN 1-58113-961-6. URL http://doi.acm.org/10.1145/1030083.1030110. (Cited on page 143.)

Andrey Chudnov, George Kuan, and David A. Naumann. Information flow monitoring as abstract interpretation for relational logic. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 48–62. IEEE, 2014. URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6954678. (Cited on page 30.)

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:354–363, 1936. (Cited on page 55.)

David Clark and Sebastian Hunt. Non-interference for deterministic interactive programs. In Pierpaolo Degano, Joshua D. Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, volume 5491 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2009. URL http://dx.doi.org/10.1007/978-3-642-01465-9. (Cited on page 247.)

David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. *Electr. Notes Theor. Comput. Sci*, 59 (3):238–251, 2001. (Cited on page 242.)

Dave Clarke, Stijn de Gouw, Reiner Hähnle, Einar Broch Johnsen, Ilham W. Kurnia, Radu Muschevici, Bjarte M. Østvold, José Proença, Ina Schaefer, Jan Schäfer, Martin Steffen, and Arild B. Torjusen. Full ABS modeling framework. Technical Report Deliverable D1.2, HATS project, March 1 2010. URL http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable12.pdf. (Cited on page 231.)

Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-Verlag. (Cited on page 56.)

Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst*, 8(2):244–263, 1986. URL `http://doi.acm.org/10.1145/5397.5399`. (Cited on page 56.)

Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. URL `http://dx.doi.org/10.3233/JCS-2009-0393`. (Cited on page 118.)

Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society, 2008. URL `http://doi.ieeecomputersociety.org/10.1109/SP.2008.32`. (Cited on pages 210, 224, and 250.)

Christina B. Class. Software-Qualität, Ethik und wir alle. *FIfF-Kommunikation*, 25(3):28–30, September 2008. ISSN 0938-3476. (Cited on page 2.)

Ellis S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977. (Cited on pages 5, 6, 24, 118, 123, 125, 128, 131, 136, 147, 207, 244, and 246.)

Ernie Cohen and Bert Schirmer. From total store order to sequential consistency: A practical reduction theorem. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2010. ISBN 978-3-642-14051-8. URL `http://dx.doi.org/10.1007/978-3-642-14052-5`. (Cited on page 240.)

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Berlin, August 2009. Springer-Verlag. (Cited on pages 9, 231, and 240.)

Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. Technical Report MSR-TR-2010-9, Microsoft Research, January 2010. URL `http://res`

earch.microsoft.com/apps/pubs/default.aspx?id=118664. (Cited on pages 231 and 240.)

David Cok. Adapting JML to generic types and Java 1.6. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, number CS-TR-08-07 in Technical Report, pages 27–34, 2008. (Cited on page 184.)

David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005. URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3362&spage=151. (Cited on page 249.)

Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput*, 17(4):807–841, 2007. URL http://www.cs.ncl.ac.uk/research/pubs/trs/papers/1029.pdf. (Cited on pages 89, 92, 234, and 235.)

Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 277–308. The MIT Press, 2000. ISBN 978-0-262-16188-6. (Cited on page 235.)

Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference record of third annual ACM symposium on theory of Computing*, pages 151–158, Shaker Heights, Oh., 1971. ACM. (Cited on page 18.)

Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7:70–90, 1978. (Cited on page 77.)

James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, New York, NY, June 2000. ACM Press. (Cited on page 230.)

Véronique Cortier. Electronic voting: how logic can help. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning – 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19–22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer*

Science, pages 16–25. Springer, 2014. ISBN 978-3-319-08586-9. URL http://dx.doi.org/10.1007/978-3-319-08587-6. (Cited on page 210.)

Véronique Cortier and Steve Kremer. Formal models and techniques for analyzing security protocols: A tutorial. *Foundations and Trends in Programming Languages*, 1(3):151–267, 2014. doi: 10.1561/2500000001. URL http://www.loria.fr/~skremer/Papers/CK-fntpl-14.pdf. (Cited on page 250.)

Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning*, 46(3-4):225–259, April 2010. doi: 10.1007/s10817-010-9187-9. URL http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/CKW-jar10.pdf. (Cited on page 250.)

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Language, Los Angeles*, pages 238–252. ACM Press, New York, January 1977. (Cited on pages 235 and 243.)

CVE. Common vulnerabilities and exposures, 2015. URL http://cve.mitre.org/. Website (accessed 16/07/2015). (Cited on page 1.)

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 25–35, Austin, TX, USA, January 1989. ACM Press. ISBN 0-89791-294-2. (Cited on page 58.)

Mads Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, 11 April 1994. (Cited on page 56.)

Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005. (Cited on pages 29, 118, 128, and 241.)

M. Davis. *The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions*. Dover Pubns, 2004. (Cited on pages 10 and 55.)

Frank S. de Boer. A sound and complete shared-variable concurrency model for multi-threaded Java programs. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *Formal Methods for Open Object-Based*

Distributed Systems (9th FMOODS'07), volume 4468 of *Lecture Notes in Computer Science (LNCS)*, pages 252–268. Springer-Verlag (New York), Paphos, Cyprus, June 2007. (Cited on page 234.)

Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. Proof pearl: The KeY to correct and stable sorting. *Journal of Automated Reasoning*, 53(2): 129–139, 2014. URL `http://dx.doi.org/10.1007/s10817-013-9300-y`. (Cited on page 166.)

Stijn de Gouw, Jurriaan Rot, Frank de Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's `java.utils.collection.sort()` is broken: The good, the bad and the worst case. In Daniel Kroening and Corina Păsăreanu, editors, *27th International Conference on Computer Aided Verification (CAV 2015) July 18–24 2015, San Francisco*, Lecture Notes in Computer Science. Springer, 2015. (Cited on page 166.)

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, 2008. Springer-Verlag. (Cited on page 164.)

Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001. (Cited on pages 4, 8, 32, 49, 88, 182, and 234.)

Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976. (Cited on page 19.)

Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977. (Cited on page 27.)

David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center (SRC), 1998. (Cited on page 29.)

David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London Ser. A*, A400:97–117, 1985. (Cited on page 18.)

Deutscher Bundestag. *Siebter Zwischenbericht der Enquete-Kommission „Internet und digitale Gesellschaft" – Demokratie und Staat*. Deutscher Bundestag, 6 February 2013. URL `http://dip21.bundestag.de/dip21/btd/17/122/1712290.pdf`. Drucksache 17/12290. (Cited on pages 2, 209, and 210.)

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975. (Cited on page 57.)

Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 169–185. Springer, 2012. ISBN 978-3-642-27939-3. URL `http://dx.doi.org/10.1007/978-3-642-27940-9`. (Cited on pages 86 and 247.)

Crystal Chang Din. *Verification Of Asynchronously Communicating Objects*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, March 2014. (Cited on pages 162 and 236.)

Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3): 227–256, 2012. (Cited on page 236.)

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer, 2010. ISBN 978-3-642-14106-5. (Cited on page 239.)

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. *SIGPLAN Notices*, 48(1):287–300, January 2013. ISSN 0362-1340. doi: 10.1145/2480359.2429104. URL `http://doi.acm.org/10.1145/2480359.2429104`. (Cited on page 241.)

Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In G. Castagna, editor, *Programming Languages and Systems*, number 5502 in Lecture Notes in Computer Science, pages 363–377. Springer-Verlag, 2009. (Cited on page 240.)

Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. (Cited on pages 18 and 212.)

Felix Dörre and Vladimir Klebanov. Pseudo-random number generator verification: A case study. In Arie Gurfinkel and Sanjit A. Seshia, editors, *Proceedings, Verified Software: Theories, Tools, and Experiments (VSTTE*

2015), Lecture Notes in Computer Science. Springer, 2015. To appear. (Cited on page 249.)

Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Verification of concurrent objects with asynchronous method calls. In *International Conference on Software – Science, Technology and Engineering. SwSTE '05*, pages 141–150. IEEE Computer Society, 2005. ISBN 0-7695-2335-8. URL `http://doi.ieeecomputersociety.org/10.1109/SWSTE.2005.24`. (Cited on pages 3, 33, 49, and 236.)

Mark Dowson. The ARIANE 5 sofware failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, March 1997. (Cited on page 7.)

Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008. URL `http://dx.doi.org/10.1109/TCAD.2008.923410`. (Cited on page 227.)

Guillaume Dufay, Amy Felty, and Stan Matwin. Privacy-sensitive information flow with JML. In Robert Nieuwenhuis, editor, *Automated Deduction (CADE-20)*, volume 3632 of *Lecture Notes in Computer Science*, pages 738–738. Springer, 2005. ISBN 978-3-540-28005-7. (Cited on page 249.)

Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. URL `http://yices.csl.sri.com/tool-paper.pdf`, 2006. (Cited on page 164.)

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press / ACM Press, 1999. URL `http://www.acm.org/pubs/articles/proceedings/soft/302405/p411-dwyer/p411-dwyer.pdf`. (Cited on page 230.)

Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*, volume 2. Springer, 1995. (Cited on page 97.)

Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer-Verlag, Berlin, 2nd edition, 1994. (Cited on page 54.)

E. Allen Emerson and A. Prasad Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, June 1984. (Cited on page 56.)

Sarah Ereth, Heiko Mantel, and Matthias Perner. Towards a common specification language for information flow security in RS^3 and beyond: RIFL 1.0 – The language. Technical Report TUD-CS-2014-0115, Technische Universität Darmstadt, 2014. (Cited on pages 183 and 207.)

Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis, Portland, OR*, pages 24–27. Jonathan Cook, May 2003. (Cited on page 29.)

Burkhard Ewert, Nermin Fazlic, and Johannes Kollbeck. E-Demokratie – Stand, Chancen und Risiken. In Christiane Schulzki-Haddouti, editor, *Bürgerrechte im Netz*, volume 382 of *Schriftenreihe*, pages 227–260. Bundeszentrale für politische Bildung, Bonn, 2003. ISBN 3-89331-458-X. (Cited on page 2.)

Ingo Feinerer and Gernot Salzer. A comparison of tools for teaching formal software verification. *Formal Asp. Comput*, 21(3):293–301, 2009. URL `http://dx.doi.org/10.1007/s00165-008-0084-5`. (Cited on page 166.)

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In Rocco De Nicola, editor, *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*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007. ISBN 978-3-540-71314-2. URL `http://dx.doi.org/10.1007/978-3-540-71316-6_13`. (Cited on page 239.)

Luminous Fennell and Peter Thiemann. LJGS: Gradual security types for object-oriented languages (extended abstract). In Deepak Garg and Boris Köpf, editors, *Workshop on Foundations of Computer Security (FCS 2015)*, July 2015. URL `http://software.imdea.org/~bkoepf/FCS15/paper11.pdf`. (Cited on page 28.)

Jean-Christophe Filliâtre. Deductive software verification. *Software Tools Technology Transfer (STTT)*, 13(5):397–403, 2011. URL `http://dx.doi.org/10.1007/s10009-011-0211-0`. (Cited on page 8.)

Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013. (Cited on page 9.)

Jean-Christophe Filliâtre, Andrei Paskevich, and Aaron Stump. The 2nd verified software competition: Experience report. In Bernhard Beckert, Armin Biere, Vladimir Klebanov, and Geoff Sutcliffe, editors, *1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE)*, 2012. (Cited on page 166.)

Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification – 26th International Conference, CAV 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2014. ISBN 978-3-319-08866-2. URL `http://dx.doi.org/10.1007/978-3-319-08867-9`. (Cited on pages 113 and 199.)

Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2): 194–211, April 1979. (Cited on page 56.)

Melvin Fitting. *First-order logic and automated reasoning (2nd ed.)*. Graduate texts in computer science. Springer, 1996. ISBN 978-0-387-94593-4. (Cited on page 54.)

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002. (Cited on pages 30 and 231.)

Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1):153–183, 2005. (Cited on page 235.)

Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society. (Cited on page 8.)

Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 341–350. ACM, 2011. ISBN 978-1-4503-0948-6. URL `http://doi.acm.org/10.1145/2046707.2046746`. (Cited on page 250.)

Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. (Cited on page 35.)

Nissim Francez and Amir Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978. (Cited on page 88.)

Hubert Garavel and Susanne Graf. Formal methods for safe and secure computer systems. Technical Report 875, Bundesamt für Sicherheit in der Informationstechnik, Bonn, 2013. URL `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/formal_methods_study_875/formal_methods_study_875.pdf?__blob=publicationFile`. (Cited on page 166.)

Deepak Garg, Jason Franklin, Dilsun Kirli Kaynar, and Anupam Datta. Compositional system security with interface-confined adversaries. *Electr. Notes Theor. Comput. Sci*, 265:49–71, 2010. URL `http://dx.doi.org/10.1016/j.entcs.2010.08.005`. (Cited on page 244.)

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39(2–3):176–210, 405–431, 1935. (Cited on page 66.)

Roberto Giacobazzi and Isabella Mastroeni. A proof system for abstract non-interference. *J. Log. Comput*, 20(2):449–479, 2010. URL `http://dx.doi.org/10.1093/logcom/exp053`. (Cited on page 243.)

Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, 2015. doi: 10.1007/s10207-014-0257-6. URL `http://dx.doi.org/10.1007/s10207-014-0257-6`. (Cited on pages 6, 25, 146, and 244.)

Christoph Gladisch. Test data generation for programs with quantified first-order logic specifications. In *Testing Software and Systems – 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Proceedings*, number 6435 in Lecture Notes in Computer Science, pages 158–173. Springer, 2010. (Cited on page 256.)

Christoph Gladisch and Shmuel Tyszberowicz. Specifying a linked data structure in JML for formal verification and runtime checking. In Leonardo de Moura and Juliano Iyoda, editors, *Brazilian Symposium on Formal Methods (SBMF)*, volume 8195 of *LNCS*. Springer, 2013. (Cited on page 232.)

Stephan Gocht. Refinement of path conditions for information flow analysis. Bachelor's thesis, Karlsruhe Institute of Technology, 2014. (Cited on page 28.)

Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38: 173–198, 1931. (Cited on pages 10 and 75.)

Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, Los Alamitos, CA, USA, 1982, 1982. (Cited on pages 24 and 123.)

Rajeev Goré. Tableau methods for modal and temporal logics. In Marcello D'Agostino, Dov Gabbay, Reiner Hähnle, and Joachim Posegga, editors, *Handbook of Tableau Methods*, pages 297–396. Kluwer Academic Publishers, Dordrecht, 1999. ISBN 0-7923-5627-6. (Cited on page 229.)

Rajeev Goré, Jimmy Thomson, and Florian Widmann. An experimental comparison of theorem provers for CTL. In Carlo Combi, Martin Leucker, and Frank Wolter, editors, *Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, Lübeck, Germany, September 12–14, 2011*, pages 49–56. IEEE, 2011. ISBN 978-1-4577-1242-5. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6063703`. (Cited on page 229.)

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. The Java Series. Addison-Wesley, Boston, Mass., May 2014. (Cited on pages 3, 50, 151, 152, 162, 167, 168, 169, 178, and 179.)

Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. *SIGPLAN Notices*, 44 (1):16–28, January 2009. ISSN 0362-1340. doi: 10.1145/1594834.1480886. URL `http://doi.acm.org/10.1145/1594834.1480886`. (Cited on pages 102, 227, 231, 239, and 257.)

Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for information flow control in Java programs – A practical guide. In Stefan Wagner and Horst Lichter, editors, *Software Engineering (Workshops)*, volume 215 of *Lecture Notes in Informatics*, pages 123–138. Gesellschaft für Informatik, 2013. ISBN 978-3-88579-609-1. (Cited on pages 28, 212, 222, and 244.)

Daniel Grahl and Christoph Scheben. *Functional and Information Flow Verification of an Electronic Voting System*, chapter 18. In Ahrendt et al. [2016], 2016. In preparation. (Cited on page 15.)

Daniel Grahl and Mattias Ulbrich. *From Specification to Proof Obligations*, chapter 8. In Ahrendt et al. [2016], 2016. In preparation. (Cited on pages iii, 15, 128, 174, 182, 184, and 187.)

Daniel Grahl, Richard Bubel, Wojciech Mostowski, Mattias Ulbrich, and Benjamin Weiß. *Modular Specification and Verification*, chapter 9. In Ahrendt et al. [2016], 2016. In preparation. (Cited on pages iii, 14, 31, 94, 182, 194, 196, and 199.)

Simon Greiner, Pascal Birnstill, and Erik Krempel. Privacy preserving surveillance and the tracking paradox. In *Proceedings, Future Security Conference 2013, 15–19 September 2013, Berlin*, 2013. URL `http://formal.iti.kit.edu/~greiner/pub/grein131.pdf`. (Cited on pages 242 and 243.)

David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*,

pages 366–373. Springer-Verlag, New York, NY, 1995. (Cited on pages 33 and 63.)

Christian Haack and Clément Hurlin. Separation logic contracts for a Java-like language with fork/join. In José Meseguer and Grigore Roşu, editors, *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings*, volume 5140 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2008. ISBN 978-3-540-79979-5. (Cited on page 238.)

Christian Haack, Erik Poll, and Aleksy Schubert. Explicit information flow properties in JML. In *3rd Benelux Workshop on Information and System Security (WISSec)*, November 2008. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.4289&rep=rep1&type=pdf`. (Cited on page 249.)

Christian Haack, Marieke Huisman, Clément Hurlin, and Afshin Amighi. Permission-based separation logic for multithreaded Java programs. *Logical Methods in Computer Science*, 11, 27 February 2015. doi: 10.2168/LMCS-11(1:2)2015. URL `http://www.lmcs-online.org/ojs/viewarticle.php?id=1250`. (Cited on page 238.)

Reiner Hähnle. Tableaux and related methods. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 4, pages 101–176. Elsevier, 2001. ISBN 0-444-508139. (Cited on page 66.)

Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IPGL*, 13(4):415–433, July 2005. (Cited on pages 33 and 63.)

Reiner Hähnle. The Abstract Behavioral Specification language: a tutorial introduction. In Elena Giachino, Reiner Hähnle, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7866 of *Lecture Notes in Computer Science*, pages 1–37. Springer, 2012. ISBN 978-3-642-40614-0. URL `http://dx.doi.org/10.1007/978-3-642-40615-7`. (Cited on page 231.)

