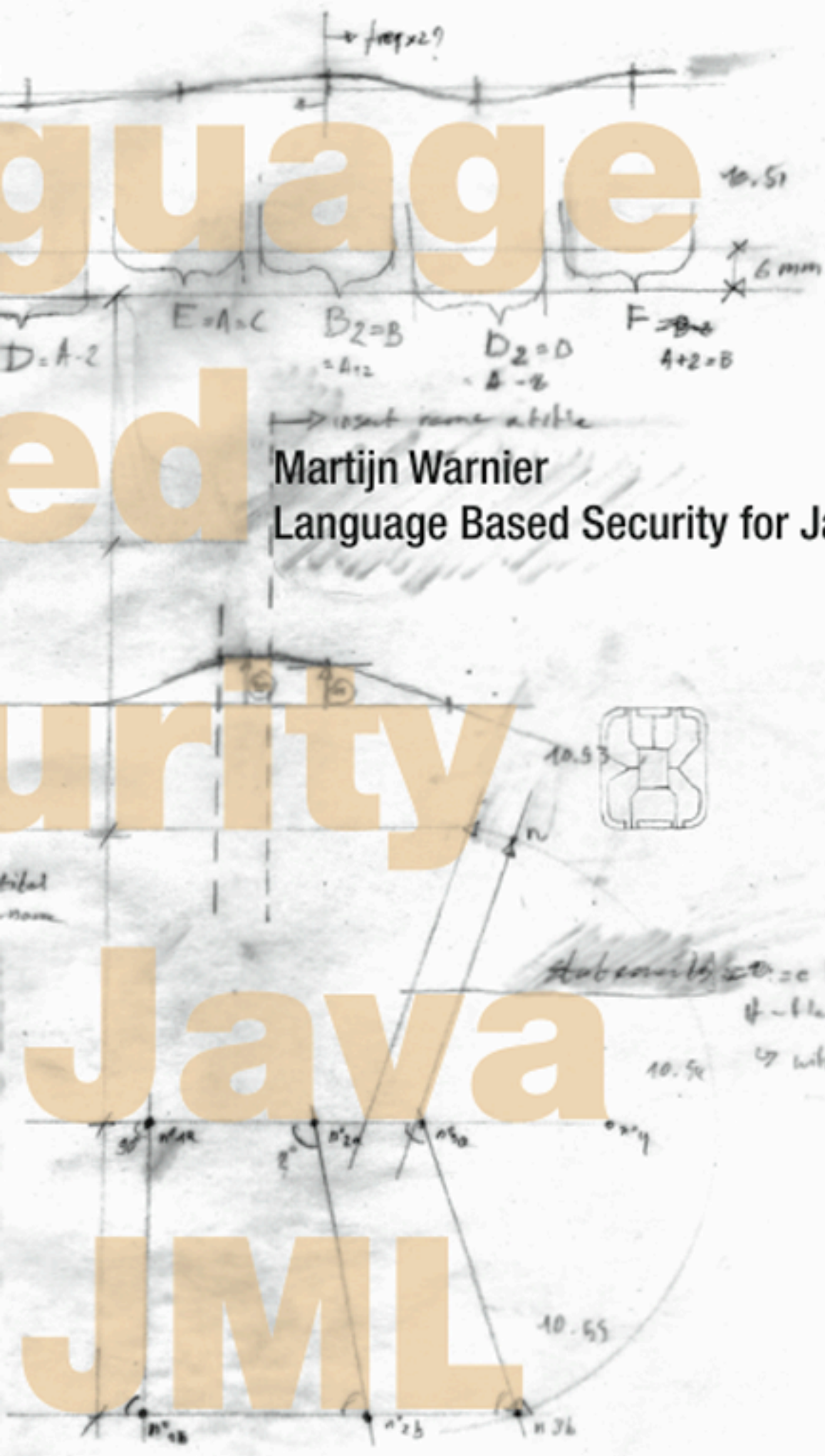

Language Based Security for Java and JML

Martijn Warnier

Language Based Security for Java and JML

Martijn Warnier

ISBN-10: 90-9020922-0
ISBN-13: 978-90-9020922-7
IPA dissertation series 2006-16

Typeset with LaTeX 2ε
All hypertext links are courtesy of the hyperref package
Cover design by Dries Verbruggen, unfold.be
Printed by Print Partners Ipskamp, Enschede

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The author was employed at the Radboud University Nijmegen and funded by the NWO project Security Analysis for Multi-Applet Smart Cards (SAMACS).

Language Based Security for Java and JML

een wetenschapelijke proeve op het gebied
van de Natuurwetenschappen, Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen,
op gezag van de Rector Magnificus prof. dr. C.W.P.M. Blom,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op maandag 27 november 2006
des namiddags om 1.30 uur precies
door

Martinus Elisabeth Warnier

geboren op 20 mei 1976
te Heerlen

Promoter:
Prof. dr. B.P.F. Jacobs
Copromotor:
Dr. M.D. Oostdijk

Manuscriptcommissie:

Prof. dr. P.H. Hartel	University of Twente
Prof. dr. D. Sands	Chalmers University of Technology
Dr. J.R. Kiniry	University College Dublin

Preface

The first time I met Bart Jacobs is now five years ago. At the time I was a student in Utrecht with an interest in Java and its semantics. Jan Bergstra suggested that if I wanted to pursue this interest I should go to Nijmegen and talk to Bart. It turned out that was very good suggestion indeed! After first finishing my Masters thesis in Nijmegen (under Bart's supervision) I was asked to stay on as a PhD student. The result of which you read at this moment.

I learned a lot in these last five years and I had a lot of fun along the way. One can hardly ask for a better atmosphere, both scientifically and socially, as the one at the sixth flour of the FNWI building in Nijmegen were the SoS group resides. In this preface I want to thank everybody who helped me with the writing of this thesis.

The first person I want to thank is Bart Jacobs. He gave me the chance to start as a PhD student in his group. I learned a lot from him about security, theorem proving and other subjects of which some are encountered in this thesis. I'm grateful to know him, both as a researcher and as a person.

I also want to thank Martijn Oostdijk who, as my daily supervisor and copromotor, was always the person who had to read another first draft of a paper or one of the chapters of this thesis. I do not think that anybody has read so much of my writing as he has. He is probably also the person who can best judge how this thesis has improved from its infant state to its current form. These improvements stem in large part from his suggestions and comments on earlier drafts. Another thing I liked was the teaching we did together. Preparing and grading all those exercises was annoying at times, but it certainly taught me a lot about teaching.

Erik Poll also read this entire thesis. His comments are always 'spot on'. I want to thank him for all the valuable comments and suggestions he gave me both on this thesis as well as on other papers and presentations I gave during the years.

My thanks are also due to the members of the reading committee, Pieter Hartel, David Sands and Joe Kiniry, for their comments that improved the overall quality of this thesis considerably. Furthermore I want to thank the following people for reading earlier drafts of chapters of my thesis: Engelbert Hubbers, Christian Haack, Wojciech Mostowski, Wolter Pieters, Ling Cheung and Ruby Groen for proof reading the Dutch summary.

I'm very thankful for being part of such a talented group of people as the SoS group. They are thanked for the fun we had during those endless coffee breaks, lunches and beers we shared. I explicitly want to thank the following (past and present) members of the SoS group: my '*roomie*' Cees-Bart Breunesse, Martijn Oostdijk, Bart Jacobs, Erik Poll, Joachim van den Berg, Jesse Hughes, Ling Cheung, Engelbert Hubbers, Joe Kiniry, Flavio Garcia, David Galindo, Ichiro Hasuo, Wolter Pieters and Harco Kuppens, for teaching me everything I ever wanted to know about '*sugar clumbs*'.

I furthermore want to thank all the PhD students (and others) I met during the years at various summer schools and conferences. The '*Marktoberdorf crowd*', Jeroen Ketema, Hendrik de Haan, Sander Bruggink and Arthur van Leeuwen, are especially thanked. We enjoyed ourselves in Marktoberdorf and we had lots of fun later at various evenings in

Groningen, Utrecht and Nijmegen. I furthermore would like to thank Cas Cremers and Ricardo Corin for organizing, together with myself, the SPAN security workshops. Cas was also a member of the SAMACS project and we had a lot of nice discussions over the years (and shared many glass of beer).

I also want to thank Gilles Barthe for letting me visit his group at INRIA Sophia-Antipolis for the month October in 2004. It was a very nice experience to share my ideas with others then my own SoS group. Thanks are due to Marieke Huisman, Florian Kammüller, Tamara Rezk and the other members of the Everest group for the nice time I had there.

I couldn't have completed this thesis without my friends which I want to thank here for all the fun we had over the years. You all made sure that I got some much needed diversion from my thesis and work in general. I explicitly want to thank: Juriaan Nortier, Daan Moes, Kees Hordijk & Els Bon, Rosha Atsma, Ruby Groen, Thijs Broersma ánd the Johnson sisters.

Finally, and foremost, I want to thank my family: Claire Warnier & Dries Verbruggen, it's always fun to visit the two of you in Antwerpen. Miriam Warnier & Martijn Schliekelmann, thanks for cooking for me all those times. The food is always great, as is the company. And finally I want to thank my parents, Bart & Ine Warnier, *"Bedááinkt veur alles, en zoeveul mie"*.

Martijn Warnier
May 2006
Nijmegen

Contents

Chapter 1

Introduction

"If it is provably secure, it is probably not."

–Lars Knudsen on block ciphers [And01]

As recently as 10 years ago, computer security was of importance to only a few insiders – typically in domains such as banking, the military or intelligence. Nowadays it is of interest to the general public: newspapers open with computer security headlines[1] and individual users of computers are plagued by spam, (computer) viruses and phishing attacks.

Computer security is also important to society at large. A recent trend –at least in the Netherlands– is the centralization of electronic databases. For example, in the medical field, where insurance companies, hospitals and family doctors share patient data or in the judicial systems, where the court shares (sensitive) information with the police but also with organizations such as child protection and social services. Many other electronic databases contain security sensitive information. Examples range from credit card numbers, stored by online stores, to biometric data such as digital fingerprints, required for the new Dutch passport and planned to be stored in a central database. All of these databases form attractive targets for attackers and thus need adequate (computer security) protection mechanisms. Clearly computer security is an important subject, from a popular as well as a scientific point of view.

The main subject of this thesis is *language based security*. In this field, the object of study is security on the level of programming languages. Such programming languages are studied on a semantical level and the typical goal is to (formally) prove that certain security properties hold for *classes of programs*. The field of *program verification* is closely related to language based security. However, in program verification the typical goal is to prove certain (possibly security related) properties for *specific programs*. In practice

[1] E.g., the opening headline of the Dutch national newspaper the NRC on Saturday the 7th of January 2006 was: "Cybercrime intensifies – PC viruses are mainly the work of organized crime" [translation by the author].

the distinction between language based security and program verification is somewhat arbitrary. Indeed, in this thesis we will study both: the first part of the thesis (Chapters 3 through 5) deals primarily with program verification while the second part (Chapters 6 through 8) studies language based security techniques.

In theoretical computer science, security is usually thought of as a battle against an (implicit) attacker, who has more resources than a user. An example is the Dolev-Yao attacker [DY83], used in the study of security protocols, who has complete control of a (computer) network. A system is deemed secure if, in such an attacker model, the attacker cannot reach his malicious goal, such as learning a secret key or altering the balance of a bank account. Typically, in theoretical computer science, this will also mean that one abstracts away from certain low-level details. One of the goals of the research performed in this thesis is to reduce this abstraction level by looking directly at the level of the programming language. Obviously, there are still numerous implicit assumptions, for instance, about the computer hardware, the compiler and the end-user of a program.

Thesis outline

Below we summarize the contents of each chapter. We also indicate the original publication associated with each chapter and credit co-authors.

Chapter 2: Background and preliminaries

In this chapter we give an introduction to several topics that serve as background for this thesis. The subjects presented here give the bare minimal background that is needed to understand the remainder of the thesis.

The study of programming languages is central to this thesis. Concretely we look at Java and especially the version of Java tailored to smart cards called Java Card. In this thesis we have formalized a number of verification frameworks that try to enforce security properties like confidentiality at the level of the programming language. Obviously, in practice the results of such analysis methods need to be integrated with security mechanisms on other levels such as hardware, access control or communication protocols. However, in this thesis we focus on the level of programming languages –hence the focus on *language based* security in the title.

Since not all analysis methods have been worked out for full sequential Java (and also for presentation purposes) we first define a semantics for a simple while-like [NN92] programming language. We use this simpler language to explain the basic ideas behind the proposed verification techniques. Giving a complete semantics of sequential Java lies outside the scope of this thesis. We restrict ourselves to giving some highlights of the LOOP semantics developed in Nijmegen [JP04] and refer to two earlier PhD theses [Hui01, Bre06] in our group for more details on the LOOP verification framework. A brief overview of other research on Java semantics is also given in this chapter. The Java Modeling Language (JML [LBR99b]) is used as specification language throughout the first part of this thesis.

Program verification techniques form one of the main subjects of this thesis. They originated in the late 60's with the work of Floyd [Flo67] and Hoare [Hoa69]. Over the years a lot of work has been performed in this area, although it is usually only applied to small (toy) examples and programming languages. Languages like JAVA CARD—which are complex enough to be used in practice, yet simple enough for application of formal methods– have boosted the work on verification techniques considerably. Contacts with the smart card industry during the Verificard project [Ver] already showed a clear interest in program verification techniques and, at the time of writing, even industry giants like Microsoft appear to be applying program verification techniques [BRLS05] to (parts of) commercial projects.

The (relative) success of program verification techniques together with a political climate that is favorable towards funding of security-related research boosted computer security research in recent years. Confidentiality (for programs) or secure information flow has been studied since the late seventies [Den76]. The idea is that we want a (provably sound) analysis technique that enforces a *secure information flow policy* for a program. Such a policy typically specifies what information –processed by the program– is secret and what information is publicly available. A standard solution for this problem is the use of *access control modifiers*, such as Java's `public`, `private` and `package` modifiers. But access control modifiers do not guarantee secure information flow of a whole program. We explain the concept of confidentiality in terms of non-interference, give an overview of the field and place our own research in the broader context of language based information flow security [SM03].

The final part of this introductory chapter discusses the tools that have been used in this thesis: ESC/Java2, the LOOP verification framework and the theorem prover PVS. Some other tools are also briefly discussed.

Chapter 3: Specification and verification of Java programs

This chapter shows some of the challenges one encounters when program (fragments) written in Java need to be formally verified. We aim to give a canonical set of examples for (Java) program verification methods, but the emphasis lies more on the underlying semantical issues than on verification. All examples are specified in JML and verified with both the LOOP verification framework and the ESC/Java2 tool. We comment on significant differences between the two.

The chapter is based on the "Java Program Verification Challenges" [JKW03] by Bart Jacobs, Joseph Kiniry and the author. While the main content stayed the same as in the paper, all examples have been rechecked in the latest version of ESC/Java2. Furthermore, several specifications have been refined, and one additional example has been added (in Section 3.3.2).

Chapter 4: Specification and verification of control flow properties

The chapter discusses a case study in the design, development and formal verification of secure smart card applications. An elementary JAVA CARD electronic purse applet is presented, whose specification can be simply formulated as 'in normal operation, the applet's balance field may only be decreased, never increased'. The applet features a challenge-response mechanism that permits legitimate terminals to increase the balance by setting the applet into a special operation mode.

A systematic approach is introduced to guarantee a secure flow of control within the applet: appropriate transition properties are first formalized as a finite state machine, then incorporated in the specification, and finally formally verified using the Loop verification framework together with the PVS theorem prover.

This chapter originally appeared as "Source Code Verification of a Secure Payment Applet" [JOW04] by Bart Jacobs, Martijn Oostdijk and the author. The contents have been completely revised and updated to the latest insights in this research area.

Chapter 5: Specification and verification of non-interference in JML

This short chapter serves as a bridge between the previously mentioned program verification-based chapters and the subsequent non-interference-based ones. It deals with the specification and verification of non-interference properties, in particular confidentiality, in JML.

The notion of a specification pattern for JML is introduced and it is shown how such patterns can be used to specify non-interference properties such as confidentiality and integrity. The resulting specifications can be verified using any of the JML tools.

This chapter has not been published before and appears here for the first time in print. The work discussed here originated in discussions with Bart Jacobs and Erik Poll.