Reiner Hähnle, Jing Pan, Philipp Rümmer, and Dennis Walter. Integration of a security type system into a program logic. In Ugo Montanari, Don Sanella, and R. Bruni, editors, *Proc. Trustworthy Global Computing, Lucca, Italy*, LNCS 4661. Springer, 2007. (Cited on page 243.)

Reiner Hähnle, Jing Pan, Philipp Rümmer, and Dennis Walter. Integration of a security type system into a program logic. *Theor. Comput. Sci*, 402(2-3): 172–189, 2008. URL `http://dx.doi.org/10.1016/j.tcs.2008.04.033`. (Cited on page 243.)

Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Y. H. Wong. HATS Abstract Behavioral Specification: The architectural view. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects 2011*, volume 7542 of *Lecture Notes in Computer Science*, pages 109–132. Springer, 2011. ISBN 978-3-642-35886-9; 978-3-642-35887-6. (Cited on pages 33 and 236.)

Christian Hammer. *Information Flow Control for Java – A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), July 2009. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000012049`. (Cited on pages 28, 212, 222, and 244.)

Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009. doi: 10.1007/s10207-009-0086-1. (Cited on pages 28 and 244.)

René Rydhof Hansen and Christian W. Probst. Non-interference and erasure policies for Java Card bytecode. In *6th International Workshop on Issues in the Theory of Security (WITS '06)*, 2006. (Cited on page 248.)

David Harel. *First-order dynamic logic*, volume 68 of *Lecture notes in computer science*. Springer-Verlag, New York, 1979. (Cited on pages 5, 38, 56, 77, 78, and 233.)

David Harel. Dynamic logic. In Dov Gabbay and Franz Guenther, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, volume 2, pages 497–604. D. Reidel Publishing Co., Dordrecht, 1984. (Cited on pages 56 and 242.)

David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. Software Eng*, 16(4):403–414, 1990. URL `http://doi.ieeecomputersociety.org/10.1109/32.54292`. (Cited on page 229.)

Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000. URL `http://dx.doi.org/10.1007/s100090050043`. (Cited on page 227.)

Daniel Hedin and David Sands. Timing aware information flow security for a JavaCard-like bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):163–182, 2005. (Cited on page 248.)

Daniel Hedin and David Sands. Noninterference in the presence of non-opaque pointers. In *19th IEEE Computer Security Foundations Workshop*, pages 217–229. IEEE Computer Society, 2006. ISBN 0-7695-2615-2. (Cited on page 151.)

Jesper G. Henriksen and P. S. Thiagarajan. Dynamic linear time temporal logic. *Ann. Pure Appl. Logic*, 96(1-3):187–207, 1999. URL `http://dx.doi.org/10.1016/S0168-0072(98)00039-6`. (Cited on page 228.)

Martin Hentschel, Stefan Käsdorf, Reiner Hähnle, and Richard Bubel. An interactive verification tool meets an IDE. In Gianluigi Zavattaro Elvira Albert, Emil Sekerinski, editor, *Proceedings of the 11th International Conference on Integrated Formal Methods*, LNCS, pages 55–70. Springer, September 2014. (Cited on page 163.)

Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. ISSN 0164-0925. doi: 10.1145/114005.102808. URL `http://doi.acm.org/10.1145/114005.102808`. (Cited on page 170.)

Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Abstract read permissions: Fractional permissions without the fractions. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 315–334. Springer, 2013. ISBN 978-3-642-35872-2; 978-3-642-35873-9. URL `http://dx.doi.org/10.1007/978-3-642-35873-9`. (Cited on page 237.)

David Hilbert and Wilhelm Ackermann. *Grundzüge der theoretischen Logik*. Springer-Verlag, Berlin, 3rd edition, 1949. (Cited on page 54.)

David Hilbert and Paul Bernays. *Die Grundlagen der Mathematik II*. Springer-Verlag, Berlin, 1939. (Cited on page 236.)

Mike Hinchey, Michael Jackson, Patrick Cousot, Byron Cook, Jonathan P. Bowen, and Tiziana Margaria. Software engineering and formal methods. *Commun. ACM*, 51(9):54–59, 2008. URL `http://doi.acm.org/10.1145/1378727.1378742`. (Cited on page 8.)

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969. (Cited on pages 4, 8, 29, 38, 51, 57, 88, 232, 234, and 241.)

C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972. (Cited on pages 57 and 118.)

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978. (Cited on page 88.)

C. A. R. Hoare and Jayadev Misra. Verified software: Theories, tools, experiments. Vision of a grand challenge project. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005. ISBN 978-3-540-69147-1. URL `http://dx.doi.org/10.1007/978-3-540-69149-5`. (Cited on pages 181 and 254.)

Charles Anthony Richard Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2), February 1981. (Cited on pages 32 and 188.)

Ian Hodkinson and Mark Reynolds. Temporal logic. In Patrick Blackburn, Johan van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*, number 3 in Studies in Logic and Practical Reasoning, chapter 11, pages 655–720. Elsevier, 2007. (Cited on page 56.)

Marieke Huisman and Wojciech Mostowski. A symbolic approach to permission accounting for concurrent reasoning. In Daniel Grosu, Hai Jin, and George Papadopoulos, editors, *14th International Symposium on Parallel and Distributed Computing, ISPDC 2015*, pages 165–174. IEEE, 2015. ISBN 978-1-4673-7147-6; 978-1-4673-7148-3. (Cited on page 237.)

Marieke Huisman and Trí Minh Ngô. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *FoVeOOS*, volume 7421 of *Lecture Notes in Computer Science*, pages 178–195. Springer, 2012. ISBN 978-3-642-31761-3. (Cited on pages 25 and 247.)

Marieke Huisman and Gustavo Petri. The Java Memory Model: a formal explanation. In Christian Haack, Marieke Huisman, Joe Kiniry, and Erik Poll, editors, *Verification and Analysis of Multi-threaded Java-like Programs*, pages 81–96, 2007. URL `http://www-sop.inria.fr/everest/personnel/Gustavo.Petri/publis/jmm-vamp07.pdf`. (Cited on page 167.)

Marieke Huisman, Pratik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *19th IEEE Computer Security Foundations Workshop (CSFW 2006)*, page 3. IEEE Computer Society, 2006. ISBN 0-7695-2615-2. URL `http://doi.ieeecomputersociety.org/10.1109/CSFW.2006.6`. (Cited on pages 25, 136, 144, 145, 147, and 245.)

Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. VerifyThis verification competition 2012 – organizer's report. Technical Report 2013-1, Department of Informatics, Karlsruhe Institute of Technology, 2013. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000034373. (Cited on page 181.)

Marieke Huisman, Wolfgang Ahrendt, Daniel Bruns, and Martin Hentschel. Formal specification with JML. Technical Report 2014-10, Department of Informatics, Karlsruhe Institute of Technology, 2014. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000041881. (Cited on pages iii, 14, 182, and 184.)

Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. VerifyThis 2012. *International Journal on Software Tools for Technology Transfer*, 17(6):647–657, 2015. ISSN 1433-2779. doi: 10.1007/s10009-015-0396-8. URL http://dx.doi.org/10.1007/s10009-015-0396-8. (Cited on page 181.)

Sebastian Hunt and David Sands. On flow-sensitive security types. In J. Gregory Morrisett and Simon Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 79–90. ACM, 2006. ISBN 1-59593-027-2. URL http://doi.acm.org/10.1145/1111037.1111045. (Cited on page 28.)

Clément Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. Theses, Université Nice Sophia Antipolis, September 2009. URL https://tel.archives-ouvertes.fr/tel-00424979/file/these-clement-hurlin.pdf. (Cited on page 238.)

Faraz Hussain and Gary T. Leavens. temporaljmlc: A JML runtime assertion checker extension for specification and checking of temporal properties. Technical Report CS-TR-10-08, UCF, Dept. of EECS, Orlando, Florida, July 2010. (Cited on pages 230 and 231.)

Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 271–282. ACM, 2011. ISBN 978-1-4503-0490-0. URL http://dl.acm.org/citation.cfm?id=1926385. (Cited on page 237.)

Bart Jacobs and Wolter Pieters. Electronic voting in the Netherlands: From early adoption to early abolishment. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 121–144, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-03828-0. doi:

10.1007/978-3-642-03829-7_4. URL http://dx.doi.org/10.1007/978-3-642-03829-7_4. (Cited on page 210.)

Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997. (Cited on page 198.)

Bart Jacobs, Wolter Pieters, and Martijn Warnier. Statically checking confidentiality via dynamic labels. In Catherine Meadows, editor, *Proceedings of the POPL 2005 Workshop on Issues in the Theory of Security, WITS 2005, Long Beach, California, USA, January 10-11, 2005*, pages 50–56. ACM, 2005. ISBN 1-58113-980-2. URL http://doi.acm.org/10.1145/1045405.1045411. (Cited on page 243.)

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011. ISBN 978-3-642-20397-8. URL http://dx.doi.org/10.1007/978-3-642-20398-5. (Cited on pages 9, 232, 238, and 240.)

Jean-Baptiste Jeannin and André Platzer. dTL^2: Differential temporal dynamic logic with nested modalities for hybrid systems. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Int. Joint Conference on Automated Reasoning 2014*, volume tba of *LNCS*. Springer, 2014. (Cited on pages 66, 83, and 228.)

Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1-2):23–66, 2006. (Cited on page 236.)

Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010. ISBN 978-3-642-25270-9. (Cited on pages 33, 49, 231, and 236.)

Cliff B. Jones. *Development Methods for Computer Programs Including a Notion of Interference*. PhD thesis, Oxford University, June 1981. (Cited on page 234.)

Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. URL http://doi.acm.org/10.1145/69575.69577. (Cited on pages 4, 13, 88, 89, 90, 91, 92, 97, 99, 107, 202, and 234.)

Cliff B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, 1996. URL `http://dx.doi.org/10.1007/BF00122417`. (Cited on pages 32 and 182.)

Cliff B. Jones. Wanted: a compositional approach to concurrency. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, Monographs in Computer Science, chapter 1, pages 5–15. Springer, New York, 2003. ISBN 978-1-4419-2964-8. doi: 10.1007/978-0-387-21798-7_20. (Cited on pages 113 and 254.)

Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, 2000. (Cited on pages 29, 118, 123, and 241.)

Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005. ISBN 3-540-00701-6. (Cited on page 249.)

Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In T. Nipkow E. Sekerinski, J. Misra, editor, *Formal Methods (FM)*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283, Berlin, 2006. Springer-Verlag. (Cited on pages 200 and 201.)

Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects Computing*, 23(3):267–288, 2011. URL `http://dx.doi.org/10.1007/s00165-010-0152-5`. (Cited on pages 91, 200, and 201.)

Matt Kaufmann and J Strother Moore. Structured theory development for a mechanized logic. *Journal of Auntomated Reasoning*, 26(2):161–203, March 1999. (Cited on page 42.)

Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976. (Cited on page 88.)

James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976. (Cited on page 57.)

Joseph R. Kiniry, Alan E. Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The KOA remote voting system: A summary of work to date. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *Proceedings of Trustworthy Global Computing (TGC)*, volume 4661 of *Lecture Notes in Computer Science*, pages 244–262. Springer, 2006. ISBN 978-3-540-75333-9. (Cited on page 250.)

Michael Kirsten. Proving well-definedness of JML specifications with KeY. Studienarbeit, Karlsruhe Institute of Technology, 2013. (Cited on page 185.)

Vladimir Klebanov. *Extending the Reach and Power of Deductive Program Verification*. PhD thesis, Universität Koblenz, 2009. (Cited on pages 167 and 234.)

Vladimir Klebanov. Precise quantitative information flow analysis – a symbolic approach. *Theoretical Computer Science*, 538(0):124–139, 2014. doi: 10.1016/j.tcs.2014.04.022. (Cited on pages 242 and 256.)

Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st Verified Software Competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM)*, volume 6664 of *LNCS*. Springer, 2011. Materials available at `www.vscomp.org`. (Cited on pages 166 and 181.)

Donald E. Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion. Letter to Peter van Emde Boas, March 29 1977. (Cited on page 256.)

Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology—CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 18–22 August 1996. (Cited on page 22.)

Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983. (Cited on page 56.)

Max E. Kramer, Anton Hergenröder, Martin Hecker, Simon Greiner, and Kaibin Bao. Specification and verification of confidentiality in component-based systems. Poster at the 35th IEEE Symposium on Security and Privacy, 2014. URL `http://www.ieee-security.org/TC/SP2014/posters/KRAME.pdf`. (Cited on page 249.)

Saul Kripke. Semantical considerations on modal logic. *Acta philosphica fennica*, 16:83–94, 1963. (Cited on pages 55, 64, and 229.)

Viktor Kuncak. Developing verified software using leon (invited contribution). In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods (NFM 2015)*, volume 9058 of *Lecture Notes in Computer*

Science. Springer-Verlag, 2015. ISBN 978-3-319-17523-2 (Print) 978-3-319-17524-9 (Online). URL http://lara.epfl.ch/~kuncak/papers/Kuncak15DevelopingVerifiedSoftwareUsingLeonNFM.pdf. (Cited on page 9.)

Ralf Küsters and Tomasz Truderung. Security in e-voting. *it-Information Technology*, 56(6):300–306, 2014. (Cited on page 210.)

Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Verifiability, privacy, and coercion-resistance: New insights from a case study. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P)*, pages 538–553, Oakland, California, USA, 2011. IEEE Computer Society. (Cited on pages 15, 210, 212, 215, 216, and 224.)

Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A framework for the cryptographic verification of Java-like programs. Technical report, Karlsruhe Institute of Technology, 2012. (Cited on pages 18 and 212.)

Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Jürgen Graf, and Christoph Scheben. A hybrid approach for proving noninterference and applications to the cryptographic verification of Java programs. In Christian Hammer and Sjouke Mauw, editors, *Grande Region Security and Reliability Day 2013*, Luxembourg, 2013. URL http://grsrd.uni.lu/papers/grsrd2013_submission_2.pdf. Extended Abstract. (Cited on pages 15, 30, 211, 222, and 224.)

Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. A hybrid approach for proving noninterference of Java programs. In Cédric Fournet and Michael Hicks, editors, *28th IEEE Computer Security Foundations Symposium*, 2015. (Cited on pages 15, 30, 222, 224, 254, and 258.)

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558–565, 1978. (Cited on page 167.)

Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. URL http://doi.ieeecomputersociety.org/10.1109/TC.1979.1675439. (Cited on page 167.)

Leslie Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14:21–37, 1980. (Cited on page 88.)

Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Proceedings of the IFIP Congress on Information Processing*, pages 657–667, Amsterdam, 1983. North-Holland. (Cited on page 189.)

Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10), October 1973. (Cited on page 22.)

Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 113–135. Cambridge University Press, 2000. URL `citeseer.ist.psu.edu/leavens00concepts.html`. (Cited on page 190.)

Gary T. Leavens and David A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 2006. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR06-36/TR.pdf`. (Cited on page 190.)

Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, 1995. doi: 10.1007/BF01178658. (Cited on page 190.)

Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 221–236, New York, NY, USA, 2006a. ACM. ISBN 1-59593-237-2. (Cited on page 181.)

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006b. URL `http://doi.acm.org/10.1145/1127878.1127884`. (Cited on pages 10, 183, and 199.)

Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007. ISSN 1433-299X. URL `http://dx.doi.org/10.1007/s00165-007-0026-7`. (Cited on pages 181 and 254.)

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*, May 31 2013. URL `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`. Draft revision 2344. (Cited on page 183.)

Gary Todd Leavens. *Verifying Object-Oriented Programs that use Subtypes*. PhD thesis, Massachusetts Institute of Technology, December 1988. (Cited on page 190.)

K. Rustan M. Leino. *Towards Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. URL `http://caltechcstr.library.caltech.edu/234/00/95-03.ps`. Available as Technical Report Caltech-CS-TR-95-03. (Cited on page 181.)

K. Rustan M. Leino. Specification and verification of object-oriented software. In *Engineering Methods and Tools for Software Safety and Security*, volume 22 of *NATO Science for Peace and Security, Series D: Information and Communication Security*, pages 231–266. IOS Press, 2009. (Cited on pages 9 and 231.)

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning, 16th International Conference, LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer-Verlag, 2010. (Cited on pages 9, 12, 38, and 231.)

K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, Edition 0. In *Tools and Experiments workshop at VSTTE 2010*, Edinburgh, UK, 2010. (Cited on page 232.)

K. Rustan M. Leino and Michał Moskal. Co-induction simply. Technical Report MSR-TR-2013-49, Microsoft Research, July 2013. (Cited on page 197.)

K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393, Berlin, March 2009. Springer-Verlag. (Cited on page 239.)

K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24 (5):491–553, September 2002. (Cited on pages 194 and 201.)

K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010. ISBN 978-3-642-12001-5. (Cited on page 38.)

K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03828-0. doi: 10.1007/978-3-642-03829-7_7. URL `http://dx.doi.org/10.1007/978-3-642-03829-7_7`. (Cited on pages 12 and 238.)

Leonid Levin. Universal'nye perebornye zadachi. *Problemy Peredachi Informatsii*, 3(9):115–116, 1973. English translation: "Universal search problems". (Cited on page 18.)

Azriel Levy. *Basic Set Theory*. Springer, Berlin, 1979. ISBN 3-540-08417-7. (Cited on page 196.)

Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. *Principles of Programming Languages 2005 (32nd POPL'2005), ACM SIGPLAN Notices*, 40(1), January 2005. (Cited on page 242.)

Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985. ISBN 3-540-15648-8. URL `http://dx.doi.org/10.1007/3-540-15648-8_16`. (Cited on pages 56 and 58.)

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. The Java Series. Addison-Wesley, Boston, Mass., May 2014. (Cited on pages 54, 149, and 167.)

Barbara Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, pages 17–34, May 1988. (Cited on pages 190 and 202.)

Barbara Liskov and Jeanette M. Wing. Specifications and their use in defining subtypes. In Andreas Paepcke, editor, *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 16–28, Washington DC, USA, 1993. ACM Press. ISBN 0-201-58895-1. (Cited on pages 188 and 190.)

Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994. (Cited on page 190.)

Tianhai Liu, Mihai Herda, Shmuel Tyszberowicz, Daniel Grahl, Mana Taghdiri, and Bernhard Beckert. Computing specification-sensitive abstractions for program verification. Submitted, 2015. (Cited on page 259.)