Chapter 6: Statically checking termination-insensitive non-interference

This chapter presents a new approach for verifying confidentiality for programs, based on abstract interpretation. The notion of confidentiality that is analyzed in this chapter is the one most commonly found in the literature: termination insensitive non-interference, which means that confidentiality is only guaranteed in case the program terminates normally.

The framework is formally developed in the theorem prover PVS. Dynamic labeling functions are used to abstractly interpret programs via modification of security levels of variables. Our approach is sound and compositional and results in an algorithm for statically checking confidentiality. The working of the algorithm is illustrated on some examples and we briefly discuss how to extend this algorithm to check confidentiality for full Java (and which difficulties might arise there).

The main advantage of the proposed approach is that, while it is still sound and automatic, it is less restrictive, i.e., rejects fewer non-interfering programs, than standard type-checking based approaches. The chapter is based on "Statically checking confidentiality via dynamic labels" [JPW05] together with Bart Jacobs and Wolter Pieters.

Chapter 7: Interactively proving termination-sensitive non-interference

This chapter discusses a new approach for checking the stronger confidentiality property –termination *sensitive* non-interference– that is assured for all termination modes of a program. A programming language such as Java has three termination modes: the program can terminate normally, or it can terminate abruptly via an exception, or it can fail to terminate at all (diverges)[2]. Bisimulations for Java classes are used to formally define non-interference, which can be proved using a relational Hoare logic. This relational Hoare logic has been specified on top of the LOOP semantics. These rules are realized as (provable) lemmas in LOOP's Java semantics in PVS.

The main contribution of this chapter is a new verification framework for proving a strong form of confidentiality for sequential Java. The framework is both sound and complete. It uses a mix of reasoning on a syntactic level, with the relational Hoare logic, and direct reasoning at a semantical level. The price we pay for completeness of our approach is that the proofs now need to be provided by a (human) user. Interactive reasoning in PVS is necessary to obtain these proofs, though powerful tactics allow the user to only concentrate on the complicated parts.

The research presented in this chapter has not been published before and appears here for the first time in print. The work discussed here originated in discussions with Bart Jacobs.

Chapter 8: Enforcing time-sensitive non-interference

Timing channels constitute one form of covert channels through which programs may be leaking information about the confidential data they manipulate. Such timing channels are typically eliminated by design, employing ad-hoc techniques to avoid information leaks through execution time, or by program transformation techniques, that transform programs which satisfy some form of non-interference property into programs that are time-sensitive termination-sensitive non-interfering. However, existing program transformations for this purpose are confined to simple languages without objects or exceptions.

This chapter introduces a program transformation that uses a transaction mechanism to prevent timing leaks in sequential object-oriented programs. Under some strong but reasonable hypotheses, the transformation preserves the semantics of programs and yields, for

[2]This only describes the situation as can be observed from the outside. Internally Java programs can also terminate abruptly via a `return` or a (labeled or unlabeled) `break` or `continue` statement. Also see Section 7.3.1.

every termination-sensitive non-interfering program, a time-sensitive termination-sensitive non-interfering program. It is based on "Preventing timing leaks through transactional branching instructions" [BRW06] by Gilles Barthe and Tamara Rezk and the author.

Chapter 9: Conclusions

This chapter revisits the conclusions of each individual chapter and places them in a broader context. Some general conclusions of the thesis will be drawn here and we end with suggestions for future work.

We have chosen to make each of the main chapters (Chapter 3 through Chapter 8), where possible, self-contained. In particular each chapter has the same setup: it starts with an introduction of the topic at hand, then presents the main research contribution and ends the chapter with (closely) related work, conclusions and suggestions for future work. As a result of this choice, there is some overlap between the different chapters.

Chapter 2

Background and preliminaries

This chapter serves as an introduction to the different concepts, tools and theories used in this thesis. While far from complete, it hopefully gives the reader enough background to appreciate the complexities involved. The chapter furthermore gives a general overview of related work. Subsequent individual chapters will also have a section on more closely related work.

We start with some context, in particular we give a general description of the Java Card programming language and associated framework. Section 2.2 briefly discusses the topic of semantics of programming languages, with a focus on Java (card). Section 2.3 introduces the specification language JML and Section 2.4 gives some background on program verification techniques, again focusing on Java (card). Section 2.5 introduces the notion of confidentiality as non-interference and this chapter ends with a discussion of the (main) tools that were used in the research discussed in this thesis.

2.1 Java Card

The Java Card [Che00] programming language is a Java-like language that is specifically tailored for writing applications (called applets) for smart cards. Smart cards are small computers, usually without an independent power supply, and with a limited interface to the outside world. Typical examples of smart cards are SIMs in mobile phones and electronic wallets –like the Dutch Chipknip [Chi] or the Belgian Proton [Pro] systems.

Since smart cards are typically used to store sensitive information it is important that the cards are *tamper resistant* [RE00]. This usually means that it is hard to obtain secret information from a card (confidentiality is maintained) and that it is hard to change sensitive information on a card or forge a card (integrity is maintained). In this context 'hard' can mean a lot of things. One of the more useful criteria in this context is *economical feasibility*: if the economic costs for an attacker to break a smart card are higher than the economic gains then the smart card is deemed *economically secure*. Of course it is possible that an attacker has another motive for breaking the system, either because of a certain ideology or simply out of curiosity. We refer to [And01, Sch00] for an extensive discussion

on this subject.

In the JAVA CARD framework all communication between a card and a client occurs by exchanging Application Protocol Data Units (APDUs). The international ISO7816 standard (part 4) [ISO] describes the structure of APDUs. In a JAVA CARD applet these are represented as simple byte arrays with little more structure than a header containing instructions as to which method should be called, information about the length of the data and some simple (optional) parameters.

Since APDUs provide a low level way of communication, a lot of the effort in programming JAVA CARD applets and clients is spent on finding the correct translation from the high level Java (card) method calls to low level APDUs and vice-versa. Since version 2.2, the JAVA CARD standard also allows the use of remote method calls using the JAVA CARD remote method invocation (JCRMI) framework [OW03].

JAVA CARD-enabled smart cards contain a small operating system which includes a Java virtual machine (JVM [LY99]). The JAVA CARD language forms a superset of a subset of Java. JAVA CARD is a subset in the sense that it supports only single-threaded programs and some larger integral data types (such as `long`, `float` and `double`) are omitted. It is a superset because there are some JAVA CARD-specific constructs such as an applet firewall mechanism, which ensures that multiple JAVA CARD applets can be loaded safely on a card without compromising security, and a dedicated cryptographic API.

Another difference between Java and JAVA CARD is that the latter distinguishes between two kinds of memory: transient and persistent memory. Transient memory has a temporary nature (that is guaranteed by the underlying hardware), its contents is lost after a card tear. In contrast, persistent memory preserves its contents even if the card has no power. Determining which kind of memory to use in what situation forms one of the –low level– challenges in the implementation of JAVA CARD applets. Performance and security typically need to be balanced here. Persistent memory – implemented in EEPROM– is a lot slower then the fast volatile memory that is used to realize transient memory, but in some situations security sensitive information, e.g., an encryption key, needs to be available for many sessions. Such low level details make reasoning about JAVA CARD both interesting and non-trivial.

JAVA CARD is also ideal for application of formal methods, since the language is small enough to be actually feasible for study from a formal perspective, yet it is used in real world applications.

2.2 Semantics of programming languages

This thesis has the ambition to (formally) reason about properties for full sequential Java. However, in the next section we first introduce a relatively simple imperative programming language that does not have any object-oriented features. The reason for introducing this language is twofold:

i. For presentation purposes it is better to first explain a new verification technique for a simpler programming language. Once the reader is familiar with the basic ideas

behind such a new verification technique, the simple programming language can be extended with more esoteric features.

ii. It is not feasible to extend all the verification techniques presented in this thesis to full sequential Java within the limited time associated with a PhD project. We have chosen to work out some techniques (e.g., in Chapter 7 and Chapter 8), extend other verification frameworks only with such things as method calls and abrupt termination and to briefly discuss how possible extensions towards a language such as Java might be formalized –without going into all the technical details.

In the next section we first introduce the syntax and semantics of a simple imperative programming language called WHILE. The subsequent section briefly discusses the topic of semantics of Java (card).

2.2.1 Semantics of While-like languages

We introduce the imperative programming language WHILE[3] with statements and expressions (with side-effects). The BNF of WHILE is given in Definition 1 below.

Definition 1 (WHILE BNF)**.**

$$\begin{array}{lrll} \text{Expressions} & \mathsf{e} & ::= & n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{e}_1 == \mathsf{e}_2 \mid \mathsf{e}_1 < \mathsf{e}_2 \mid \\ & & & \mathsf{e}_1 + \mathsf{e}_2 \mid \mathsf{e}_1 - \mathsf{e}_2 \mid \mathsf{e}_1 \times \mathsf{e}_2 \mid \mathsf{e}\texttt{++} \\ \text{Statements} & \mathsf{s} & ::= & v := \mathsf{e} \mid \texttt{skip} \mid \mathsf{s}_1; \mathsf{s}_2 \mid \texttt{if-then}(\mathsf{e})(\mathsf{s}) \mid \\ & & & \texttt{if-then-else}(\mathsf{e})(\mathsf{s}_1)(\mathsf{s}_2) \mid \texttt{while}(\mathsf{e})(\mathsf{s}) \end{array}$$

with $v \in \mathsf{Var}$ and $n \in \mathbb{Z}$.

The language WHILE supports side effects in expressions (via the ++ operator), as well as the usual language constructs assignment, composition, branching and repetition. This language will be the starting point for the new verification techniques proposed in the second half of this thesis.

The semantics of the language is rather standard. In Figure 2.1 we give the natural semantics of WHILE. In what follows we denote the state (or memory) with M. The syntactic expression $[\![\mathsf{e}]\!]\mathsf{M}$ denotes the result value of evaluating expression e in memory M. The remainder of the semantics of expressions is completely standard and omitted (see for example [NN92, Sch86]). As mentioned before, we will use WHILE to explain the main idea and prove some basic properties of new verification techniques. Since our ultimate goal is to cover full (sequential) Java, WHILE will be extended with features such as method calls, abrupt termination modes and object-oriented aspects.

[3]Except for the fact that our language supports side-effects in expressions, WHILE is the same as the language with the same name used in [NN92]

$$\frac{}{\langle v := \mathsf{e}, \mathsf{M}\rangle \rightsquigarrow \mathsf{M}[v \mapsto [\![\mathsf{e}]\!]\mathsf{M}]} \qquad \frac{}{\langle \texttt{skip}, \mathsf{M}\rangle \rightsquigarrow \mathsf{M}}$$

$$\frac{\langle \mathsf{s}_1, \mathsf{M}\rangle \rightsquigarrow \mathsf{M}' \quad \langle \mathsf{s}_2, \mathsf{M}'\rangle \rightsquigarrow \mathsf{M}''}{\langle \mathsf{s}_1; \mathsf{s}_2, \mathsf{M}\rangle \rightsquigarrow \mathsf{M}''}$$

$$\frac{[\![\mathsf{e}]\!]\mathsf{M} = \mathsf{true} \quad \langle \mathsf{e}, \mathsf{M}\rangle \rightsquigarrow \mathsf{M}' \quad \langle \mathsf{s}, \mathsf{M}'\rangle \rightsquigarrow \mathsf{M}''}{\langle \texttt{if-then}(\mathsf{e})(\mathsf{s}), \mathsf{M}\rangle \rightsquigarrow \mathsf{M}''}$$

$$\frac{[\![\mathsf{e}]\!]\mathsf{M} = \mathsf{false} \quad \langle \mathsf{e}, \mathsf{M}\rangle \rightsquigarrow \mathsf{M}'}{\langle \texttt{if-then}(\mathsf{e})(\mathsf{s}), \mathsf{M}\rangle \rightsquigarrow \mathsf{M}'}$$

$$\frac{[\![\mathsf{e}]\!]\mathsf{M} = \mathsf{true} \quad \langle \mathsf{e}, \mathsf{M}\rangle \rightsquigarrow \mathsf{M}' \quad \langle \mathsf{s}_1, \mathsf{M}'\rangle \rightsquigarrow \mathsf{M}''}{\langle \texttt{if-then-else}(\mathsf{e})(\mathsf{s}_1)(\mathsf{s}_2), \mathsf{M}\rangle \rightsquigarrow \mathsf{M}''}$$

$$\frac{[\![\mathsf{e}]\!]\mathsf{M} = \mathsf{false} \quad \langle \mathsf{e}, \mathsf{M}\rangle \rightsquigarrow \mathsf{M}' \quad \langle \mathsf{s}_2, \mathsf{M}'\rangle \rightsquigarrow \mathsf{M}''}{\langle \texttt{if-then-else}(\mathsf{e})(\mathsf{s}_1)(\mathsf{s}_2), \mathsf{M}\rangle \rightsquigarrow \mathsf{M}''}$$

$$\frac{[\![\mathsf{e}]\!]\mathsf{M} = \mathsf{true} \quad \langle \mathsf{e}, \mathsf{M}\rangle \rightsquigarrow \mathsf{M}' \quad \langle \mathsf{s}, \mathsf{M}'\rangle \rightsquigarrow \mathsf{M}'' \quad \langle \texttt{while}(\mathsf{e})(\mathsf{s}), \mathsf{M}''\rangle \rightsquigarrow \mathsf{M}'''}{\langle \texttt{while}(\mathsf{e})(\mathsf{s}), \mathsf{M}\rangle \rightsquigarrow \mathsf{M}'''}$$

$$\frac{[\![\mathsf{e}]\!]\mathsf{M} = \mathsf{false} \quad \langle \mathsf{e}, \mathsf{M}\rangle \rightsquigarrow \mathsf{M}'}{\langle \texttt{while}(\mathsf{e})(\mathsf{s}), \mathsf{M}\rangle \rightsquigarrow \mathsf{M}'}$$

Figure 2.1: Natural semantics for WHILE.

2.2.2 Semantics of Java-like languages

Obviously, a proper semantics of an object-oriented language –and in particular Java– is more complicated then the semantics of the simple language described above. These complications stem from, amongst others, the following language features:

- **method calls** Method calls form an essential abstraction mechanism that is present in any practical programming language.
- **data types** Java supports several native data types for integers and floating point data types. Pointers to objects form another crucial language concept.
- **multiple termination modes** Java statements can hang, terminate normally or terminate abruptly. Abrupt termination is further refined in exceptional termination

or termination via a `return`, `break` or `continue` statement.

- **inheritance** Field hiding, method overriding and late binding are all mechanisms that (should) enhance modularization and re-usability of Java programs. These features also complicate the semantics of Java considerably.
- **parallelism** Threaded execution complicates Java's semantics even further. Deadlock and live locks and race conditions are among the difficulties involved here. In this thesis we will not discuss this matter further and will only look at the sequential part of Java. A classical reference in this context is [CKRW99], later works include [RM02, MBRS04].

Giving a (formal) description of Java's semantics is infeasible in the context of this thesis. Instead we refer to [AF99] for an introduction to a formal semantics of Java and Java Card. Another issue is that Java is a language that still changes considerably, thus its semantics also changes. See e.g., [MP05] for a discussion of the memory model in Java's latest incarnation (major version J2SE 1.5).

In the remainder of the thesis we shall give the necessary (semantic) background where relevant, in particular we give a small overview of the Java semantics for sequential Java developed in the LOOP [JP04] project in Section 7.3.1.

2.3 JML

The Java Modeling Language, JML [LBR99b, JML], is a behavioral interface specification language designed to specify Java modules. It can be used for classes as a whole, via class invariants and constraints, and for the individual methods of a class, via method specifications consisting of pre-, post- and frame-conditions (assignable clauses). In particular, it is also possible within a method specification to indicate if a particular exception may occur and which post-condition results in that case. We only describe the JML constructs that are used in the thesis here and we refer to the JML reference manual [LPC+05] for a full description of all JML's features.

JML annotations are to be understood as predicates or relations (on the memory M), which should hold for the associated Java code. These annotations are included in the Java source files as special comments indicated by `//@`, or enclosed between `/*@` and `*/`. They are recognised by special tools like the JML run-time checker [CL02a], the LOOP compiler [BJ01], the static checker ESC/Java [FLL+02] or the Krakatoa verification condition generator [CDF+04], see [BCC+05] for an overview of JML tools.