Andreas Lochbihler. Making the Java memory model safe. *ACM Transactions on Programming Languages and Systems*, 35(4):12:1–12:65, 2014. doi: 10.1145/2518191. (Cited on page 167.)

Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995. ISBN 0-387-94459-1. (Cited on pages 36 and 55.)

William Mansky and Elsa L. Gunter. Using locales to define a rely-guarantee temporal logic. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2012. ISBN 978-3-642-32346-1. URL `http://dx.doi.org/10.1007/978-3-642-32347-8`. (Cited on page 237.)

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 378–391. ACM, 2005. ISBN 1-58113-830-X. URL `http://dl.acm.org/citation.cfm?id=1040305`. (Cited on page 167.)

Heiko Mantel. Information flow control and applications – bridging a gap. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 153–172. Springer, 2001. ISBN 3-540-41791-5. (Cited on page 242.)

Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, volume 6345 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2010. ISBN 978-3-642-15496-6. URL `http://dx.doi.org/10.1007/978-3-642-15497-3`. (Cited on page 245.)

Heiko Mantel and Henning Sudbrock. Types vs. PDGs in information flow analysis. In Elvira Albert, editor, *LOPSTR*, volume 7844 of *Lecture Notes in Computer Science*, pages 106–121. Springer, 2012. ISBN 978-3-642-38196-6; 978-3-642-38197-3. (Cited on page 28.)

Heiko Mantel, Henning Sudbrock, and Tina Kraußer. Combining different proof techniques for verifying information flow security. In Germán Puebla, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR)*,

volume 4407 of *Lecture Notes in Computer Science*, pages 94–110. Springer, 2006. ISBN 978-3-540-71409-5. URL http://dx.doi.org/10.1007/978-3-540-71410-1_8. (Cited on page 245.)

Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 218–232. IEEE Computer Society, 2011. ISBN 978-1-61284-644-6. URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5991608. (Cited on pages 246 and 259.)

Heiko Mantel, Matthias Perner, and Jens Sauer. Noninterference under weak memory models. In Anupam Datta, Cédric Fournet, Deepak Garg, and Laura Kovacs, editors, *Computer Security Foundations Symposium 2014*. IEEE, 2014. (Cited on page 167.)

Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical report, Columbia University, 4 May 1991. (Cited on page 170.)

John McCarthy. Towards a mathematical science of computation. In *Information Processing '62*, pages 21–28, Amsterdam, 1962. North-Holland. (Cited on pages 38, 39, and 64.)

John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969. (Cited on page 89.)

Daryl McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society, 1987. ISBN 0-8186-0771-8. (Cited on pages 28 and 137.)

Daryl McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy*, pages 177–186. IEEE Computer Society, 1988. ISBN 0-8186-0850-1. (Cited on pages 24 and 135.)

Annabelle McIver and Carroll Morgan. A probabilistic approach to information hiding. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, Monographs in Computer Science, pages 441–460. Springer, New York, 2003. ISBN 978-1-4419-2964-8. doi: 10.1007/978-0-387-21798-7_20. URL http://dx.doi.org/10.1007/978-0-387-21798-7_20. (Cited on page 23.)

John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992. (Cited on pages 6, 20, 25, 118, 135, 138, and 144.)

John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994. (Cited on page 26.)

John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, January 1996. (Cited on pages 26 and 246.)

Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992. (Cited on pages 181, 189, 192, and 213.)

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997. ISBN 0-13-629155-4. (Cited on page 188.)

Bertrand Meyer. The grand challenge of trusted components. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 660–667, Piscataway, NJ, 3–10 May 2003. IEEE Computer Society. URL `http://www.inf.ethz.ch/~meyer/publications/ieee/trusted-icse.pdf`. (Cited on page 256.)

Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation – 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2014. ISBN 978-3-642-54012-7. URL `http://dx.doi.org/10.1007/978-3-642-54013-4`. (Cited on page 235.)

Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981. (Cited on page 89.)

John C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 365–458. Elsevier, 1990. (Cited on page 153.)

Martin Mohr, Jürgen Graf, and Martin Hecker. Jodroid: Adding Android support to a static information flow control tool. In *Proceedings of the 8th Working Conference on Programming Languages (ATPS'15)*, Lecture Notes in Informatics (LNI) 215. Springer Berlin / Heidelberg, February 2015. to appear. (Cited on page 248.)

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965. (Cited on page 2.)

Scott Moore, Aslan Askarov, and Stephen Chong. Knowledge and effect: A logic for reasoning about confidentiality and integrity guarantees (extended abstract). In Deepak Garg and Boris Köpf, editors, *Workshop on Foundations of Computer Security (FCS 2015)*, July 2015. URL `http://software.imdea.org/~bkoepf/FCS15/paper10.pdf`. (Cited on page 242.)

Wojciech Mostowski. Formal reasoning about non-atomic Java Card methods in Dynamic Logic. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings, Formal Methods (FM) 2006*, volume 4085 of *LNCS*, pages 444–459. Springer, August 2006. (Cited on pages 57 and 162.)

Wojciech Mostowski. Fully verified Java Card API reference implementation. In Bernhard Beckert, editor, *Proceedings of 4th International Verification Workshop in connection with CADE-21, Bremen, Germany, July 15-16, 2007*, volume 259 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007a. URL `http://ceur-ws.org/Vol-259/paper12.pdf`. (Cited on page 166.)

Wojciech Mostowski. From sequential JAVA to JAVA CARD. In Beckert et al. [2007a], chapter 9, pages 375–405. (Cited on pages 57 and 162.)

Wojciech Mostowski. The Demoney case study. In Beckert et al. [2007a], chapter 14, pages 533–568. (Cited on page 166.)

Wojciech Mostowski. A case study in formal verification using multiple explicit heaps. In Dirk Beyer and Michele Boreale, editors, *Formal Techniques for Distributed Systems – Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3–5, 2013. Proceedings*, volume 7892 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2013. ISBN 978-3-642-38591-9. URL `http://dx.doi.org/10.1007/978-3-642-38592-6`. (Cited on page 238.)

Wojciech Mostowski. Dynamic frames based verification method for concurrent Java programs. In Arie Gurfinkel and Sanjit A. Seshia, editors, *Verified Software: Theories, Tools, Experiments (VSTTE 2015)*, Lecture Notes in Computer Science. Springer, 2015. (Cited on pages 238, 254, 255, and 258.)

Ben Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2), February 1985. (Cited on page 56.)

Chunyan Mu. Quantitative information flow for security: a survey. Technical Report 08-06, Department of Computer Science, King's College London, 2008. URL `http://www.dcs.kcl.ac.uk/technical-reports/papers/TR-08-06.pdf`. (Cited on page 23.)

Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*, pages 228–241, New York, January 1999. Association for Computing Machinery. ISBN 1-58113-095-3. (Cited on pages 28, 154, 248, and 250.)

Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Security and Privacy – 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*, pages 186–197, 1998. (Cited on page 28.)

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent type theory for verification of information flow and access control policies. *ACM Trans. Program. Lang. Syst*, 35(2):6, 2013. (Cited on page 241.)

David A. Naumann. From coupling relations to mated invariants for secure information flow. In *European Symposium on Research in Computer Security (ESORICS)*, number 4189 in Lecture Notes in Computer Science, pages 279–296, 2006. (Cited on pages 148, 154, and 249.)

Trí Minh Ngô. *Qualitative and quantitative information flow analysis for multi-thread programs*. PhD thesis, University of Twente, 2014. (Cited on page 247.)

Trí Minh Ngô, Mariëlle Stoelinga, and Marieke Huisman. Effective verification of confidentiality for multi-threaded programs. *Journal of Computer Security*, 22(2):269–300, 2014. doi: 10.3233/JCS-130492. (Cited on page 247.)

Pavel Nikolov. Combining theorem proving and type systems for precise and efficient information flow verification. Studienarbeit, Karlsruhe Institute of Technology, 2014. (Cited on page 248.)

Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 161–170. ACM, 1998. ISBN 0-89791-979-3. URL `http://dl.acm.org/citation.cfm?id=268946`. (Cited on pages 167 and 179.)

Bashar Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, May/June 1997. ISSN 0740-7459. (Cited on page 7.)

Kirsten Nygaard and Ole-Johan Dahl. Simula 67. In R. W. Wexelblat, editor, *History of Programming Languages*. ACM press, 1981. (Cited on page 188.)

Colin O'Halloran. A calculus of information flow. In *ESORICS*, pages 147–159. AFCET, 1990. (Cited on page 26.)

Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, May 2007. ISSN 0304-3975 (print), 1879-2294 (electronic). (Cited on page 237.)

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. ISBN 3-540-42554-3. URL `http://dx.doi.org/10.1007/3-540-44802-0_1`. (Cited on page 232.)

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *ACM Transactions on Programming Languages and Systems*, 31(3):11:1–11:50, April 2009. ISSN 0164-0925 (print), 1558-4593 (electronic). doi: 10.1145/1498926.1498929. URL `http://portal.acm.org/browse_dl.cfm?idx=J783`. (Cited on page 232.)

Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976. (Cited on pages 4, 88, 107, 234, 235, 237, 241, and 255.)

Jing Pan. A theorem proving approach to analysis of secure information flow using data abstraction. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2005. (Cited on page 28.)

Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12–14, 2005*, pages 247–258. ACM, 2005. ISBN 1-58113-830-X. URL `http://dl.acm.org/citation.cfm?id=1040305`. (Cited on page 238.)

Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. (Cited on page 235.)

David Peleg. Concurrent dynamic logic. *Journal of the ACM*, 34(2):450–479, April 1987. (Cited on page 233.)

Michael Philippsen. A survey of concurrent object-oriented languages. *Concurrency – Practice and Experience*, 12(10):917–980, 2000. (Cited on page 3.)

André Platzer. An object-oriented dynamic logic with updates. Master's thesis, Universität Karlsruhe, 2004. (Cited on pages 78 and 179.)

André Platzer. A temporal dynamic logic for verifying hybrid system invariants. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2007, New York, NY, USA, June 4-7, 2007, Proceedings*, volume 4514 of *Lecture Notes in Computer Science*, pages 457–471. Springer, 2007. ISBN 978-3-540-72732-3. URL `http://dx.doi.org/10.1007/978-3-540-72734-7_32`. (Cited on pages 78 and 228.)

André Platzer and Jan-David Quesel. KeYmaera: A hybrid theorem prover for hybrid systems (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 2008. ISBN 978-3-540-71069-1. URL `http://dx.doi.org/10.1007/978-3-540-71070-7_15`. (Cited on page 163.)

Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 18th Annual Symposium*, pages 46–57. IEEE, October 1977. doi: 10.1109/SFCS.1977.32. (Cited on page 9.)

Arnd Poetzsch-Heffter, Ilham W. Kurnia, and Christoph Feller. Verification of actor systems needs specification techniques for strong causality and hierarchical reasoning. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software (FoVeOOS 2011), Papers Presented at the 2nd International Conference, October 5–7, 2011, Turin, Italy*, pages 289–305, 2011. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000024780`. (Cited on page 205.)

Arndt Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999. (Cited on page 38.)

Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Proving concurrent noninterference. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs – Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2012. ISBN 978-3-642-35307-9. (Cited on pages 26 and 243.)

Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Formal verification of language-based concurrent noninterference. *Journal of Formal Reasoning*, 6(1):1–30, 2013a. (Cited on page 243.)

Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Noninterfering schedulers—when possibilistic noninterference implies probabilistic noninterference. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science – 5th International Conference, CALCO 2013, Warsaw, Poland, September 3–6, 2013. Proceedings*, volume 8089 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2013b. ISBN 978-3-642-40205-0. URL `http://dx.doi.org/10.1007/978-3-642-40206-7`. (Cited on page 245.)

Leonor Prensa Nieto. Completeness of the Owicki-Gries system for parameterized parallel programs. In Michel Charpentier and Beverly Sanders, editors, *6th International Workshop on Formal Methods for Parallel Programming: Theory and Applications. Held in conjunction with the 15th International Parallel and Distributed Processing Symposium*. IEEE CS Press, 2001. URL `http://computer.org/cspress/CATALOG/pr00990.htm`. (Cited on page 88.)

Leonor Prensa Nieto. *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002. URL `http://mediatum.ub.tum.de/doc/601717/601717.pdf`. (Cited on pages 89, 91, 92, 107, 234, and 235.)

Leonor Prensa Nieto. The Rely-Guarantee method in Isabelle/HOL. In P. Degano, editor, *European Symposium on Programming (ESOP'03)*, volume 2618 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2003. (Cited on page 91.)

Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report 00-03e, Iowa State University, Department of Computer Science, May 2005. (Cited on page 187.)

Wolfgang Reif. The KIV-approach to software verification. In Manfred Broy and Stefan Jähnichen, editors, *KORSO – Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, pages 339–370. Springer, 1995. ISBN 3-540-60589-4. URL `http://dx.doi.org/10.1007/BFb0015471`. (Cited on page 9.)

John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages 1975*, pages 157–168, Rocquencourt, France, 1975. IFIP Working Group 2.1 on Algol, INRIA. (Cited on page 196.)

John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, UK, 1998. (Cited on page 41.)

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. (Cited on page 232.)

Mark Reynolds and Clare Dixon. Theorem-proving for discrete temporal logic. In Michael Fisher, Dov Gabbay, and Lluis Vila, editors, *Handbook of temporal reasoning in artificial intelligence*. Elsevier Science, 2005. (Cited on page 229.)

Martin C. Rinard. Analysis of multithreaded programs. In Patrick Cousot, editor, *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, volume 2126 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. ISBN 3-540-42314-1. URL `http://dx.doi.org/10.1007/3-540-47764-0_1`. (Cited on page 227.)

Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: A flexible framework for creating software model checkers. In Phil McMinn, editor, *Testing: Academia and Industry Conference; Practice And Research Techniques (TAIC PART), Windsor, United Kingdom*, pages 3–22. IEEE Computer Society, 2006. (Cited on page 227.)

A. W. Roscoe. CSP and determinism in security modelling. In *IEEE Symposium on Security and Privacy*, pages 114–127. IEEE Computer Society, 1995. ISBN 0-8186-7015-0. URL `http://www.computer.org/csdl/proceedings/sp/1995/7015/00/index.html`. (Cited on pages 25 and 144.)

A. W. Roscoe, Jim Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–54, 1996. URL `http://dx.doi.org/10.3233/JCS-1996-4103`. (Cited on pages 25 and 144.)

Stan Rosenberg, Anindya Banerjee, and David A. Naumann. Decision procedures for region logic. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 379–395, Berlin Heidelberg, 2012. Springer. ISBN 978-3-642-27939-3. doi: 10.1007/978-3-642-27940-9_25. URL `http://dx.doi.org/10.1007/978-3-642-27940-9_25`. (Cited on page 233.)

Andreas Roth. *Specification and Verification of Object-oriented Software Components*. PhD thesis, Universität Karlsruhe, 2006. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000004542`. (Cited on page 175.)

Fabian Ruch. Efficient logic-based information flow analysis of object-oriented programs. Bachelor thesis, Karlsruhe Institute of Technol-

ogy, 2013. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000036850. (Cited on pages 243 and 248.)

Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In Miki Hermann and Andrei Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 422–436. Springer-Verlag, 2006. (Cited on pages 57, 58, 61, 69, and 108.)

Philipp Rümmer. Construction of proofs. In Beckert et al. [2007a], chapter 4, pages 179–242. (Cited on page 164.)

John M. Rushby. Automated formal methods enter the mainstream. *J.UCS: Journal of Universal Computer Science*, 13(5):650–660, 2007. URL http://www.jucs.org/jucs_13_5/automated_formal_methods_enter/jucs_13_5_0650_0660_rushby.pdf. (Cited on page 8.)

Alejandro Russo and Andrei Sabelfeld. Securing interaction between threads and the scheduler. In *Computer Security Foundations Workshop 2006*, pages 177–189. IEEE Computer Society, 2006. ISBN 0-7695-2615-2. URL http://doi.ieeecomputersociety.org/10.1109/CSFW.2006.29. (Cited on pages 245, 246, and 259.)

Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Computer Security Foundations*, pages 186–199. IEEE Computer Society, 2010. ISBN 978-0-7695-4082-5. (Cited on pages 29 and 222.)

Alejandro Russo, John Hughes, David A. Naumann, and Andrei Sabelfeld. Closing internal timing channels by transformation. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science – ASIAN 2006. Secure Software and Related Issues*, volume 4435 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2006. ISBN 978-3-540-77504-1. URL http://dx.doi.org/10.1007/978-3-540-77505-8_10. (Cited on page 245.)

Andrei Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, July 2-6, 2001, Revised Papers*, volume 2244 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2001. ISBN 3-540-43075-X. URL http://dx.doi.org/10.1007/3-540-45575-2_22. (Cited on page 245.)

Andrei Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers*, volume 2890 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2003. ISBN 3-540-20813-5. URL `http://dx.doi.org/10.1007/978-3-540-39866-0_27`. (Cited on page 245.)

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003a. (Cited on pages 5, 21, 22, 24, 123, 125, 126, 131, 132, 144, 147, 242, and 247.)

Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *International Symposium on Software Security – Theories and Systems (ISSS)*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2003b. ISBN 3-540-23635-X. (Cited on page 27.)

Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 352–365. Springer, 2009. ISBN 978-3-642-11485-4. (Cited on page 27.)

Andrei Sabelfeld and David Sands. Probabilistic noninterference for multithreaded programs. In *13th IEEE Computer Security Foundations Workshop*, Cambridge, UK, July 2000. IEEE Computer Society Press. URL `http://www.cs.chalmers.se/~andrei/probnon.ps`. (Cited on pages 26, 245, and 246.)

Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1): 59–91, 2001. (Cited on page 242.)

Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009. URL `http://dx.doi.org/10.3233/JCS-2009-0352`. (Cited on pages 21, 127, and 242.)

Aneesa Saeed and S.H.A. Hamid. Theorem prover based static analyzer: Comparison analysis between ESC/Java2 and KeY. In Hamzah Asyrani Sulaiman, Mohd Azlishah Othman, Mohd Fairuz Iskandar Othman, Yahaya Abd Rahim, and Naim Che Pee, editors, *Advanced Computer and Communication Engineering Technology*, volume 315 of *Lecture Notes*

in Electrical Engineering, pages 727–737. Springer International Publishing, 2015. ISBN 978-3-319-07673-7. doi: 10.1007/978-3-319-07674-4_68. URL http://dx.doi.org/10.1007/978-3-319-07674-4_68. (Cited on page 166.)

Christoph Scheben. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014. submitted. (Cited on pages 14, 15, 20, 118, 121, 126, 128, 155, 157, 159, 163, 182, 204, 205, 206, 207, 211, 218, 224, 241, 243, 249, 250, and 258.)

Christoph Scheben and Peter H. Schmitt. Verification of information flow properties of JAVA programs without approximations. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software International Conference, FoVeOOS 2011, Revised Selected Papers*, volume 7421 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 2012a. (Cited on pages 21, 22, 29, 38, 57, 118, 121, 123, 127, 128, 147, 154, 157, 158, 160, 162, 179, 204, 205, 206, 211, 241, 242, 243, and 249.)

Christoph Scheben and Peter H. Schmitt. Dynamic logic for product programs. Unpublished, February 2012b. (Cited on pages 138 and 242.)

Christoph Scheben and Peter H. Schmitt. Efficient self-composition for weakest precondition calculi. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: 19th International Symposium on Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 579–594, Singapore, 2014. Springer-Verlag. ISBN 978-3-319-06409-3. doi: 10.1007/978-3-319-06410-9_39. URL http://link.springer.com/chapter/10.1007/978-3-319-06410-9_39. (Cited on page 121.)

Gerhard Schellhorn, Bogdan Tofan, Gidon Ernst, and Wolfgang Reif. Interleaved programs and rely-guarantee reasoning with ITL. In Carlo Combi, Martin Leucker, and Frank Wolter, editors, *Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, Lübeck, Germany, September 12-14, 2011*, pages 99–106. IEEE, 2011. ISBN 978-1-4577-1242-5. URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6063703. (Cited on page 235.)

Gerhard Schellhorn, Bogdan Tofan, Gidon Ernst, Jörg Pfähler, and Wolfgang Reif. RGITL: A temporal logic framework for compositional reasoning about interleaved programs. *Annals of Mathematics and Artificial Intelligence*, 71(1-3):131–174, 2014. (Cited on page 235.)

Peter H. Schmitt. A computer-assisted proof of the Bellman-Ford lemma. Technical Report 15, Karlsruhe Institute of Technology, Fakultät für Informatik, 2011. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022513. (Cited on page 63.)

Peter H. Schmitt and Isabel Tonin. Verifying the Mondex case study. In *Software Engineering and Formal Methods 2007*, pages 47–58. IEEE Computer Society, 2007. ISBN 978-0-7695-2884-7. (Cited on page 166.)

Peter H. Schmitt and Mattias Ulbrich. Typed first-order logic. In *VERIFY 2014*, 2014. (Cited on pages 55 and 76.)

Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in Java dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Revised Selected Papers, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011. (Cited on pages 38, 158, 200, and 201.)

Wolfram Schulte, Xia Songtao, Jan Smans, and Frank Piessens. A glimpse of a verifying C compiler, April 03 2008. (Cited on page 231.)

Stephan Schwendimann. A new one-pass tableau calculus for PLTL. In Harrie de Swart, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-98)*, volume 1397 of *LNAI*, pages 277–291, Berlin, May 5–8 1998. Springer. ISBN 3-540-64406-7. (Cited on page 229.)

Dana S. Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Proceedings Symposium on Computers and Automata*, volume 21 of *Microwave Institute Symposia Series*, pages 19–46, New York, NY, 1971. Polytechnic Institute of Brooklyn. (Cited on page 41.)

Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Massachusetts, 1967. ISBN 0201070286. (Cited on pages 42, 66, and 105.)

Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275, Berlin, April 2008. Springer-Verlag. doi: 10.1007/978-3-540-78743-3_19. URL `https://lirias.kuleuven.be/handle/123456789/178243`. (Cited on page 38.)

Jan Smans, Dries Vanoverberghe, Dominique Devriese, Bart Jacobs, and Frank Piessens. Shared boxes: Rely-guarantee reasoning in VeriFast. Technical Report CW 662, KU Leuven, Department of Computer Science, May 2014. URL `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW662.pdf`. (Cited on pages 232, 233, and 240.)

Geoffrey Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Computer Security Foundations Workshop*, pages 3–13.

IEEE Computer Society, 2003. ISBN 0-7695-1927-X. URL http://doi.ieeecomputersociety.org/10.1109/CSFW.2003.1212701. (Cited on pages 26 and 245.)

Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 19–21, 1998. (Cited on pages 28, 244, and 245.)

Gregor Snelting. Understanding probabilistic software leaks. *Science of Computer Programming*, 97(1):122–126, January 2015. doi: 10.1016/j.scico.2013.11.008. URL http://www.sciencedirect.com/science/article/pii/S016764231300292X. Special Issue on New Ideas and Emerging Results in Understanding Software. (Cited on pages 26, 136, 256, and 259.)

Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using JOANA. *it – Information Technology*, 56:280–287, November 2014. doi: 10.1515/itit-2014-1051. (Cited on page 28.)

Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: definition, verification, validation.* Springer-Verlag, 2001. ISBN 3-540-42088-6. (Cited on pages 32 and 179.)

Kurt Stenzel. *Verification of Java Card Programs.* PhD thesis, Fakultät für angewandte Informatik, University of Augsburg, 2005. URL http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/dissertations/2005_stenzel_diss/diss-stenzel.pdf. (Cited on page 38.)

Jacques Stern. Why provable security matters? In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, volume 2656 of *Lecture Notes in Computer Science*, pages 449–461. Springer, 2003. ISBN 3-540-14039-5. (Cited on page 250.)

Colin Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58(1–3):347–359, July 1988. (Cited on pages 88, 99, and 234.)

Colin Stirling. Modal and temporal logics. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science. Volume 2. Background: Computational Structures*, pages 477–563. Oxford University Press, 1992. (Cited on page 55.)

Ketil Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, University of Manchester, 1990. (Cited on pages 107, 113, 234, and 257.)

Ketil Stølen. A method for the development of totally correct shared-state parallel programs. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 510–525, Amsterdam, The Netherlands, August 1991. Springer-Verlag. (Cited on pages 35, 88, 107, 113, and 257.)

Volker Stolz and Eric Bodden. Temporal assertions using aspectJ. *Electr. Notes Theor. Comput. Sci*, 144(4):109–124, 2006. URL `http://dx.doi.org/10.1016/j.entcs.2006.02.007`. (Cited on page 231.)

Martin Strecker. Formal analysis of an information flow type system for MicroJava. Technical report, Technische Universität München, July 2003. (Cited on page 28.)

Aaron Stump. Programming with proofs: Language-based approaches to totally correct software. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 502–509. Springer, 2005. ISBN 978-3-540-69147-1. URL `http://dx.doi.org/10.1007/978-3-540-69149-5`. (Cited on page 166.)

Tachio Terauchi. A type system for observational determinism. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008*, pages 287–300. IEEE Computer Society, 2008. ISBN 978-0-7695-3182-3. URL `http://doi.ieeecomputersociety.org/10.1109/CSF.2008.9`. (Cited on pages 25, 135, 144, 145, 146, and 147.)

Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005. ISBN 3-540-28584-9. (Cited on pages 29 and 118.)

The Washington Post. NSA slideshow on 'The TOR problem', November 2013. URL `http://apps.washingtonpost.com/g/page/world/nsa-slideshow-on-the-tor-problem/499/`. Accessed 18/06/2015. (Cited on page 19.)

Andreas Thums, Gerhard Schellhorn, Frank Ortmeier, and Wolfgang Reif. Interactive verification of statecharts. In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification*

Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report, volume 3147 of *Lecture Notes in Computer Science*, pages 355–373. Springer, 2004. ISBN 3-540-23135-8. URL http://dx.doi.org/10.1007/978-3-540-27863-4_20. (Cited on pages 229 and 235.)

Kerry Trentelman and Marieke Huisman. Extending JML specifications with temporal logic. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 334–348. Springer, 2002. ISBN 3-540-44144-1. URL http://link.springer.de/link/service/series/0558/bibs/2422/24220334.htm. (Cited on page 230.)

Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015*, volume 9035 of *Lecture Notes in Computer Science*. Springer, January 15 2015. ISBN 978-3-662-46680-3 (Print) 978-3-662-46681-0 (Online). URL http://arxiv.org/abs/1501.03063. (Cited on page 9.)

Thomas Tuerk. A formalisation of smallfoot in HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOL*, volume 5674 of *Lecture Notes in Computer Science*, pages 469–484. Springer, 2009. (Cited on page 232.)

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936. (Cited on page 55.)

Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, June 1949. University Mathematical Laboratory. (Cited on page 8.)

Mattias Ulbrich. Software verification for Java 5. Master's thesis, Universität Karlsruhe, 2007. (Cited on page 162.)

US Government. *Classified National Security Information*, December 29 2009. Executive Order 13526. (Cited on page 20.)

Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *Concurrency Theory, 18th International Conference, CONCUR 2007*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271.

Springer, 2007. ISBN 978-3-540-74406-1. (Cited on pages 91, 239, 240, and 255.)

Johan van Benthem. Guards, bounds, and generalized semantics. *Journal of Logic, Language and Information*, 14:263–279, 2005. ISSN 0925-8531. URL http://dx.doi.org/10.1007/s10849-005-5786-y. (Cited on page 55.)