Class invariants and constraints are written as follows:

JML

```
/*@                              /*@
  @ invariant <predicate>          @ constraint <relation>
  @*/                              @*/
```

An invariant must hold after termination of constructors, and also after termination (both normal and exceptional) of methods, provided it holds before. Thus, invariants are implicitly added to postconditions of methods and constructors, and to preconditions of normal (non-constructor) methods. A constraint is a relation between two states, expressing what should hold about the pre-state and post-state of all methods. In this thesis we use a constraint in Chapter 4, and invariants in Section 3.3.4 and again in Chapter 4. In practice they contain important information, and make explicit what a programmer had in the back of his/her mind when writing a program.

Next we give an example JML method specification of some method `m()`.

JML

```
/*@ behavior
  @    requires <precondition>;
  @  assignable <items that can be modified>;
  @     ensures <normal postcondition>;
  @     signals (E) <exceptional postcondition>;
  @*/
public void m();
```

Such method specifications may be understood as an extension of correctness triples $\{P\}m\{Q\}$ used in classical Hoare logic [Hoa69], because they allow both normal and exceptional termination. Moreover, the postconditions in JML are relations, because the pre-state, indicated by `\old(--)`, can be referred to in postconditions . We shall see many method specifications below.

In JML it is not necessary to specify behavior completely. JML allows a specification style where a user only specifies what interests him. This is called a lightweight specification. In this case other behaviors are simply ignored. In contrast, in a heavyweight specification case, JML requires that the user is fully aware of the defaults involved. In a heavyweight specification case, JML expects that a user only omits parts of a specification when the user believes that the default is satisfactory [LPC+05, §2.3]. These default values are `true` for all the clauses except for the `diverges` clause (for which the default is `false`) and the `assignable` clause (for which the default is `\everything`). Heavyweight specifications also use appropriate behavior keywords –`normal_behavior` if a method *must* terminate normally[4], `exceptional_behavior` if a method *must* terminate with an exception[5] and simply `behavior` in all other cases. Throughout this thesis we will use heavyweight JML specifications.

JML is intended to be usable by Java programmers. Its syntax is therefore very much like Java's. However, it has a few additional keywords, such as `==>` (for implication), `\old` (for evaluation in the pre-state), `\result` (for the return value of a method, if any), and `\forall` and `\exists` (for quantification).

[4]The `signals` and `diverges` clauses have the value `false` in this case.
[5]The `ensures` and `diverges` clauses have the value `false` in this case.

This thesis will not pay much attention to the semantics of JML (interested readers should read [LPC+05] for more details). Hopefully, most of the JML assertions are self-explanatory. However, there are three points that we would like to mention explicitly:

- In principle, expressions within assertions (such as an array access `a[i]`) may throw exceptions. The JML approach, see [LPC+05], is to turn such exceptions into arbitrary values. E.g., the expression `a[i] == 0` can have any truth value –true or false– if array `a` is `null`. Of course, one should try to avoid such exceptions by including appropriate requirements. For instance, for the expression `a[i]` one should add `a != null && i >= 0 && i < a.length`, in case this is not already clear from the context. This is what we shall always do. Also see [CR05] for a proposal to change the defaults in JML to obtain these specification cases 'for free'.

- JML uses the behavioral subtype semantics for inheritance [LW94]. This means that overriding methods in subclasses must satisfy the specifications of the overridden ancestors in superclasses. This is a non-trivial restriction, but one which is essential in reasoning about methods in an object-oriented setting. The specification of such an overridden method and its ancestor can contain JML keywords `\typeof` and `\type`, referring to the dynamic type of an object and type of a class respectively. These keywords are used to specify which parts of the assertion is relevant in a certain context[6] (see, for instance, Section 3.4).

- JML method specifications yield proof obligations. But they can also, once proved, be used as facts in correctness proofs of other methods. In that case one first has to establish the precondition and invariant of the method that is called, and subsequently one can use the postcondition in the remainder of the verification.

  An alternative approach is to reason not on the basis of the specification, but on the basis of the implementation of the method that is called[7]. Basically, this means that the body of the called method gets substituted at the appropriate place. However, this may lead to duplication of verification work, and makes proofs more vulnerable to implementation changes. In general, if no specification is available, one is forced to reason with the implementation.

  In the remainder of this thesis we shall see illustrations of both alternative approaches.

JML also supports so-called `model` fields [BP03]. A model field is a field that is not accessible by the Java code, i.e., a specification-only field. It should be thought of as the abstraction of one or more concrete fields [CLSE05]. Model fields are related to concrete fields via JML's `depends` clause, which indicates on which concrete fields a model field depends, and a `represents` clause, which specifies exactly how a model field is related to concrete fields.

[6] Thereby basically circumventing behavioral subtyping in specifications.

[7] This only works if one actually knows the run-time type of the object on which the method is called.

Finally, the `ghost` keyword indicates another specification-only field in JML. Values of ghost fields are not determined by `represents` clauses; instead, they are assigned directly using the JML keyword `set`. Values of ghost fields can be changed in assertions *inside* the body of a method. A consequence of the use of ghost fields is that assertions have state.

For readers unfamiliar with JML, this thesis may hopefully serve as an introduction via examples. More advanced use of JML in specifying API-components may be found for instance in [PBJ00][8]. We wish to stress that JML is a very active research project and while the core of JML is relatively stable, more esoteric features are still undergoing major changes.

2.4 Java program verification

Program verification techniques originated in the late 60s with the work of Floyd [Flo67] and Hoare [Hoa69]. Over the years a lot of work has been performed in this area, although it is usually only applied to small (toy) examples and programming languages (such as WHILE from Definition 1). Classical Hoare logics allow one to reason about simple imperative languages without side-effects, without different termination modes and without object-oriented features such as inheritance. While interesting from a theoretical perspective, verification techniques based on such logics were rarely if ever used on programs used in 'real world' applications[9].

Java, and especially Java Card,changed all this. Mobile Java programs (applets) which could be executed on the client side became popular at the dawn of the Internet era. Security concerns about such programs led to paradigms such as *proof carrying code* [NL97], which in turn led to more interest and research in program verification. A parallel development was the emergence of security-sensitive Java Card applications. These programs were so small[10] that (formal) verification actually seemed feasible. This perception boosted the development of logics and tools for applying program verification techniques, both in academia and industry.

Two kind of approaches for Java program verification can be distinguished: those that attempt to verify correctness conditions for Java *bytecode* and approaches for verifying formal specifications for Java *source code*. We will focus in this thesis on source code and refer to [Moo99, Boy03] which give an overview of related work on bytecode verification[11]. In what follows we give an overview of what (in our view) are the most important program verification approaches and tools for JML-like languages and refer to [HM01] for an overview of program verification techniques for Java source and bytecode in general.

Perhaps the easiest verification technique[12] employed, at least from a users perspective,

[8] See also on the web at `http://www.cs.ru.nl/~erikpoll/publications/jc211_specs.html` for a specification of the Java API for smart cards.

[9] One notable exception is the verification work on Ada, see e.g., [GMP93].

[10] In terms of lines of code.

[11] Note that this refers to program verification on Java bytecode and not the type-checking as preformed by Java's standard bytecode verifier.

[12] Note that this is not verification in the sense of Hoare logics.

is what is called *runtime assertion checking*. Annotations are added to the Java source code and a dedicated compiler transforms the annotated source code to a program that includes runtime checks (based on the annotations). If at runtime a check fails the program will typically throw an exception or error. Java's own `assert` construct[13] is probably the best know example of a runtime assertion checking system. Other examples that extend Java's primitive assert construct to the *design by contract* [Mey92] paradigm include the JML runtime assertion checker [CL02a] (which translates JML specifications in runtime checks), iContract [Kra98, Ese01] (a runtime assertion checker that uses OCL [RG02] as a specification language), and Jass [BFMW01, Jas] (another runtime assertion checker that uses a JML-like syntax for specifications).

The original ESC/Java [FLL+02] tool –developed at the Digital Equipment Corporation's Systems Research Lab– uses a subset of JML and tries to check JML specifications without user interaction. This is accomplished by an executable weakest-precondition calculus [Dij76] for Java that feeds its proof obligations to the automatic theorem prover Simplify [DNS05]. Though neither sound nor complete, the tool is remarkably useful in practice. Verification of array bound and (the absence of) null-dereferences are among the more typical things that ESC/Java excels in. ESC/Java was eventually made open-source and its successor ESC/Java2 [CK04, ESC] can handle (almost) full JML. In Section 2.6.1 below we discuss ESC/Java2 in more detail. Another tool that follows this approach is the Boogie tool –developed at Microsoft research– that statically verifies JML-like annotations written in Spec# [BRLS05] for the C# programming language.

The most precise tools –that, not surprisingly, also require the most user interaction– are those based on interactive theorem provers. The Jive [MMPH00] tool was originally developed at the University of Kaiserslautern and is currently a joint project of Kaiserslautern and ETH Zurich. A language similar to JML is used as a specification language and, like ESC/Java, it uses a syntactic Hoare logic for verification of annotated Java programs. Proof obligations can be fed to the (interactive) theorem provers PVS [ORSH95] (see Section 2.6.3 below) and Isabelle [Pau94]. A user can then try to prove the correctness of these proof obligations. The KeY [ABB+04] tool uses a similar approach. Specifications can be written in JML and OCL. A customized dynamic logic [Bec01] is used to reason about annotated Java (card) programs and a dedicated interactive theorem prover –the key prover– can be used to prove the correctness of the annotations. Yet another tool that follows such an approach is the Krakatoa [CDF+04, The02] tool. It uses JML as annotation language, the verification condition generator Why [Fil], and the theorem prover of choice is Coq [Coq]. The LOOP [JP04] verification project, developed at the Radboud University Nijmegen, uses JML as specification language. The LOOP tool [BJ01] translates annotated Java source files into PVS theories which include the proof obligations. We discuss the LOOP verification frameworks in more detail in Section 2.6.2 below. A final tool that can be used for verifying JML annotated Java source code is the Jack tool [BLR02]. The tool was originally developed at Gemplus research and is currently maintained at INRIA, Sophia-Antipolis as part of the Everest project. It also uses a fully interactive Hoare logic

[13] Present in Java since J2SE 1.4.

and can feed its prove obligations to both Simplify and interactive theorem provers such as B [Abr96], Isabelle, and PVS.

2.5 Confidentiality as non-interference

Confidentiality is one of the basic notions in the study of computer security. Many different (sub)disciplines within computer science and cryptography have studied it. Examples include the study of security protocols that aim (amongst others) to keep sensitive information secret, as well as the many cryptographic schemes studied in the field of cryptography. The focus of this thesis is confidentiality from a programming language perspective. In particular the question how one can ensure that public information is completely independent from sensitive (secret) information is central to this thesis.

Enforcing high level security properties –such as confidentiality– for programs is a non-trivial task. Typically, modern programming languages –like Java– try to solve this problem by using access control modifiers to ensure confidentiality (see [GJSB00, § 6.6]). However access control modifiers cannot ensure end-to-end security. Language based information flow security forms the subject within computer security that tries to enforce end-to-end security for programs [SM03].

Confidentiality in programming languages has been studied since the seventies, going back to the influential work of the Dennings [Den76, DD77, Den82]. They were the first to systematically study *secure information flow* in programs. Another novelty of their work is that it was the first to consider a secure information flow policy in terms of a mapping from variables into a lattice, where such a lattice defines the different security levels and how these levels relate to each other.

The notion of *non-interference* was first introduced by Goguen and Meseguer [GM82]. They define non-interference for automata by treating them as black-box processes and looking at the input and output of the automaton. The input and output channels are labeled with security levels, typically using to levels High for secret and Low for public information. By fixing the low input channels and varying (all) the high input channels, one can check if high input channels are indeed independent of low outputs. Figure 2.2 below illustrates non-interference in graphical form. This idea of confidentiality as non-interference turned out to be very natural when studying confidentiality for programs written in particular programming languages. This research on language-based secure information flow received a new impulse by the influential work of Volpano and Smith [VS97a]. They were the first who showed that type systems are especially well suited for (automatically) enforcing non-interference properties. And while Volpano and Smith only considered a small imperative programming language (similar to WHILE), others extended their main idea to (subsets of) languages like (sequential) Java [Mye99, BN05, Str03], Java bytecode [BRB05, Aga00a] and ML [PS03]. Using type-systems to enforce secure information flow proved to be highly successful. The combination of soundness and automation of the type checking algorithm is an obvious advantage. There are, of course, also disadvantages, most notably that a large group of secure programs are rejected by security type checking

algorithms. Below we show an example code fragment that does not leak any secret information, but is deemed insecure by type-based approaches to secure information flow. Here variable `low` has a low security level and `high` has a high security level.

Java

```
low = high ; low = 5;
```

This program fragment does not leak information (assuming that we work in a sequential setting), since after executing the (whole) fragment variable `low` will always have the same value (`5`) and can thus not leak information about the `high` variable. Type-based approaches will typically mark this code as leaking information. Indeed, the assignment of `high` to variable `low` *does* leak information, but this is undone by the subsequent assignment of a constant to variable `low`. Such *temporary breaches of confidentiality* are not detected by security type-checking algorithms.

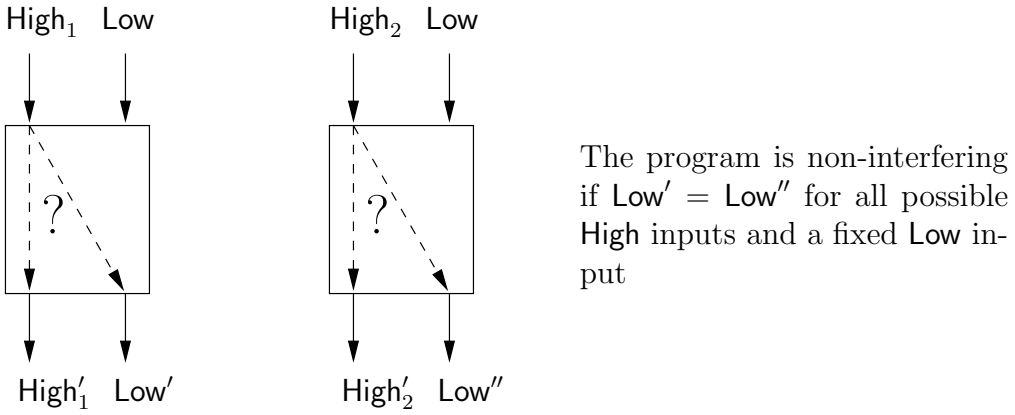

Figure 2.2: Non-interference.

Approaches based on other formalisms than type-checking have tried to deal with this problem. Most notable are approaches based on abstract interpretation [Cou96, CC77] that are still automatic and sound and exhibit fewer false positives than traditional type-based approaches [GM04, Zan02]. We will discuss a new approach for (automatically) checking non-interference that is based on abstract interpretation in detail in Chapter 6.

Another possibility is to give up automation of the analysis of non-interference and instead focus on approaches that are both sound and complete. The influential paper by Joshi and Leino [JL00] was one of the first to do this. Others have used theorem proving [DHS05], program logics [CHH02, ABB06] and partial equivalence relations [SS01] to interactively prove non-interference. In Chapter 7 we introduce a new program logic,

implemented in the theorem prover PVS [ORSH95] that can be used for proving non-interference properties of sequential Java programs. It is a Hoare logic based on relations instead of on predicates.

Most approaches proposed in the literature formalize confidentiality as *termination-insensitive* non-interference. This is the weak form of non-interference discussed before that only considers normal termination modes. We also consider this form of non-interference in Chapter 5 and Chapter 6.

A stronger version of non-interference does take the termination behavior of a program into account, which results in the notion of *termination-sensitive* non-interference. Thus, in this case non-interference ensures that no matter how a program terminates (or hangs) high variables are independent of low ones. In other words, how the program terminates is independent of high variables. This form of non-interference is studied in detail in Chapter 7.

Still stronger versions of non-interference also take *covert channels* [Lam73] under consideration. If a program leaks secret information via channels that are not intended for communication, we speak of information leakage via covert channels[14]. Resource consumption (memory/CPU/caching etc.) and timing behavior of a program can be used as a covert channel. In Chapter 8 we study *time-sensitive termination-sensitive non-interference* which considers –besides termination sensitive non-interference– information leakage via a timing channel.

Terminology

We will often use the words non-interference and confidentiality as synonymous. Though –as should be clear from the discussion above– technically not entirely correct, this is common in the literature. Other terms that are used in the literature for non-interference are *secure information flow* and *confinement*.

We prefer the term non-interference, since this is the most technical and least culturally loaded term. Both (a form of) confidentiality and (a form of) integrity can be expressed in terms of non-interference. We will focus on confidentiality, but all analysis methods proposed can be equally applied to the form of integrity that is the dual of confidentiality, see Section 5.1. 'Secure information flow' and 'confinement' are typical terms that are used in the program languages community. In this thesis all terminology will be interpreted as non-interference unless explicitly noted otherwise.

2.5.1 Security policies and security lattices

In order to be able to verify that a program is non-interfering, all output and input channels need to be labeled with appropriate security levels. This labeling is called a *secure information flow policy*. Instead of just labeling the inputs and outputs of a program we

[14] Notice that the termination behavior of a program can also be regarded as a covert channel.

will typically label all global variables (fields) and discard those that are only used locally afterwards[15].

Security levels need to be ordered ($\sqsubseteq$ below) in some way. As is standard in the literature [Den76], we will use a lattice for this ordering. The basic lattice one can use is the so-called *simple security lattice*. This lattice will suffice to explain the main ideas in most cases. It is defined in Definition 2 below.

Definition 2 (Simple security lattice)**.** The simple security lattice Σ is defined as:

$$\Sigma \;=\; \{\mathsf{High}, \mathsf{Low}\} \qquad \text{with } \mathsf{Low} \sqsubseteq \mathsf{High}$$

.

For presentation purposes we will only use the simple security lattice in examples in this thesis. In Chapter 6 we show how a more complicated lattice can be used for modeling the security levels. In practice, more complicated secure information flow policies will be needed, but all analysis methods presented in this thesis can be extended –at the cost of some more 'book-keeping'– to such complex lattices.

2.5.2 Downgrading

Non-interference –even the termination insensitive form– is a very strong (security) property that is typically too strict in practice. Numerous examples can be found of this:

- *verifying a PIN code* leaks (partial) information, since a device typically reveals if a PIN is incorrect, i.e., by not granting access. In practice this does not form a security risk since only a small number of false authentication attempts are allowed (typically 3), which makes the chance of guessing the correct PIN very low.
- *encrypted data* will also leak (partial) information, because an encrypted message depends on a (secret) encryption key. In practice a (secure) encryption scheme will (with a very high probability) not reveal the secret key, but the encrypted message is still not non-interfering.

A number of *downgrading policies* have been proposed in the literature [ZM01, CC04, CM04, LZ05, MSZar] that address this problem. Downgrading policies are extensions of a normal secure information flow policy that express how certain (secret or secret-depending) information can be released safely. Some additional mechanisms will have to ensure that only the correct, i.e., named in the downgrading policy, information is released. Safely declassifying/downgrading information is a research subject in itself that lies outside the scope of this thesis. It suffices to remark that while non-interference itself is usually too strong a notion to enforce in real world applications, together with a suitable downgrading policy, non-interference has indeed a more practical value.

[15] *Local* (stack) variables are not labeled by a secure information flow policy. These obtain the same security level as the security level of the expression that is assigned to it.

2.6 Tools

In this section we briefly discuss the main tools that have been used in the research covered by this thesis.

2.6.1 ESC/Java2

The Extended Static Checker for Java version 2 (ESC/Java2 [CK04, ESC]) is the open source successor of ESC/Java [FLL+02]. It is mainly developed by Joe Kiniry and David Cok. It is a tool that can be used to check JML specifications automatically. JML-annotated Java programs are checked using program verification of the program code and its formal annotations. While the tool is neither sound or complete, it finds a lot of errors, especially runtime exceptions such as null-pointer and array-out-of-bounds exceptions.

When ESC/Java2 checks an annotated program, it returns one of three results. The result "passed" indicates that ESC/Java2 believes the implementation of a method fulfills its specification; in that case we will say ESC/Java2 accepts the input. A result of "warning" indicates that an error potentially exists in the program. Typically examples include run-time violations like a `NullPointerException` or an `ArrayIndexOutOfBoundsException`. The result "error" indicates that something else went wrong, e.g., a syntactic error in the source file or the absence of a Java library, etc.

Users can control the amount and kinds of checking that ESC/Java2 performs by annotating their programs with specially formatted comments called pragmas. In its default mode, ESC/Java2 will try to find null dereferences or possible division by zero etc. Pragmas such as `nowarn Null` or `nowarn ZeroDiv` are used to suppress such warnings. Notice that pragmas are not part of JML (and can thus not be used by other JML-tools).

Verification conditions in ESC/Java2 are passed on to the back-end (automatic) theorem prover, Simplify [DNS05]. Simplify is no longer maintained. Currently, work is underway to integrate ESC/Java2 with other theorem provers, both interactive and automatic ones.

In this thesis ESC/Java2 is mainly used in Chapter 3. In other places we will sometimes briefly comment when we think extensions to ESC/Java2 might be useful. The tool is available for download from its website[16].

2.6.2 The LOOP verification framework

The LOOP (for Logic of Object-Oriented Programming) project [JP04] is a research project that aims at (interactive) verification of JML annotations for Java programs. In the LOOP framework, a Java source file annotated with JML assertions is translated automatically (using the LOOP tool) into a theory inside the theorem prover PVS [PVS]. This generated PVS theory contains the semantic representation of the Java program and translated JML annotations in the form of proof obligations. A semantic prelude defines the semantics

[16] http://secure.ucd.ie/products/opensource/ESCJava2/

of Java and JML in PVS. Correctness proofs are not generated automatically. Instead, interactive theorem proving in PVS is required to obtain these. However, a number of dedicated proof-strategies are available that automate this process as much as possible. Figure 2.3 summarizes the LOOP verification framework.

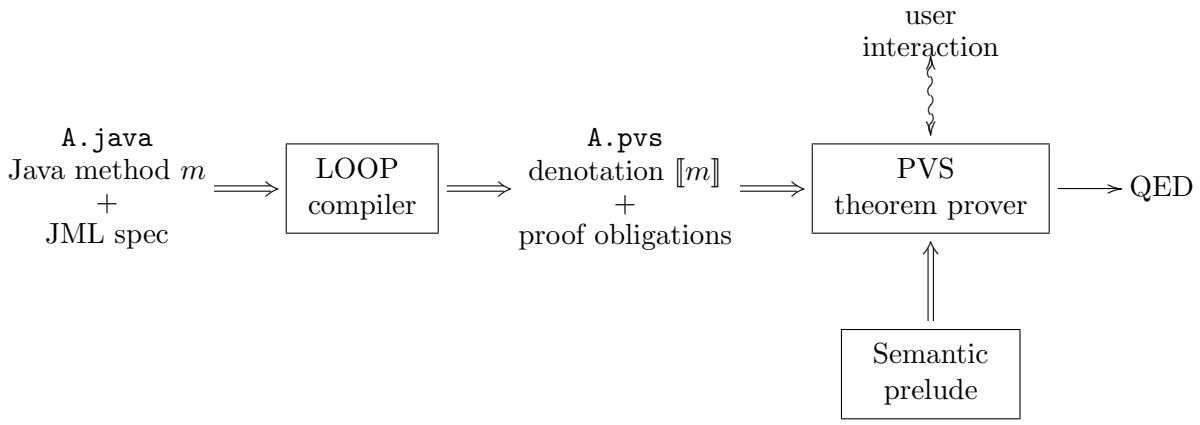

Figure 2.3: The LOOP verification framework.

One of the main aims of the LOOP project is to reason about a *real* programming language. Thus the semantical prelude contains a semantics in PVS of the whole sequential part of Java. Almost all Java features, such as aliasing, inheritance, abrupt termination via exceptions or labeled and unlabeled breaks and continues etc. are defined here. The only notable exceptions are inner classes and floating point data types.

In this thesis, the LOOP verification framework is used in Chapter 3 and Chapter 4. In Chapter 6 a new approach for statically checking non-interference is formalized on top of (a subset of) LOOP's Java semantics, and in Chapter 7 we present a new Hoare logic on relations that has been integrated in the LOOP verification project. In Section 7.3.1 we also give a small overview of LOOP's Java semantics. We refer to [JP04] for a more extensive overview of the LOOP project.

2.6.3 PVS

PVS is not only a theorem prover, it is a complete verification system: that is, a specification language integrated with support tools and a theorem prover. The specification language of PVS is typed higher-order logic wich supports dependent types. The Java and JML semantics from the LOOP project are formalized in this language. The actual theorem prover[17] supports a range of powerful (primitive) inference procedures. These can

[17]We prefer the term 'proof assistant', since PVS (usually) does not prove theorems. It assists the user and handles the easier cases automatically so that the user can concentrate on the harder (parts of a) proof. However, since the name 'theorem prover' seems to the be standard in the literature, we will stick with it.

be combined into even more powerful *strategies*[18], written in Lisp [Gra95]. PVS has been used on numerous case studies, e.g., to specify, design and verify safety-critical flight control systems, including a software requirements specification for the Space Shuttle [CDV96].

PVS is used extensively throughout this thesis. In Chapter 3, Chapter 4 and Chapter 5 we use PVS as a back-end to the LOOP tool to prove the correctness of the JML specifications. The proposed frameworks for proving non-interference in Chapter 6 and Chapter 7 have also been completely formalized in PVS. We have chosen to present our work as general as possible and will thus not go into all the details associated with the verification and formalizations in PVS. However, a lot of time and effort have been spent on this work in PVS and it should ensure a higher level of confidence in the techniques presented in this thesis.

We refer to the PVS manuals [OSRSC99, SORSC99] and website[19] for more information (including download and install details) on PVS. Also see [GH98] for an introduction to PVS (and a comparison to the theorem prover Isabelle).

2.6.4 Other tools

Numerous other tools have been used in this thesis. We briefly discuss these here.

The Iowa State JML tools [JML] form a tool-suite which include a type checker, a runtime assertion checker [CL02a] and a unit-test generator [CL02b]. The JML type-checker has been used extensively during the JML specification of the Java programs in Chapter 3 through 5.

Findbugs [Fin] is a tool that analyzes Java programs and warns for indications of bugs. It is based on heuristics and can be easily extended. Possible bug-pattern indicators are things like `equals` methods that do *not* override the `equals` method from `java.lang.Object` or mutably dependent static fields. In Chapter 3 we apply Findbugs to some examples.

JIF [Jif] is a tool for checking non-interference in Java programs. It is based on security types and covers the whole of (sequential) Java. As far as we know, there is no soundness result for the type-checking algorithm used by JIF.

Flow Caml [Flo] is a tool that can check non-interference for the ML dialect Caml. The tool is also based on security types and has a (provably) sound type-checking algorithm.

We have not used the last two tools, they are mentioned here because we aim to have at least the same level of coverage that these tools have. In Chapter 7 we introduce a framework for (interactively) checking non-interference properties of (full) sequential Java. This approach relates to the last two tools in the same way as the LOOP verification framework relates to ESC/Java2.

[18] Strategies are called 'tactics' in the theorem provers Coq [Coq] and Isabelle [Pau94].
[19] http://pvs.csl.sri.com/

Chapter 3

Specification and verification of Java programs

Contemporary software projects typically have specifications given in informal natural language, possibly in combination with some semi-formal diagrams (e.g., in UML [Fow00]). While this in itself is not a problem for most software projects, it can become a problem when one wants to certify the quality of the software.

Software certification mechanisms, such as the Common Criteria[20] (CC), typically require the use of formal methods for the highest assurance levels (CC level EAL7).

Using a formal language to write specifications then has two obvious advantages: (i) the specification is unambiguously clear (in a mathematical sense) and (ii) the correctness of the specification, with respect to the program it specifies, can formally be proved (verified).

Both writing a useful specification and proving the correctness of this specification with respect to the program are non-trivial tasks. If specifications are too high level (e.g., "the program should be secure"), they are not useful at all; the same holds for specifications that contain too much detail (e.g., a program forms a perfect –if trivial– specification of itself).

Moreover, in the study of (sequential) program verification one usually encounters the same examples over and over again (e.g., stacks, lists, alternating bit protocol, sorting functions, etc.), often going back to classic texts like [Hoa69, Dij76, Gri81]. These examples typically use an abstract programming language with only a few constructs, and the logic for expressing the program properties (or specifications) is some variation on first order logic. While these abstract formalisms were useful at the time for explaining the main ideas in the field of programming verification, they are not so helpful anymore when it comes to actual program verification for modern programming and specification languages.

In this chapter we give an updated series of program verification examples/challenges written in Java. The programs are in no way 'real world' applications, though some of the examples below are inspired by such programs. They form a set of semantically challenging program fragments. They should illustrate both the difficulties in writing clear behavioral

[20] http://www.commoncriteriaportal.org/

specifications and give an idea of the semantic complexity involved in formalizing a real world language like (sequential) Java. The chapter is based on [JKW03]. We use the Java specification language JML [LBR99b], see Section 2.3 above, to specify our examples.

The examples presented below are based on our experience with the LOOP tool [BJ01, JP04] over the past years. Although the examples have actually been verified, the particular verification technology based on the LOOP tool does not play an important rôle: we shall focus on the underlying semantical issues. The examples may in principle also be verified via another approach, like those of Jack [BLR02], Jive [MPH00], Krakatoa [CDF+04], and KeY [ABB+04]. All examples have been verified with the LOOP tool and ESC/Java2. We indicate where verification with LOOP or with the (automatic) ESC/Java2 [CK04] tool brings up interesting semantical issues in our examples[21].

The examples below do not cover all possible topics. Most noticeably, floating point numeric types and multi-threading are not covered. In general, our work has focused on Java for smart cards and several examples stem from that area. Other examples are taken from work of colleagues [BL99], from earlier publications, or from test sets that we have been using internally. They cover a reasonable part of the semantics of Java and JML. Most of the examples can easily be translated in other programming languages. The examples do not involve semantical controversies, as discussed for instance in [BS99].

The original idea behind the paper [JKW03] –which this chapter is based on– was to collect a number of challenging examples which could be used as a test set for anybody working in the program verification field, especially for those working on (sequential) Java. We do not claim that the examples presented here are meant to be canonical or proscriptive, however the examples should give a flavor of the level of completeness (and thus, complexity) necessary to cover modern programming language semantics.

We have classified the verification examples in the following categories:

- Side-effects
- Data types
- Control flow
- Inheritance
- Static initialization

Some of the examples below are not restricted to one category. Since the more interesting examples often combine multiple features; we take the liberty to just classify them where we think is most appropriate. Finally, our explanations focus on the (semantic) issues involved, and not so much on the actual code snippets. They should be relatively self-explanatory.

[21] We use the (at the time of writing) latest alpha binary release of ESC/Java2 (ESC/Java-2.0a9, 1 October 2005).

3.1 Side-effects

One of the most common abstractions in program verification is to omit side-effects from expressions in the programming language. This is a serious restriction. Below we show a nice and simple example from [BL99] where such side-effects play a crucial part, in combination with the logical operators. Recall that in Java there are two disjunctions (`|` and `||`) and two conjunctions (`&` and `&&`). The double versions (`||` and `&&`) are the so-called conditional operators: their second argument is only evaluated if the first argument is false (for `||`) or true (for `&&`).

Java

```
Class Logic{

        boolean b = true;
        boolean result1, result2;

        /*@ normal_behavior
          @    requires true;
          @  assignable b;
          @     ensures b == !\old(b) && \result == b;
          @*/
        boolean f(){ b = !b; return b; }

        /*@ normal_behavior
          @    requires true;
          @  assignable b, result1, result2;
          @     ensures (\old(b) ==> !result1 && result2) &&
          @             (!\old(b) ==> result1 && result2);
          @*/
        void m(){
           result1 = f() || !f();
           result2 = !f() && f();
         }
}
```

When the field `b` above is true, the expression `f() || !f()` evaluates to false while `!f() && f()` evaluates to true, going against standard logical rules.

LOOP may either use the implementation[22] or the specification of `f()` in the verification of the specification for method `m()`. The assertion itself is verified without any difficulty in either case and can be handled completely without user interaction.

[22] As an aside, it is generally not desirable to use the implementation of a method instead of its specification when verifying programs, since reasoning with implementations is not modular. However, sometimes this is unavoidable (at least using JML's behavioral subtyping semantics, see section 3.3.4).

ESC/Java2 uses the specification of method `f()` in the verification of method `m()`. The tool has no problems with verifying this example: it says "pass".

3.2 Data types

Another common abstraction in program verification is to consider only two data types: the booleans and the (infinite) integers. In reality modern programming languages contain data types such as references, arrays and bounded numeric types.

3.2.1 Aliasing

Aliasing is the phenomena where two (or more) names refer to the same object. An example is shown below.