Bart van Delft. Abstraction, objects and information flow analysis. Master's thesis, Chalmers University of Technology and Radboud University Nijmegen, 2011. (Cited on page 28.)

Jeffrey M. Voas. Certifying high assurance software. In *COMPSAC*, pages 99–105. IEEE Computer Society, 1998. ISBN 0-8186-8585-9. URL http://csdl2.computer.org/dl/proceedings/compsac/1998/8585/00/85850099.pdf. (Cited on page 256.)

Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996. URL http://www.cs.nps.navy.mil/research/languages/papers/atsc/jcs.ps.Z. (Cited on page 27.)

Dennis M. Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, pages 156–169, 1997. (Cited on pages 5, 27, and 227.)

Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(1), 1999. URL http://iospress.metapress.com/content/h9dl8k682hrwwygc/. (Cited on pages 245, 246, and 247.)

Klaus von Gleissenthall, Andrey Rybalchenko, and Santiago Zanella Béguelin. Towards automated proving of relational properties of probilistic programs. Invited talk at VPT 2014, 2014. (Cited on page 250.)

Simon Wacker. Blockverträge. Studienarbeit, Karlsruhe Institute of Technology, 2012. (Cited on page 207.)

Andreas Wagner. Trace based reasoning with KeY and JML. Studienarbeit, Karlsruhe Institute of Technology, 2013. URL http://www.key-project.org/DeduSec/2013_Wagner_TraceSpecification.pdf. (Cited on pages 72, 79, and 230.)

Kai Wallisch. Maps in Java dynamic logic. Studienarbeit, Karlsruhe Institute of Technology, April 2014. (Cited on pages 138 and 173.)

Mitchell Wand. A new incompleteness result for Hoare's system. *Journal of the ACM*, 25(1):168–175, January 1978. (Cited on page 77.)

Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999. (Cited on page 231.)

Martijn Warnier. *Language Based Security for Java and JML*. PhD thesis, Radboud University, Nijmegen, The Netherlands, 2006. (Cited on page 249.)

Samuel D. Warren and Louis D. Brandeis. The right to privacy. *Harvard Law Review*, 4(5), December 1890. (Cited on page 1.)

Nathan Wasser. *Automatic Generation of Sepcifications using Verification Tools*. PhD thesis, Technische Universität Darmststadt, 2015. (Cited on page 243.)

Benjamin Weiß. *Deductive Verification of Object-oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, January 2011. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1600837`. (Cited on pages iii, 34, 38, 39, 63, 76, 91, 95, 97, 101, 152, 153, 157, 179, 182, 200, 201, and 255.)

David A. Wheeler. Preventing heartbleed. *IEEE Computer*, 47(8):80–83, 2014. URL `http://dx.doi.org/10.1109/MC.2014.217`. (Cited on pages 7 and 8.)

Niklaus Wirth. Modula: a language for modular multiprogramming. *Software Practice and Experience*, 7:3–35, 1977. (Cited on page 188.)

Pierre Wolper. Temporal logic can be more expressive. In *Proceedings 22nd Annual Symposium on Foundations of Computer Science*, pages 340–348, 1981. (Cited on page 56.)

Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 28(110–111):119–136, June–September 1985. (Cited on page 229.)

Jim Woodcock, Susan Stepney, David Cooper, John A. Clark, and Jeremy Jacob. The certification of the Mondex electronic purse to ITSEC level E6. *Formal Asp. Comput*, 20(1):5–19, 2008. URL `http://dx.doi.org/10.1007/s00165-007-0060-5`. (Cited on page 224.)

Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely–guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997. (Cited on pages 4, 47, 89, 91, 99, 234, and 236.)

Marina Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. PhD thesis, University of Twente, 2015. (Cited on page 239.)

Marina Zaharieva-Stojanovski and Marieke Huisman. Verifying class invariants in concurrent programs. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering, (FASE)*. Springer, 2014. (Cited on page 239.)

Aris Zakinthinos and E. Stewart Lee. A general theory of security properties. In *IEEE Symposium on Security and Privacy*, pages 94–102. IEEE Computer Society, 1997. ISBN 0-8186-7828-3. URL `http://doi.ieeecomputersociety.org/10.1109/SECPRI.1997.601322`. (Cited on page 26.)

Steve Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004. (Cited on page 6.)

Steve Zdancewic and Andrew C. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, June 2001. URL `http://www.cs.cornell.edu/andru/papers/csfw01.ps.gz`. (Cited on page 26.)

Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop*, page 29. IEEE Computer Society, 2003. ISBN 0-7695-1927-X. URL `http://doi.ieeecomputersociety.org/10.1109/CSFW.2003.1212703`. (Cited on pages 14, 23, 25, 26, 33, 118, 135, 136, 144, 145, 146, 147, 245, and 247.)

Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Programming Language Design and Implementation (PLDI)*, pages 349–361, New York, NY, 2008. ACM. (Cited on page 232.)

Andreas Zeller. Specifications for free. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 2–12. Springer Berlin / Heidelberg, 2011. URL `http://dx.doi.org/10.1007/978-3-642-20398-5_2`. (Cited on page 10.)

Ernst Zermelo. Beweis, daß jede Menge wohlgeordnet werden kann. *Mathematische Annalen*, 59(4):514–516, 1904. (Cited on page 45.)

Ernst Zermelo. Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65(2):261–281, 1908. (Cited on page 54.)

Job Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989. (Cited on page 236.)

List of Symbols

$GVar$ global program variables

$LVar$ local program variables

$SVar$ special program variables

V logical variables

$[\cdot]$ box modality

$\langle\cdot\rangle$ diamond modality

`estp` special variable indicating environment steps in execution traces

`heap'` special variable representing the previous heap state

`heap` special variable representing the current heap state

$\langle\cdot\rangle$ singleton sequence

$\langle\rangle$ empty sequence

$\mathcal{D}$ domain

$\mathcal{F}$ functions

$\mathcal{P}$ predicates

$\mathfrak{S}$ signature

$\approx$ low-equivalence

$\divideontimes$ universal agreement relation

$\eqsim^{\Omega}$ object-sensitive low-equivalence

GiB gibibyte (unit of 1024^3 bytes)

$\oplus$ sequence concatenation

`threads` special variable representing the current thread pool

$\mathbb{F}$ field data type

$\mathbb{H}$ heap data type

$\mathbb{L}$ location set data type

$\mathbb{S}$ sequence data type

$\mathbb{T}$ thread pool data type

$\mathbb{B}$ boolean data type

$\mathbb{Z}$ integer data type

List of Abbreviations

ACSL ANSI/ISO C Specification Language

ATL Alternating Time Logic

ABS Abstract Behavioral Specification

ADT abstract data type

ANSI American National Standards Institute

API application programming interface

BMBF Bundesministerium für Bildung und Forschung (Federal Ministry of Education and Research)

BSL Bandera specification language

CAS compare-and-swap

CC Common Criteria

CDL Concurrent Dynamic Logic

CDTL Concurrent Dynamic Trace Logic

COL Common Object Language

CTL Computation Tree Logic

CVE Common Vulnerabilities and Exposures

DFG Deutsche Forschungsgemeinschaft (German National Science Foundation)

DbC Design by Contract

DeduSec Program-level Specification and Deductive Verification of Security Properties

DL dynamic logic

DTL Dynamic Trace Logic

dTL Temporal Dynamic Logic

dWRF deterministic While-Release-Fork language

ESC extended static checking

FIfAKS Formal Information-Flow Analysis of Component-Based Systems

FLOSS free/libre/open source software

FOL first order logic

FSI flexible scheduler-independent

GNI generalized noninterference

GNU GNU's not Unix

GPL GNU General Public License

GUI graphical user interface

HOL higher order logic

ISO International Organization for Standardization

ITL Interval Temporal Logic

JavaDL Java dynamic logic

JDK Java Development Kit

JIF Java with Information Flow

JLS Java Language Specification

JML Java Modeling Language

JMM Java memory model

JRE Java Runtime Environment

JVM Java virtual machine

KOA Kiezen op Afstand

KIT Karlsruhe Institute of Technology

KIV formerly 'Karlsruhe Interactive Verifier'

LSOD low-security observational determinism

LTL Linear Temporal Logic

MODL multi-threaded object-oriented dynamic logic

NSA National Security Agency

OCL Object Constraint Language

OOP object-oriented programing

OSNI object-sensitive noninterference

PVS Prototype Verification System

PDG program dependence graph

PDL propositional dynamic logic of regular languages

PER partial equivalence relation

RAM random access memory

RIFL Requirements for Information Flow Language

RLSOD relaxed low-security observational determinism

RS^3 Reliable Secure Software Systems

SMT satisfiability modulo theories

SV-COMP Competition on Software Verification

SIFUM secure information flow using modes

SLoC single lines of code

SSA static single assingment

SSOD scheduler-specific observational determinism

TINI termination-insensitive noninterference

TIS^{2}NI termination-insensitive semi-strong noninterference

TLS Transport Layer Security

TS^{3}NI termination-sensitive semi-strong noninterference

TSNI termination-sensitive noninterference

UML Unified Modeling Language

VCC formerly 'Verifying C Compiler'

VCG verification condition generation

VMP valid modulo precondition

Index