Java

```
class C {
    C a;
    int i;

    /*@ normal_behavior
      @    requires true;
      @  assignable a, i;
      @     ensures a == null && i == 1;
      @*/
    C() { a = null; i = 1; }
}

class Alias {
    /*@ normal_behavior
      @    requires true;
      @  assignable \nothing;
      @     ensures \result == 4;
      @*/
    int m() {
        C c = new C();
        c.a = c;
        c.i = 2;
        return c.i + c.a.i;
    }
}
```

This second example might seem trivial to some readers. The return expression of the method `m()` references the value of the field `i` of the object `c` value via an aliased reference to itself, i.e., `c`, in the field `a`. We present this example because it represents (in our view) the bare minimum necessary to model a language with pointers (like Java).

LOOP verifies this example automatically.

ESC/Java2 has no problem verifying this program.

3.2.2 Overflow of numeric types

The next example (due to Cees-Bart Breunesse) shows a program that does not terminate:

Java

```
class Diverges{

    /*@ behavior
      @    requires true;
      @  assignable \nothing;
      @     ensures false;
      @     signals (Exception e) false;
      @    diverges true;
      @*/
    public void m(){
        for (byte b = Byte.MIN_VALUE; b <= Byte.MAX_VALUE; b++)
            ;
    }
}
```

In the specification above the JML keyword `diverges` followed by the predicate `true` indicates that the program may fail to terminate. Since the specification further asserts that the program cannot terminate normally or with an exception the specifications as a whole indicates that the program fails to terminate.

The non-termination of the program above can easily be explained once one realizes that overflow in Java is silent, because `Byte.MAX_VALUE + 1` evaluates to `Byte.MIN_VALUE`, i.e., 127 + 1 = -128, and thus the guard in the for loop will never fail. Note that in order to verify this program, both overflow and non-termination must be modeled appropriately.

LOOP models both overflowing and non-termination. The example is verified without any problems,though user interaction is required.

ESC/Java2 cannot prove the correctness of methods that do not terminate. Moreover, overflow of numerical data types is not modeled inside ESC/Java2's logic. The above specification will actually pass, but this result is meaningless since if we change the diverges

clause to false and leave the rest of the specification the same, ESC/Java2 will also pass the example. Other incorrect specifications for this method give similar (incorrect) results.

3.2.3 Bitwise operations

Our next example is not of the sort one finds in textbooks on program verification. Yet, it is a good example of the realistic code that verification tools have to deal with in practice, specifically in Java Card applets, for selecting relevant parts of incoming APDU's.

Java

```
static final byte ACTION_ONE = 1, ACTION_TWO = 2,
                  ACTION_THREE = 3, ACTION_FOUR = 4;


private /*@ spec_public @*/ byte state;


/*@ behavior
  @    requires true;
  @  assignable state;
  @     ensures (cmd == 0 && state == ACTION_ONE)   ||
  @             (cmd == 16 && state == ACTION_TWO)  ||
  @             (cmd == 4 && state == ACTION_THREE) ||
  @             (cmd == 20 && state == ACTION_FOUR);
  @     signals (Exception)
  @             ((cmd & 0x07) != 0 || (cmd != 0 && cmd != 16))
  @             &&
  @             ((cmd & 0x07) != 4 || (cmd != 4 && cmd != 20));
  @*/
void selectstate(byte cmd) throws Exception {
    byte cmd1 = (byte)(cmd & 0x07), cmd2 = (byte)(cmd >> 3);
    switch (cmd1) {
    case 0x00: switch (cmd2) {
               case 0x00: state = ACTION_ONE; break;
               case 0x02: state = ACTION_TWO; break;
               default: throw new Exception(); }
               break;
    case 0x04: switch (cmd2) {
               case 0x00: state = ACTION_THREE; break;
               case 0x02: state = ACTION_FOUR; break;
               default: throw new Exception(); }
               break;

    default: throw new Exception(); }
```

```
    // ... more code
}
```

It involves a 'command' byte `cmd` which is split in two parts: the rightmost three, and leftmost five bits. Depending on these parts, a dedicated `state` field is given an appropriate value (also see Chapter 4 for more information about using such a `state` field). This happens in a nested switch. The specification is helpful because it tells in decimal notation what is going on, although the `signals` clause is still not very readable.

LOOP verifies the example using the bitvector semantics described in [Jac03].

ESC/Java2 has no semantics for bitwise operators like signed right shift (`>>`) or bitwise and (`&`) thus cannot establish the postcondition.

3.2.4 Numeric types in specification and implementation

Representations of numeric types (e.g., `int`, `short`, `byte`, etc.) in Java are finite. A review of annotated programs that use integral types indicates that specifications are often written with infinite numeric models in mind [Cha02]. Programmers seem to be aware of the issues of overflow (and underflow, in the case of floating point numbers) in program code, but not in specifications.

Additionally, it is often the case that specifications use functional method invocations. Such methods, which have no side-effects, are referred to as "pure" methods in JML[23]. Method `abs(int a)` below is such a pure method.

Java

```
package java.lang.Math;

/*@ normal_behavior
  @    requires true;
  @  assignable \nothing;
  @     ensures \result == ((a >= 0
  @                 || a == Integer.MIN_VALUE) ? a : -a);
  @*/
public static /*@ pure @*/ int abs(int a)
```

The example `isqrt(int x)` method below highlights these complications, as it uses method invocations in a specification and integral values that can potentially overflow. The method `isqrt()`, which computes an integer square root of its input, is inspired by

[23] There is still debate in the community about the meaning of "pure". Many Java methods, for example, which claim to have no side-effects, and thus should be pure, actually do modify the state due to caching, lazy evaluation, etc. See e.g., [BNSS05, DM05].

a specification (see method `isqrt2(int x)` below) of a similar function included in the documentation of an early JML releases [LBR99a].

Note that the `abs()` method is used in the specification of `isqrt()` to emphasize that both the negative and the positive square root are an acceptable return value, as all we care about is its magnitude. Actually, the given implementation of `isqrt()` only computes the positive integer square root.

Java

```
/*@ normal_behavior
  @    requires x >= 0 && x <= 2147390966;
  @  assignable \nothing;
  @     ensures java.lang.Math.abs(\result) < 46340 &&
  @             \result * \result <= x &&
  @             x < (java.lang.Math.abs(\result) + 1) *
  @                 (java.lang.Math.abs(\result) + 1);
  @*/
int isqrt(int x) {
    int count = 0, sum = 1;
     /*@ maintaining 0 <= count &&
       @             count < 46340 &&
       @             count * count <= x &&
       @             sum == (count + 1) * (count + 1);
       @  decreasing x - count;
       @*/
    while (sum <= x) { count++; sum += 2 * count + 1; }
    return count;
}
```

The semantics of JML states that expressions in the requires and ensures clauses are to be interpreted in the semantics of Java. Consequently, a valid implementation of the specification of method `isqrt2(int x)` is permitted to return `Integer.MIN_VALUE` when `x` is 0, indeed in Java $(\texttt{Integer.MIN\_VALUE})^2 = 0$ (as already noted in [LBR99a]).

This surprising situation exists because Java's numeric types are bounded and because the definition of absolute value in the Java Language Specification is somewhat unexpected: `abs(Integer.MIN_VALUE) == Integer.MIN_VALUE`. While this is the documented behavior of `java.lang.Math.abs(int)`, it is often overlooked by programmers because they presume that a function as mathematically uncomplicated as absolute value will produce unsurprising, mathematically correct results for all input. The absolute value of `Integer.MIN_VALUE` is equal to itself because `Math.abs()` is implemented with Java's unary negation operator '`-`'. This operator silently overflows when applied to the maximal negative integer or long [GJSB00, §15.15.4].

Java

```
/*@ normal_behavior
  @   requires x >= 0;
  @    ensures java.lang.Math.abs(\result) <= x &&
  @            \result * \result <= x &&
  @            x < (java.lang.Math.abs(\result) + 1) *
  @                (java.lang.Math.abs(\result) + 1);
  @*/
int isqrt2(int x)
```

The precondition of method `isqrt(int x)` is explained in Figure 3.1. We wish to ensure that no operation, either in the implementation of `isqrt()` or in its specification, overflows. The critical situation that causes an overflow is when we attempt to take an integer square root of a very large number. In particular, if we attempt to evaluate the postcondition of `isqrt()` for values of `x` larger than 2,147,390,966 an overflow takes place. The small but critical interval between the precondition's bound 2,147,390,966 and `Integer.MAX_VALUE` = 2,147,483,647 ($= 2^{31} - 1$) is indicated by the dark interval on the right of Figure 3.1: to check the postcondition, the prospective root (the arrow labeled 1) must be determined, one is added to its value (arrow 2), and the result is squared (arrow 3). This final result will thus overflow. Indeed, $(46,340 + 1)^2 > 2,147,483,647$ $(= 2^{31} - 1)$.

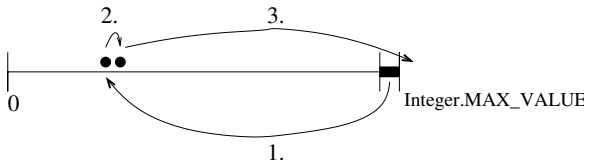

Figure 3.1: The positive integers.

The erroneous nature of a specification involving potential overflows should become clear when one verifies the method using an appropriate bit-level representation of integral types [Jac03]. Unfortunately, such errors are not at all apparent, even when performing extensive unit testing, because the boundary conditions for arithmetic expressions, like the third term of the postcondition of `isqrt(int x)`, are rarely automatically derivable, and full state-space coverage is simply too computationally expensive.

LOOP proves the specification of method `isqrt(int x)`, but since the support for the use of pure methods in specifications is still rudimentary, this takes some considerable effort.

ESC/Java2 cannot prove the correctness of the specification of method `isqrt(int x)` since it does not handle overflow of Java's numeric types.

Specifications involving integral types, and thus potential overflows, are frequently seen in application domains that involve numeric computation, both complex (e.g., scientific

computation, computer graphics, embedded devices, etc.) and relatively simple (e.g., currency and banking). The former category are obviously challenging due to the complexity of the related data-structures, algorithms, and their specifications, and the latter are problematic because it is there that implementation violations can have drastic (financial) consequences. This specification raises the question: What is the appropriate model for arithmetic specifications? Recently, both Chalin [Cha03, Cha04] and Breunesse [Bre06, Chapter 4] proposed extensions to JML that address this issue.

3.3 Control flow

Java has some interesting control flow features ranging from abrupt termination, exceptions and breaks, to method calls. Each of these have their own semantic difficulties.

3.3.1 Return inside try-catch-finally

Typical of Java is its systematic use of exceptions[24], via its statements for throwing and catching [Kin06].

Java

```
int m;

/*@ normal_behavior
  @    requires true;
  @  assignable m;
  @     ensures \result == ((d == 0) ? \old(m) : \old(m) / d)
  @                && m == \old(m) + 10;
  @*/
int returnfinally(int d){
    try { return m / d; }
    catch(Exception e){ return m / (d+1); }
    finally{ m += 10; }
}
```

They require a suitable control flow semantics. Special care is needed for the `finally` part of a try-catch-finally construction. Semantically, unless the `try` or `catch` parts do not terminate at all, the `finally` part is always executed. The simple example above (adapted from [Jac01]) combines many aspects. The subtle point is that the assignment `m += 10` in the finally block will still be executed, despite the earlier return statements, but has no effect on the value that is returned. The reason is that this value is bound earlier.

[24] At least in Sun's standard APIs.

LOOP has no problems with this example.

ESC/Java2 initially fails on this example and warns for a possible division by zero. Only after we add the annotation `//@ nowarn ZeroDiv;` the program passes.

3.3.2 Throwing exceptions

The example in this section illustrates how specifications can become complicated for even (very) small programs. The question that inspired this example is if it is possible to throw an exception in Java that is a null reference. It turns out that this is not possible, because before the null reference can be thrown a `NullPointerException` is raised. A correct semantics of Java should reflect this behavior.

Java

```
class Small{

 Exception ex;

    /*@ exceptional_behavior
      @    requires true;
      @  assignable \nothing;
      @     signals (Exception e) ((ex == null) ==>
      @               \typeof(e) == \type(NullPointerException))
      @                          && ((ex != null) ==> e == ex);
      @     signals (NullPointerException ne) ex == null ||
      @                       ex instanceof NullPointerException;
      @*/
    public void throwIt() throws Exception{
        throw ex;
    }
}
```

The specification is rather subtle. We know that if an exception is thrown and the `ex` is null, then the thrown exception has dynamic type `NullPointerException`. The JML keyword `\typeof` is used to refer to the *most-specific dynamic type of an expression's value* [LPC+05, §11.4.12], while the `\type` keyword is used in order to interpret a specific class as a type, i.e., `\type(A)` is "class `A` as type" (also see Section 3.4 and [LPC+05, §11.4.14]). In case `ex` is not null, we only know that an exception is thrown (and have no idea about its runtime type).

In case a `NullPointerException` is thrown we know that either `ex` is `null`, or `ex` has runtime type `NullPointerException` or the runtime type of one of `NullPointerException`'s subclasses. Notice that the `signals` clause with the `NullPointerException` is a refinement of the general `signals` clause regarding the top-level `Exception` type.

This example clearly shows that writing 'complete' specifications in JML is not always feasible and is indeed unwanted. Writing clear specifications at the right level of abstraction is probably harder then writing a correct program.

LOOP has no problem verifying the correctness of the specification.

ESC/Java2 also has no problems verifying this example if the pragma `//@ nowarn Null` is added. ESC/Java2 typically warns the user when unwanted behavior (such as division by zero or null dereferencing) can occur.

3.3.3 Breaking out of a loop

While and for loops are typically used for going through an enumeration, for instance to find or modify an entry meeting a specific condition. Upon hitting this entry, the loop may be aborted via a break statement. This presents a challenge for the underlying control flow semantics.

Java

```
int[] ia;

/*@ normal_behavior
  @    requires ia != null;
  @  assignable ia[*];
  @     ensures \forall int i; 0 <= i && i < ia.length ==>
  @         ((\old(ia[i]) < 0 &&
  @         (// i is the first position with negative value
  @          \forall int j; 0 <= j && j < i ==>
  @                  \old(ia[j]) >= 0))
  @         ? ia[i] == -\old(ia[i])
  @         : ia[i] == \old(ia[i]));
  @*/
void negatefirst() {
    /*@ maintaining 0 <= i && i <= ia.length &&
      @             (\forall int j; 0 <= j && j < i ==>
      @               (ia[j] >= 0 ));
      @  decreasing ia.length - i;
      @*/
    for(int i = 0; i < ia.length; i++) {
        if (ia[i] < 0) { ia[i] = -ia[i]; break; } }
}
```

Above a simple example is shown of a method with a `for` loop that goes through an array of integers in order to change the sign of the first negative entry. The two

lines of Java code are annotated with the loop invariant, with JML-keyword `maintaining` asserting what holds while going through the loop, and the loop variant, with JML-keyword `decreasing`. The loop variant is a mapping to the natural numbers which decreases with every loop cycle that must reach zero on loop termination. It is used in verifications to show that the repetition terminates.

LOOP verifies the example without any problems.

ESC/Java2's result on this program is not very interesting because of its limited handling of loops: they are executed (symbolically) only once by default. In general, this may indicate a basic problem with the invariant, but the coverage is far from complete[25].

As an aside: ESC/Java2 has difficulty with this example due to limitations in its parser as quantified expressions cannot be used inside conditional expressions (`? :`). If we rewrite the specification of method `negatefirst()` as a conjuction of disjoint implications, ESC/Java2 accepts the program.

3.3.4 Class invariants and callbacks

Class invariants are extremely useful in specification because they often make explicit what programmers have in the back of their mind while writing their code. A typical example is: "integer `i` is always non-zero" (so that one can safely divide by `i`).

The naive[26] semantics for class invariants is: an invariant should hold after construction, and when it holds in the pre-state of a (non-constructor) method, it must also hold in the post-state. Note that this post-state can result from either normal or exceptional termination. An invariant may thus be temporarily broken within a method body, as long as it is re-established at the end. A simple example is method `decrementk()` below.

Things become more complicated when, inside such a method body, the class invariant is broken and another method is called. The current object `this` is then left in an inconsistent state. This is especially problematic if control returns at some later stage to the current object. This re-entrance or callback phenomenon is discussed in [Szy98, Sections 5.4 and 5.5]. The commonly adopted solution to this problem, which is also defined in the semantics of JML, is to require that the caller establishes the invariant of `this` before a method call. Hence the proof obligation in a method call `a.m()` involves the invariants of both the caller (`this`) and the callee (`a`).

LOOP can *not* prove the specification for the method `incrementk()` since it uses the naive semantics (with respect to callbacks). However, a proof using the implementations of method `go(A arg)` and `decrementk()` is possible, if we make the additional assumptions that the run-time type of the field `b` is actually `B`, and that the method `incrementk()` is executed on an object of class `A`. These restrictions are needed because if, for instance, field

[25] We can use the (undocumented) `-LoopSafe` option, which proves the termination of a loop, given a suitable loop invariant and loop variant. In that case ESC/Java2 proves the correctness of the specification.

[26] We use the naive semantics here, since this is the semantics incorporated in the LOOP framework. Besides, for this particular example, the naive semantics coincides with the standard semantics which states that the invariant holds in all *visible* states.

`b` has a subclass of `B` as run-time type, a different implementation will have to be used if the method `go(A arg)` is overridden in the subclass (see Section 3.4.2 for another example of this).

ESC/Java2 warns about the potential for invariant violation during the callback.

Java

```
class A {
    private /*@ spec_public @*/ int k, m;
    B b;

    /*@ invariant k + m == 0; @*/

    /*@ normal_behavior
      @    requires true;
      @  assignable k, m;
      @     ensures k == \old(k) - 1 && m == \old(m) + 1;
      @*/
    void decrementk () { k--; m++; }

    /*@ normal_behavior
      @    requires b != null;
      @  assignable k, m;
      @     ensures k == \old(k) && m == \old(m);
      @*/
    void incrementk () { k++; b.go(this); m--; }
}

class B {
    /*@ normal_behavior
      @    requires arg != null;
      @  assignable arg.k, arg.m;
      @     ensures arg.k == \old(arg.k) - 1 &&
      @               arg.m == \old(arg.m) + 1;
      @*/
    void go(A arg) { arg.decrementk(); }
}
```

Another issue related to class invariants is whether or not they should be maintained by *private* methods. JML does require this, but supports a special category of so-called 'helper' methods that need not maintain invariants. We do not discuss this matter further.

3.4 Inheritance

Inheritance, together with other object-oriented features, is crucial in building modular software systems, but its semantics can often be tricky. This section describes some typical object-oriented features of Java such as inheritance, method overriding and late-binding.

3.4.1 Combining late- and early-binding

The next example is an adaption of an example by Kim Bruce[27]. In the example three different objects are created, and the question is which of the three `equals` methods will be called.

Notice that the `equals(Point x)` method in the `ColorPoint` class not only *overrides* the `equals(Point x)` method of the (super) class `Point`, it is also *overloads* the `equals` name in class `ColorPoint`. Overloading in Java is resolved at compile time (early binding), while overriding is resolved at run-time (late binding).

When this is kept in mind, most examples should be self-explanatory. As an example we explain why `p2.equals(cp)` (field `r8`) calls method `equals(Point x)` in class `ColorPoint` and *not* the `equals(ColorPoint x)` method in the same class. The static type of object `p2` is `Point`; during compiling the method call is thus bound to the `equals` method in class `Point` (early binding). Then, at run-time, the dynamic type of `p2` (a `ColorPoint`) decides via inheritance which method is called (late binding). This is of course the method in class `ColorPoint` (because of `p2`'s run-time type) and, since only method `equals(Point x)` is overridden, this is the one that is called. The other `equals` method in class `ColorPoint` is effectively already ignored because of early binding.

Java

```
class Point{


  /*@ normal_behavior
    @    requires \typeof(this) == \type(Point);
    @  assignable \nothing;
    @     ensures \result == 1;
    @*/
 int equals(Point x) { return 1; }
}
```

[27] Originally posted to the TYPES-mailing list. The original example only has one `equals` method in class `ColorPoint`, namely the one with a `ColorPoint` as argument. See http://www.seas.upenn.edu/~sweirich/types/archive/1997-98/msg00444.html

```
class ColorPoint extends Point {

  /*@ normal_behavior
    @    requires true;
    @  assignable \nothing;
    @     ensures \result == 2;
    @*/
 int equals(ColorPoint x){ return 2; }


  /*@ also
    @ normal_behavior
    @    requires \typeof(this) == \type(ColorPoint);
    @  assignable \nothing;
    @     ensures \result == 3;
    @*/
 int equals(Point x) { return 3; }
 }

class Override{

  int r1,r2,r3,r4,r5,r6,r7,r8,r9,r10;

  /*@ normal_behavior
    @    requires true;
    @  assignable r1, r2, r3, r4, r5, r6, r7, r8, r9;
    @     ensures r1 == 1 && r2 == 1 && r3 == 3 &&
    @             r4 == 3 && r5 == 3 && r6 == 3 &&
    @             r7 == 1 && r8 == 3 && r9 == 2 && r10 == 2;
    @*/
  void m() {
      Point p1 = new Point();
      Point p2 = new ColorPoint();
      ColorPoint cp = new ColorPoint();
      r1 = p1.equals(p1);r2 = p1.equals(p2);r3 = p2.equals(p1);
      r4 = p2.equals(p2);r5 = cp.equals(p1);r6 = cp.equals(p2);
      r7 = p1.equals(cp);r8 = p2.equals(cp);r9 = cp.equals(cp);
      r10 = cp.equals((ColorPoint) p2); }
 }
```

As an aside, this example also illustrates why one should, as a rule, override the `equals` method of `java.lang.Object`. Since if this `equals` is overridden it is always clear which `equals` is called (see e.g., [Hor05, §13.8.2]). Tools like Findbugs [Fin] can detect such

unwanted equals methods[28] automatically and warn against using them.

Again, the specification of `equals` methods uses the JML keywords `\typeof`, which refers to the dynamic type of an object, and `\type`, which gives the (class) type of a class. Notice that we use these JML keywords here to circumvent behavioral subtyping.

According to JML's behavioral subtyping semantics (see Section 2.3), the `equals` method in the subclass `ColorPoint` should also satisfy the specification of the `equals` method in superclass `Point` (as indicated by the `also` keyword in the specification). This makes writing the specification of the `equals` method in the class `ColorPoint` non-trivial.

LOOP does not allow the use of method specifications which contain the `also` keyword. To prove the specification of method `m()` we can either manually desugar the specification of `equals(Point x)` in `ColorPoint` or use the implementations of the `equals` methods.

ESC/Java2 warns that the postcondition is possibly not established. It turns out that ESC/Java2 uses the static type of an object to determine which method to call. Thus `r3`, `r4` and `r8` get different values assigned to them then ESC/Java2 expects. This is a serious error in ESC/Java2's semantics. We have contacted the developers and they are aware of the issue. The latest version of ESC/Java2 –version ESCJava-2.0a9– still exhibits this behavior.

3.4.2 Inheritance and method overriding

The next program is from [HJ00a] and was originally suggested by Joachim van den Berg. On first inspection it looks like the method `test()` will loop forever.

Java

```
class C {

    /*@ behavior
      @    diverges \typeof(this) == \type(C);
      @  assignable \nothing;
      @     ensures false;
      @*/
    void m() throws Exception {
      m();
    }
}
```

[28] Such equals methods i.e., `equals(ColorPoint x)`, form an example of so-called 'covariant overloading'.

```
class Inheritance extends C {

    /*@ also
      @ exceptional_behavior
      @    requires \typeof(this) == \type(Inheritance);
      @  assignable \nothing;
      @     signals (Exception) true;
      @*/
    void m() throws Exception { throw new Exception(); }

    /*@ exceptional_behavior
      @    requires true;
      @  assignable \nothing;
      @     signals (Exception) true;
      @*/
    void test() throws Exception { super.m(); }
}
```

However, the method `test()` calls method `m()` from class `C`, which calls method `m()` in class `Inheritance`, since 'this' has runtime-type `Inheritance` and thus method `test()` will not loop. Due to the behavioral subtyping semantics used in JML for inheritance, we have to be careful writing the specifications for the `m()` methods. Again, the `also` keyword expresses that method `m()` from class `Inheritance` has to respect the specification of method `m()` of its super class `C`.

LOOP proved the correctness of method `test()` using the implementation of method `m()`. LOOP cannot handle specifications that contain the `also` keyword, though it is possible to (manually) desugar the specs and use the specification of method `m()` in class `Inheritance`.

ESC/Java2 accepts the specifications without any complaints. However, the specifications of both `m()` methods are in this case crucial for verifying method `test()` –as ESC/Java2 cannot reason on the basis of an implementation when no specification is present.

3.5 Static initialization

This section describes an example that deals with the static initialization of different classes which have mutually dependent static fields.

3.5.1 Mutually-dependent static fields

The code below shows an example of static initialization in Java (due to Jan Bergstra). In Java a class is initialized at its first *active* use (see [GJSB00, §12.4]). This means that class initialization in Java is lazy, so that the result of initialization depends on the order in which classes are initialized. This rather sick example shows what happens when two

classes, which are not yet initialized, have static fields referring to each other. In the specification we use the (non JML) keyword `\static_fields_of` in the assignable clause. It is syntactic sugar for all static fields of the class.

Java

```
class C {
  static boolean result1, result2, result3, result4;

    /*@ normal_behavior
      @    requires !\is_initialized(C)   &&
      @             !\is_initialized(C1)  &&
      @             !\is_initialized(C2);
      @  assignable \static_fields_of(C),
      @             \static_fields_of(C1),
      @             \static_fields_of(C2);
      @     ensures result1 && !result2 && result3 && result4;
      @*/
    static void m(){
        result1 = C1.b1; result2 = C2.b2;
        result3 = C1.d1; result4 = C2.d2;
    }
}

class C1 {
    static boolean b1 = C2.d2;
    static boolean d1 = true;
}


class C2 {
    static boolean d2 = true;
    static boolean b2 = C1.d1;
}
```

The first assignment in the body of method `m()` triggers the initialization of class `C1`, which in turn triggers the initialization of class `C2`. The result of the whole initialization is, for instance, that static field `C2.b2` gets value `false` assigned to it. This can be seen when one realizes that the boolean static fields from class `C1` initially get the default value `false`. Subsequently, class `C2` becomes initialized and its fields also get the default value `false`. Now the assignments in class `C2` are carried out: `d2` is set to `true` and `b2` is set to `false`. Note that `d1` is still `false` at this stage. Finally the assignments to fields in class `C1` take place, both resulting in value `true`.

One can see that the order of initializations is important. When the first two assignments in the method body of `m()` are swapped, class `C2` will be initialized before class `C1`

resulting in all fields getting value `true`.

As an aside, this example describes the semantics of static initialization in Java. It can be argued that the behavior described here is unwanted (see e.g., [KS02]) and indeed should be considered an error, because most programmers will not expect this kind of behavior. If one aims to have a complete semantics of Java, then static initialization should be modeled accordingly in a precise semantics. However, in general, mutual depended (static) fields should be considered a bug. There are tools (again Findbugs [Fin]) that can automatically detect such circularities and mark them as bugs.

LOOP proves the correctness of the specification.

ESC/Java2 cannot handle this example as its semantics does not include static initialization.

3.6 Conclusions

The main observation that inspired this chapter was that the classical examples in (sequential) program verification are no longer sufficient in today's context. They need to be updated in two dimensions: language complexity and size. This chapter focuses on complexity, by presenting a new series of semantically challenging examples, written in Java, with correctness assertions expressed in JML. The examples incorporate some of the ugly details that one encounters in real-world programs, and that any reasonable semantics should be able to handle[29].

As for the question how canonical the examples presented here are, we claim that the (sub)categories identified (again excluding the example from Section 3.5) here together form a minimum set of examples that should be verifiable by a program verification technique that aspires to deal with complete coverage (of sequential Java). A truly canonical set of examples should –at the very least– also cover all possible combinations of the (sub)categories identified here.

3.6.1 LOOP & ESC/Java2

All examples have been verified and proved to be correct with the LOOP tool, at the cost of some serious user interaction in the theorem prover PVS [PVS]. In contrast, ESC/Java2 had problems with several of the examples discussed above. This was mostly because the tool lacked semantics for properties like non-termination and bounded numeric types. However, ESC/Java2 can be used completely automatic, which makes using the tool a lot easier.

The main difference between the tools is that all the Hoare rules used in the LOOP tool are proved to be sound with respect to the underlying Java semantics. This has not yet been done for ESC/Java2's program logic and so occasionally soundness or completeness bugs

[29] Except for the obscure (but fun!) example concerning mutual referring fields in static initialization from Section 3.5.

are found[30]. Another problem for ESC/Java2 is the back-end automatic theorem prover it uses: Simplify. Simplify's problems with numeric types makes a bitvector semantics almost impossible to use. Currently, the use of other back-end provers is considered to get rid of this problem.

For most users the assurance a tool like ESC/Java2 (currently) can give will be high enough. Its automation and scalability will be far more important features then the occasional soundness bug (which become rarer and rarer). Only for small core parts of programs where the highest assurance levels are required, a tool like the LOOP tool would still be better suited.

Finally, the fact that our example programs are small does not mean that we think size is unimportant. On the contrary, once a reasonably broad semantic spectrum is covered, another challenge is to scale up program verification techniques to larger programs. In this respect, ESC/Java2 is far superior to LOOP. With LOOP we are currently able to verify programs with hundreds of lines of code (see e.g., [BCHJ05, JMR04]), while ESC/Java2 can verify thousands of lines of code (see e.g., [HJKO04]).

It turned out that the Findbugs tool is very useful in finding 'patterns' in Java that –though semantically correct– are clearly unwanted in a program. Notable examples are the covariant overloading in Section 3.4.1 and the mutually depended static fields from Section 3.5.1.

This chapter shows that it is possible to verify the correctness of specifications of (semantically) non-trivial programs in a complex language like Java. If we concentrate on security, then the obvious next step will be to express security properties in a specification language like JML. Typical examples of security properties that have already been formalized in JML are the absence of certain exceptions at the top level, no unwanted overflows and well formed transactions [PBB+04, JMR04]; others [Mos05b] have verified similar properties for JAVA CARD applets, but have not specified them in any high level specification language. In Chapter 5 we will elaborate more on the specification of security properties in JML.

In the next chapters we will specify and verify several other security properties –namely correct control flow and confidentiality– for a JAVA CARD applet in JML.

[30] We found one such completeness bug in ESC/Java2 when we verified the example in Section 3.4.1, ESC/Java2 (incorrectly) always uses the static type of a calling object to determine which method to execute.

Chapter 4

Specification and verification of control flow properties

Reasoning about the security of software systems can, in general, be done on two distinct levels. The first one is a high abstract level where implementation details or a particular programming language are abstracted away. A typical example of reasoning techniques on this level forms the work on security protocols [Pau98, Low95, BAN89, Jac04a, Mea96, CM04, GHJ05, Cor06] (also see [Mea03] for an overview of this field). Here, different principals are distinguished that communicate with each other by sending abstract messages over some untrusted network. Cryptographic primitives such as random nonce generation, one-way hashes, encryption, signatures, etc. are used to construct these messages. Security protocols are designed to establish high level properties such as authenticity, confidentiality and data integrity. Other abstract security analysis techniques include, amongst others, work on access control [SCFY96, ABLP93] and multi-level security [LB73, Bib77].

The second level is the low concrete level of program code (in the case of this thesis, JAVA CARD [Che00] source code). Specifications on this level usually consist of pre- and postconditions for methods and invariants or history constraints for a class as a whole. It is generally acknowledged that there is a substantial gap between these high and low level approaches, and that bridging this gap is very important. In this chapter we shall try to narrow this gap from the bottom upward. We choose to take this lowest level as starting point because:

- Certification of specific, security sensitive products is expected to focus primarily on the actual implementation, and not so much on a high level abstract description of the system;
- Refinement is a problematic notion within the area of computer security: it may well happen that by adding implementation details certain crucial security properties that can be established at a high abstraction level no longer hold at a low level (because an attacker has more material to exploit).

Within the setting of this chapter we study a payment system (a phone card) that can be realized with (Java) smart cards, running a specific applet. The applet involves a balance,

which represents an amount of electronic money. In general with such payment systems, there are separate protocols for paying (debiting the balance), and for charging (crediting the balance). Our debit protocol is not very elaborate because it does not involve any form of authentication. Crediting however is less trivial. It involves a challenge-response mechanism in which the terminal needs to authenticate itself and in which at most five tries are permitted. The smart card applet that we have developed involves an implementation of this protocol for crediting.

Our goal in this chapter is twofold: First we want to illustrate our current program specification and verification technology on a somewhat larger example. Thereby proving the correctness of the applet in general, and the crediting protocol in particular[31]. Second, we introduce a new applet development methodology, consisting of:

- Describing the control flow (of the applet-part of the protocol) as a finite state machine;
- Translating the transitions of this finite state machine into a class constraint, involving a special '`state`' variable capturing the states.

Elaborating the details of this methodology forms the main contribution of this chapter. It is based on [JOW04].

The general approach taken in this chapter is bottom-up. Rather than attempting to specify and verify a fully featured industrial purse applet, we describe a JAVA CARD applet of our own creation with minimal functionality, yet whose correctness is formally proved. The applet features some cryptography, which is traditionally thought of as hard to specify on the source code level and which is therefore usually specified on the (abstract) security protocol level. Therefore it plays only a minor role in our verification effort. However, since vulnerabilities are often introduced in the implementation rather than the specification of software, in order to establish true security, it is necessary to take the *use* of cryptographic operations seriously on the level of program code.

It should be emphasized that the Java code in this chapter is our own. It is written for the purpose of verification. It has been compiled and even been loaded onto a smart card, however it has—deliberately—not been tested. We wanted to see what kind of errors could be discovered with our verification technology alone, without actually running the code. And we did find several program bugs, see Section 4.3. Yet, most of the mistakes occurred in the specifications we wrote. Once again it became clear that writing good specifications is much harder than writing good programs.

We want to stress that in this chapter we only look at source code on the card-side of the whole debit protocol. For the terminal-side one can also consider an appropriate finite state machine, as done in [HOP04], possibly also with code verification.

[31] Note that the restriction to current specification technology is very relevant, and excludes, for instance, specification language extensions such as in [BH02]. As a result, the cryptographic aspects in specification and verification are not dealt with at all. On the other hand, our approach works well for the control-flow aspects of the crediting protocol.

The remainder of this chapter is organized as follows. The next section gives a high level description of the applet. Section 4.2 describes how the specification of the JAVA CARD applet came about. Section 4.3 sheds some light on different aspects of the formal proof process. Related work is discussed in Section 4.4, and the chapter ends with conclusions in Section 4.5.

4.1 The applet

The applet we discuss in this chapter has been developed and implemented by ourselves. It can be viewed as a simple rechargeable phone card: the owner of the card can credit some money on it and use it in a telephone booth. When in use the amount of money on the card is debited.

In the next sections we discuss the high-level functional and security requirements that the complete system should conform to, we give some comments about the implementation of the applet and we discuss the security protocol that is used to credit the applet.

4.1.1 Requirements

We think the following functional requirements are reasonable for such a purse applet:

1. Crediting the card should be done on-line, connected to the bank.
2. It should be possible to debit the card off-line.
3. Anybody can debit the card (until the money runs out).

Thus, the idea is that anybody can use the card for making calls and that it is not necessary for the bank to validate each transaction directly as is the case with, for example, credit cards. The bank only has to approve the transfer of money when the card is credited.

Besides these functional requirements there are several security requirements. They follow from the functional requirements above and from the requirements of the different parties involved in using such a card: the card issuer (bank), the card owner (bank client) and the service provider (phone company). We have identified the following security requirements:

4. Cards cannot be forged or cloned.
5. Only the legitimate owner of the card (and corresponding bank account) can credit it.
6. The card owner can only credit with as much money as he has in his bank account.
7. The card should lock itself after a certain number of consecutive tries to credit it have failed.

All these requirements must be taken into account when we design the applet and the systems with which it interacts. Although we focus on the applet itself, we do sketch the high level design of the complete system without going into implementation details on the terminal or bank side.

4.1.2 Design

In this system the card owner will need three things: a card, a personal identification number (PIN) and a bank account (with enough money on it). A specific card is coupled with one bank account. Only someone who has the card and knows the PIN, i.e., the card owner, can credit the card (item 5 above), see Section 4.1.4 for the specific crediting protocol (item 1). Of course, on the bank side it will be checked that the owner has enough money in his bank account when the card is credited (item 6).

Anybody can use the card for making phone calls (item 3 above). This means in particular that somebody who found (or stole) the card can debit it until the card is out of money, yet it should not be possible for anybody besides the owner to credit the card again. In our implementation we have chosen to use an extremely simple payment method: there is one dedicated command which simply decrements the amount on the card by one (item 2), provided there is still money on the card. A more realistic debit protocol would involve authentication of the card to the terminal –to prevent the use of fraudulent cards– and possibly also the functionality to debit larger sums of money at once. For presentation purposes we have not implemented such a debit protocol because it is similar to the crediting protocol we use and does not add anything –besides complexity– to the methodology we propose in this chapter. It is relatively straightforward to extend the applet with such a debit protocol, by using a second key and a challenge-response protocol where the card authenticates itself to the terminal as genuine.

Smart cards are what is often called *tamper resistant devices* [RE00]. The limited interface of a smart card allows one to send and receive data to and from it, but confidentiality of critical data (like cryptographic keys), as well as integrity of other data (like a balance field) is ensured. If the card is indeed tamper resistant then item 4 should be satisfied as well.

The last security requirement, item 7, is ensured in the implementation and will be discussed in depth in the next sections.

4.1.3 Implementation

An electronic telephone card applet should, at the very least, contain a `balance` field, say of type `short`. After card issuance it should be possible to decrease `balance` and to consult its current value. Initially `balance` is set to zero, yet the card can be credited by charging terminals. This means the applet needs a `key` field as well. The key is supposed to be loaded onto the card before the card is issued. A symmetric key is used which is shared by any legitimate terminal. Besides this key, the applet also gets a personal ID, which is used by the terminal to generate the key (see Section 4.1.4). The card authenticates

a terminal by sending its (card) ID and a challenge. Only a terminal equipped with appropriate credentials can respond to this challenge by sending the nonce back encrypted with the key. The applet keeps track of the number of consecutive unsuccessfully answered challenges in a `counter` field. As soon as `counter` reaches five, the card locks itself.

To make the verification of the applet a little bit more challenging, we restrict `balance` to 12 bit signed values[32], to be interpreted as the number of euro cents on the card. This means that the maximum amount on the card is 40 euros and 96 cents, which seems reasonable for a simple electronic purse. It also means that the applet's source code will contain several bitwise operations such as shifts and masking. The LOOP tool has support for verification of such operations, see [Jac03].

Recall from Chapter 2 that smart cards are instructed to perform certain operations through APDUs (Application Protocol Data Units). Within a JAVA CARD applet the `process` method is responsible for managing the incoming command APDUs once the applet has been selected. Every command APDU contains an instruction byte which is examined by the `process` method. The applet will throw an `ISOException` whenever the terminal sends a command that is not understood or not appropriate in the applet's current state. Exceptions show up on the terminal side as special return codes in the response APDU.

Instruction bytes for the electronic purse applet described above are encoded inside the applet as follows:

- `INS_SETKEY_AND_ID` sets the 56 bit DES (Data Encryption Standard) key used for crediting the card and the personal ID of the applet. The data field should contain the eight bytes of key data[33] and the eight bytes forming the applet ID. The response APDU is empty.

- `INS_GETVAL` returns the value of `balance`. The response APDU that is to be expected contains a data field of two bytes with the current balance.

- `INS_DECVAL` decreases the value of `balance` with 1 cent. The response APDU is empty.

- `INS_GETCHAL` asks the applet to generate a challenge (a random nonce) and send it, together with the applet ID, back to the terminal application. The response APDU that is to be expected contains the 32 byte nonce and the 8 byte ID. The applet increases the `counter` field for each requested challenge.

[32] The choice of 12 bit is somewhat arbitrary, other values would have sufficed just the same. However, representing a currency as low level bits *is* realistic; it shows the kind of low-level details that are often encountered in (commercial) JAVA CARD applets. Verifying correctness conditions in the presence of such low level operations forms a real challenge. See also Chapter 3.

[33] The JAVA CARD API needs eight bytes (64 bits) to generate a (single) DES (56 bits) key. Although DES uses a block size of 64 bits, the length of the key was restricted to 56 bits. Historically, this is usually attributed to the NSA who could apparently break 56 bits keyed DES encryption (around 1974) but not 64 bits key encryption (see e.g., [Lev01]). The redundant 8 bits are typically used as check digits for the actual key, see e.g., [PP03].

– `INS_RESPOND` responds to the card challenge by sending a 24 byte encrypted hash of the nonce. The APDU also includes a new value for `balance` in the last two bytes of the decrypted cipher text array. A successful response to a challenge resets the `counter` field. The response APDU is empty.

4.1.4 The crediting protocol

The key on the card is a *diversified key* [AB96]. This means that a legitimate terminal can generate the card key if it has the appropriate information. In this particular case this means that the terminal should know the card ID and the PIN (provided by the card owner). It can then generate the card key K_c using a master key K_m as follows:

$$K_c = \{ID \mathbin{+\!\!+} PIN\}_{K_m} \qquad (*)$$

where $+\!\!+$ is concatenation.

The use of a diversified key ensures that if the key of one card is discovered all other cards are not compromised while the terminal only needs to store one key (instead of a key for each card). Notice that for the implementation of the applet this is irrelevant.

The protocol implemented by the methods `getChallenge` and `respond` is now rather straightforward. It can be formulated in standard security protocol notation as:

$$\begin{array}{lcl} C \longrightarrow T & : & \mathsf{Nonce,ID} \\ T \longrightarrow U & : & \mathsf{PIN?} \\ U \longrightarrow T & : & \mathsf{PIN} \\ T \longrightarrow C & : & \{\mathsf{Hash(Nonce)}, \mathsf{NewValue}\}_{K_c} \end{array}$$

where C stands for the card, T for the terminal, U for the user and K_c for the card key, which is computed as in $(*)$ above.

If the response from the terminal checks out, the current value of `balance` is replaced by NewValue. Note that since the new value together with a hash of the nonce is also encrypted, a replay or a man-in-the-middle attack is not possible as an attacker would have to know K_c to change NewValue. Even though this is a very basic authentication protocol, we checked the absence of such attacks by using the security protocol compiler Casper [Low98] in combination with the model checker FDR [For00]. It presented no problems.

As an aside, we have chosen to develop and implement our own cryptographic protocol. However, it is also possible to use a dedicated API, such as the one provided by the Global Platform API[34], which ensures that the communication channel (provided by the APDUs) is encrypted. We did not consider using such a secure channel approach for three reasons: (i) using a secure channel for our simple applet is a bit of overkill and (ii) the verification effort would be less interesting since it would simply involve some API calls without actually showing what is going on at the protocol level. Of course, in commercial

[34] http://www.globalplatform.org/

applets the added abstraction of the Global Platform API can be very useful. Finally, (iii) not all cards implement this Global Platform.

4.2 Specifying the applet

In this section we explain the specification process for our applet. The starting point for the specification forms the life-cycle of the applet, which we model using an automaton. A complete listing of the JAVA CARD applet together with its JML specification (proved to be correct with respect to the implementation) can be found in appendix A.1.

4.2.1 Modeling the card life cycle

A rudimentary lifecycle model for a general smart card applet, comparable to those described for example in [MM01, Glo01] or [RE00, Chapter 10], can be visualized using a simple automaton. Figure 4.1 shows such an automaton. Usually the model identifies different stages of an applet's life, for example: *Installation*, *Personalization*, *Processing*, and *Locked*. Of course, for concrete applets the life stages can usually be refined into more detailed stages, in particular the *Processing* life stage, as we shall see below.

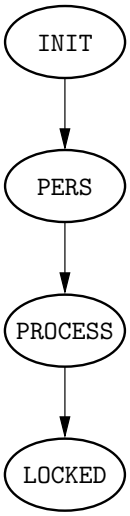

Figure 4.1: The default lifecycle model of an applet.

The applet's current state is made explicit in the implementation by using a dedicated field called `state`. Since the applet should *always* be in one of the states `PERS`, `PROCESS` or `LOCKED`[35] these modes will be represented by constants in the implementation, note that

[35] We ignore stage INIT here, since this state is only active when the applet is loaded on the card. At

these constants should be stored in the persistent (permanent) memory of the smart card[36]. This ensures that the states are indeed constants and cannot change.

In the case of the phone card applet we can further refine the `PROCESS` stage. Two states can be identified: `ISSUED`, when the card is in normal operation and `CHARGING`, when the applet is in a special state that is related to crediting the phone card. Moreover, we can associate state transitions (the arrows in the figure) with applet commands. Not every command is always appropriate. For instance, APDUs with the `INS_SETKEY_AND_ID` instruction byte should be ignored by the applet once the card has been personalized.

Figure 4.2 displays the automaton that refines the `PROCESS` state.

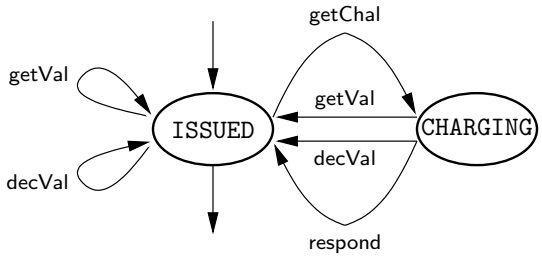

Figure 4.2: Refinement of the `PROCESS` state.

We have two choices when we combine the automata from Figure 4.1 and Figure 4.2:

1. We can model the new states `ISSUED` and `CHARGING` by a new state variable making them completely independent of the other states already modeled by the variable '`state`'. In this case we have the additional requirements that these states are only accessible when we are in the `PROCESS` state and that they are stored in transient memory[37]. Mostowski proposes this approach in his PhD thesis [Mos05a, Paper I].

2. We can simply replace state `PROCESS` by state `ISSUED` and add a new state `CHARGING`. Both states can be then be stored in persistent memory.

the end of this process the `install` method (which calls the constructor and registers the applet with the JCRE) is called and the applet will change its state to `PERS`, see [Che00, §3.10].

[36] A Java Card enabled smart card has persistent (permanent) memory, that keeps its contents when the card has no power source, and transient (temporary) memory, that is cleared each time the card is powered up. See Section 2.1, for a detailed explanation about the memory of Java Card enabled smart cards.

[37] By using transient memory and assigning `ISSUED` the default value 0 and charging some other value, it is ensured that card tears always return to state `ISSUED`.

We have chosen the second option because of its simplicity. However, in larger applets it might be clearer to keep the standard states from Figure 4.1 and refine those states further with separate automata. Another reason why one might choose the first option is that the persistent memory is realized in EEPROM, which is a lot slower and easier to break than the transient memory that is implemented using (fast) volatile memory. Another reason might be that you might want to go back to the `ISSUED` state after a card tear (for security reasons). Storing the new state in transient memory would ensure this.

The combined automaton can be seen in Figure 4.3. In state `PERS` the applet can be personalized by setting the key and corresponding ID. The `state` will then change to `ISSUED`. As soon as the terminal requests a challenge, `state` is switched to `CHARGING`. The subsequent command will switch `state` back to `ISSUED`. When five or more challenges are left unanswered, `state` becomes `LOCKED` and the applet will no longer respond to commands.

Figure 4.3 (and Figure 4.4 as well) also specifies that it is only allowed to call specific methods in certain states, e.g., only the `setKey` method can be called in the `PERS` state. Thus, if a method (represented by an arrow) does not start in a particular state then it means that the corresponding method cannot be called in that state. This behavior is enforced by the `process` method (see Section 4.2.2).

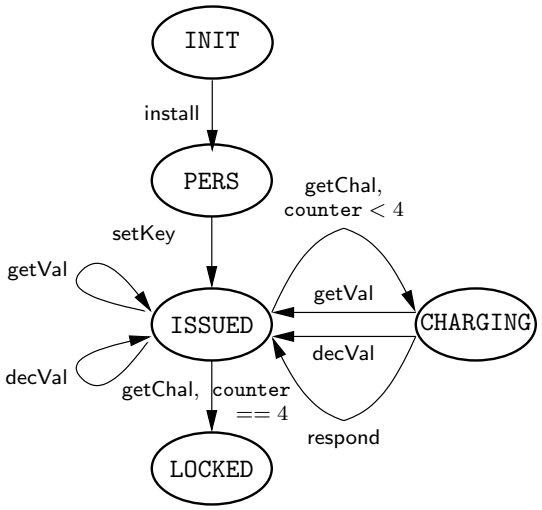

Figure 4.3: A composed automaton describing the control flow of the applet.

In order to obtain a yet more complete overview of the possible control flow of the applet we add one more aspect to the finite state machine from Figure 4.3: exceptions. All the methods of the applet can throw exceptions. We have modeled this in Figure 4.4 with branching arrows, dotted arrows represent exceptional control flow.

The automaton in Figure 4.4 clearly visualizes the possible flows of control of the applet. Some points to note:

- Both method install and method setKey are atomic in the sense that they either completely execute, and end up in a new state, or fail with an exception and return to the state in which the method was called.

  In JAVA CARD one can use a dedicated transaction mechanism [Che00, Chapter 5] to enforce this kind of behavior. We have excluded the transaction mechanism here because we have not modeled it inside the LOOP framework. A number of other researchers have studied JAVA CARD's transaction mechanism in more detail see e.g., [HP04, BM03, HM05].

- Depending on the value of the `counter` field, method getChal either terminates normally and ends up in state `CHARGING`, or it terminates with an exception and ends up in state `LOCKED`.

- All method calls in state `CHARGING` end up in state `ISSUED` regardless of termination behavior.

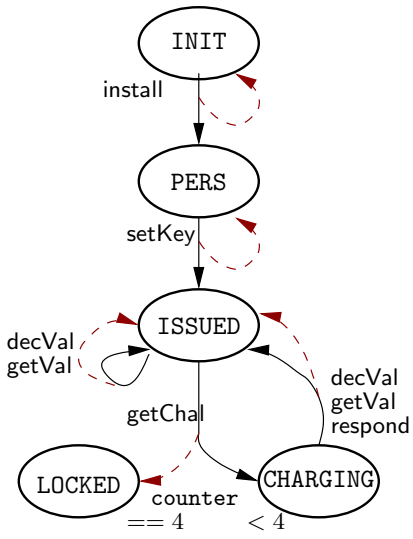

Figure 4.4: Control flow model extended with exceptional behavior.

Such an explicit representation of the applet's state makes formulating global properties much easier when writing the specification. In fact, some of the applet's code and specification can be systematically derived from the finite state machine, as we shall see in the next section.

4.2.2 The process method

In the implementation of the `process` method we created dedicated helper methods for each of the possible instruction bytes:

Java

```
public void process(APDU apdu) throws ISOException {
  byte ins = apdu.getBuffer()[OFFSET_INS];
  if(selectingApplet())
      return;
  switch(state) {
    case PERS:
      switch(ins) {
        case INS_SETKEY_AND_ID: setKey(apdu); break;
        default: ISOException.throwIt(
                    SW_CONDITIONS_NOT_SATISFIED);
      }; break;
    case ISSUED:
      switch(ins) {
        case INS_GETVAL: getValue(apdu); break;
        case INS_DECVAL: decValue(apdu); break;
        case INS_GETCHAL: getChallenge(apdu); break;
        default: ISOException.throwIt(
                    SW_CONDITIONS_NOT_SATISFIED);
      }; break;
    case CHARGING:
      switch(ins) {
        case INS_GETVAL: getValue(apdu); break;
        case INS_DECVAL: decValue(apdu); break;
        case INS_RESPOND: respond(apdu); break;
        default: ISOException.throwIt(
                    SW_CONDITIONS_NOT_SATISFIED);
      }; break;
    case LOCKED: ISOException.throwIt(
                    SW_SECURITY_STATUS_NOT_SATISFIED);
    default: ISOException.throwIt(
                SW_CONDITIONS_NOT_SATISFIED);
    }
}
```

These helper methods are only called from the `process` method, which consists of two levels of nested `switch` statements. The outermost `switch` performs a case analysis on the `state` field, while the innermost switches examine the instruction byte to call the corresponding helper method. The `selectingApplet()` method in the beginning of the `process` method returns true if the applet is selected, and false otherwise. Since we do not

need to do anything special in case the applet is selected we simply end the method via a `return` statement.

4.2.3 Global properties of the applet

Because the high level properties of the applet are captured in global invariants and constraints, the specification of the `process` method itself is not very interesting. It is shown here for completeness sake.

JML

```
/*@ behavior
  @    requires apdu != null && random != null
  @             && cipher != null && digest != null;
  @  assignable state, balance, counter, apdu.buffer[*],
  @             tmp[*], id[*], key[*], cipher, nonce[*];
  @     ensures true;
  @     signals (ISOException) true;
  @*/
```

The local specifications (in terms of pre- and postconditions) of the dedicated methods capture the more detailed functional behavior of those methods. An example of such a method specification is given in Section 4.3.2 for the methods dealing with APDUs whose instruction byte is `INS_GETCHAL` or `INS_RESPOND`. Those specifications exactly describe the behavior of the `counter` field as a result of the method invocations. The global properties are stated in the class invariant below and the class constraint on the next pages.

JML

```
/*@ invariant
  @ (state == PERS || state == ISSUED ||
  @            state == CHARGING || state == LOCKED)
  @   && 0 <= counter && counter <= 5
  @   && (state == LOCKED <==> counter == 5)
  @   && (state == PERS ==> counter == 0)
  @   && 0 <= balance && balance <= 4096
  @   && key != null && key.length == DES_KEY_SIZE
  @   && nonce != null && nonce.length == NONCE_SIZE
  @   && id  != null && id.length == ID_SIZE
  @   && tmp != null && tmp.length == TMP_SIZE
  @   && sha_nonce != null && sha_nonce.length == SHA_SIZE
  @   && plaintxt != null && plaintxt.length == SHA_SIZE+2
  @   && ciphertxt != null
  @   && ciphertxt.length == CIPHERTEXT_SIZE;
  @*/
```

The first four conjuncts are the most interesting. They express that the applet can be only in one of four states (`PERS`, `ISSUED`, `CHARGING` or `LOCKED`), that the `counter` that keeps track of the number of unsuccessful challenge-responses is at least zero and at most five, that the state `LOCKED` corresponds to the counter having the maximum value five, and that in the `PERS` state the `counter` equals zero.

The class constraint conveys more interesting information as it captures the intended flow of control. A constraint expresses a relation between the pre-state (indicated by `\old`) and the post-state, that must hold for all method invocations. Part of the constraint is generated automatically using the *AutoJML* tool described in [HOP03, HO03]. To emphasize this fact, we list two constraints. In the actual verification of the applet these are simply conjuncted.

The eight parts of the generated constraint describe the possible transitions from the different states. Note that sending a command that is not supported by the applet does not cause a change of state. This induces reflexive transitions from each state to itself, which are not explicitly specified in Figure 4.3 and Figure 4.4.

JML

```
/*@ constraint
  @    (state == LOCKED ==> \old(state) == ISSUED
  @                         || \old(state) == LOCKED) &&
  @    (state == PERS ==> \old(state) == PERS) &&
  @    (state == ISSUED ==> \old(state) == PERS
  @                         || \old(state) == ISSUED
  @                         || \old(state) == CHARGING) &&
  @    (state == CHARGING ==> \old(state) == ISSUED
  @                         || \old(state) == CHARGING) &&
  @    (\old(state) == LOCKED ==> state == LOCKED) &&
  @    (\old(state) == PERS ==> state == ISSUED
  @                         || state == PERS) &&
  @    (\old(state) == ISSUED ==> state == ISSUED
  @                         || state == CHARGING
  @                         || state == LOCKED) &&
  @    (\old(state) == CHARGING ==> state == ISSUED
  @                         || state == CHARGING);
  @*/
```

The second constraint (on the next page) was manually entered and captures the relation between the `balance`, `counter` and `state` fields. Its first conjunct expresses that once the applet becomes `LOCKED`, it will remain `LOCKED` and the balance and counter do not change anymore.

The second one expresses that three things can happen in the normal state of operation (i) `ISSUED`: the applet can stay in the same state, in which case the balance can only decrease –a crucial security property– and the counter will not change; (ii) the applet can

go into the CHARGING state, then the balance will not change, the counter will increase with one, and the value of the counter in the pre-state was less than four; (iii) the applet will become LOCKED, in which case the balance will stay the same, the counter will again increase with one and in the pre-state the counter field had value four[38]. The latter constraint describes another crucial security property for this applet. Notice that both properties taken together implement security requirement number 7 from Section 4.1.2.

JML

```
/*@ constraint
  @    (\old(state) == LOCKED ==>
  @       (balance == \old(balance) &&
  @        counter == \old(counter) &&
  @        state == LOCKED))         &&
  @    (\old(state) == ISSUED ==>
  @       ((state == ISSUED && balance <= \old(balance) &&
  @          counter == \old(counter))
  @       || (state == CHARGING && balance == \old(balance) &&
  @          counter == \old(counter + 1) && \old(counter) < 4)
  @       || (state == LOCKED && balance == \old(balance) &&
  @          counter == \old(counter + 1) &&
  @      \old(counter) == 4))) &&
  @    (\old(state) == CHARGING ==>
  @       ((state == ISSUED && counter == 0)
  @       || (state == ISSUED && counter == \old(counter)
  @                           && balance <= \old(balance))));
  @*/
```

The third conjunct expresses the two possible transitions from CHARGING to ISSUED. Either the response was valid, in which case the counter is reset to zero, or the response was invalid, in which case the counter remains unchanged (recall that the getChallenge method already incremented the counter) and the balance does not increase.

4.3 Correctness of the applet specification

Once we have a (JML) specification of the intended behavior of our applet, we want to make sure that it actually holds for the given (Java) implementation. As already explained in Chapter 2, this is done by first translating the Java + JML code into the language of the theorem prover PVS, via the LOOP tool [BJ01]. Next, the theorem prover is

[38]The observant reader might notice that some of the properties specified in the invariant and constraint are redundant. This is done deliberately, since we believe this makes the specification easier to read and thus understand. The redundancy in the specification does not hinder the verification in PVS.

used for (interactive) verification. The actual verification in the PVS [OSRSC99] proceeds via a special Hoare logic for JML [HJ00b], in combination with a weakest precondition calculus [Jac04b]. For the verification in the next sections we have used LOOP's integer arithmetic that models Java's integer arithmetic precisely, which includes overflowing and widening and narrowing of numeric types, see [Bre06, Chapter 4] and[Jac03].

Although the complete applet has been formally verified, discussing the verification of the whole applet would present too many details. Instead we concentrate on the `process` method, described in the previous section, and on two additional helper methods that are called by `process`, namely `getChallenge` and `respond`, see below.

The most important properties to be verified are the class-wide invariant and constraint properties. They express key security properties that should be established by all methods. In the semantics of JML –as incorporated in the LOOP tool– invariants are implicitly added to all preconditions, and to all postconditions, both for normal and abnormal termination. Similarly, constraints are added to all postconditions. At first sight it might seem easy to establish the trivial (normal and abnormal) postcondition `true` of the `process` method. However in reality, these postconditions are far from trivial because they involve the invariant and constraint.

4.3.1 Verifying the process method

The verification of the `process` method essentially involves making the various case distinctions in PVS, guided by the nested `switch` statements inside the `process` body[39]. In each case an appropriate helper method is called. During verification of the `process` method, we then *use* the specification of the method that is called. This means that the method's precondition must be established, so that its postcondition can be used in the remainder of the proof.

4.3.2 Verification of the two helper methods

The two most prominent helper methods, `getChallenge` and `respond`, are discussed in detail in this section. The `getChallenge` method can only be called when `state` is `ISSUED` and the incoming APDU contains the `INS_GETCHAL` instruction.

Java

```
/*@ behavior
  @    requires state == ISSUED
  @               && buffer[OFFSET_INS] == INS_GETCHAL
  @               && counter < 5
  @               && apdu != null
```

[39]It was at this stage that we found a serious program bug: in our first version, the outer `break` statements after the three inner switches in `process` were not there. But the inner breaks only break out of the inner switches, not out of the outer one!

```
  @               && apdu.buffer != null
  @               && apdu.buffer.length >=
  @                    OFFSET_CDATA + NONCE_SIZE + ID_SIZE
  @               && random != null;
  @  assignable state, counter, nonce[*], apdu.buffer[*];
  @     ensures state == CHARGING
  @               && counter == \old(counter) + 1
  @               && counter < 5;
  @     signals (ISOException)
  @              state == LOCKED && counter == 5;
  @*/
private void getChallenge(APDU apdu) throws ISOException {
  counter++;
  if (counter < 5) {
     state = CHARGING;
     random.generateData(nonce,(short)0,NONCE_SIZE);
     apdu.setOutgoing();
     apdu.setOutgoingLength(NONCE_SIZE + ID_SIZE);
     byte[] buffer = apdu.getBuffer();
     Util.arrayCopy(
       nonce,(short)0,buffer,OFFSET_CDATA,NONCE_SIZE);
     Util.arrayCopy(id, (short)0,
       buffer,ISO7816.OFFSET_CDATA+NONCE_SIZE,ID_SIZE);
   } else {
     state = LOCKED;
     ISOException.throwIt(SW_SECURITY_STATUS_NOT_SATISFIED);
   }
}
```

It provides in the outgoing APDU a new, randomly generated nonce, to be used in the challenge-response mechanism for authenticating the terminal, together with the applet ID, which the terminal needs in order to generate the card key. For security reasons, we keep track of how many times the `getChallenge` method is called until a challenge is answered by an appropriate response. Technically, we use a private byte field `counter` –in persistent memory– which is incremented with every call to `getChallenge` as explained in Section 4.2. Thus, our `getChallenge` method must inspect the value of this `counter`.

Notice that the precondition contains certain requirements about the length of the buffer of the incoming APDU. These requirements are actually redundant, because of the invariant that we use for the `APDU` class. The important point is that the `getChallenge` method only generates a challenge if the counter (in the pre-state) is less than 4. Recall that the invariant says $0 \leq$ `counter` ≤ 5. If `getChallenge` is called with `counter` equal to 4, the applet becomes locked. And the `getChallenge` method cannot be called by `process` with `counter` equal to 5, because according to the constraint this means that the applet is already locked. We thus see the importance of the invariant and constraint in determining

the appropriate flow of control.

We briefly discuss the verification of `getChallenge`. There are three points worth mentioning.

– Shortly after entering the method body, the counter is incremented and so the class invariant is broken: the logical equivalence `state == LOCKED <==> counter == 5` need not hold anymore. But at the end of the method the invariant is re-established. Our semantics and proof methods can handle such temporary violations of invariants.

  Re-establishing the class invariant at the end of the method body is sufficient for JAVA CARD programs. For JAVA CARD applets, however, a card-tear is always possible. Withdrawing the card from the terminal in the middle of the execution of this method could leave the applet in a state in which the invariant no longer holds. A solution to this is to use JAVA CARD's transaction mechanism. Unfortunately, verification of transaction blocks is not supported by the LOOP tool yet[40].

– The API methods that occur in the `then` part are handled via their specifications. These specifications are fairly weak, and do not express any functional behavior. For instance, the specification that we used for the `javacard.security.RandomData` method `generateData` can be seen below.

JML

```
/*@ normal_behavior
  @    requires buffer != null && offset >= 0 &&
  @             length >= 0 &&
  @             offset + length <= buffer.length;
  @  assignable buffer[*];
  @     ensures true;
  @*/
public abstract void generateData(
            byte[] buffer, short offset, short length);
```

  This serves our purposes. We use similarly weak specifications for the `setOutgoing` and `setOutgoingLength` methods from API class `javacard.framework.APDU`. But the `javacard.framework.Util` method `arrayCopy` has a functional specification—as in [BJP01].

[40] In particular, the LOOP framework has no support for transaction mechanisms on a *semantic* level. A program transformation approach –as proposed in [HP04]– that removes the transaction and replaces them by a suitable `try-catch-finally` construct can be used.

– The two methods `setOutgoing` and `setOutgoingLength` may generate an `APDU-Exception`. But this is a runtime exception that we ignore in our specification of `getChallenge`. Of course, one can choose to include it and have a more detailed specification.

The implementation of the `respond` method can be seen here:

Java

```
private void respond(APDU apdu) throws ISOException {
  state = ISSUED;
  byte[] buffer = apdu.getBuffer();
  if (buffer[OFFSET_LC] != ciphertxt.length) {
     ISOException.throwIt(
        SW_SECURITY_STATUS_NOT_SATISFIED);};
  readBuffer(apdu,ciphertxt);
  cipher.doFinal(ciphertxt,(short)0,
                 CIPHERTEXT_SIZE,plaintxt,(short)0);
  digest.doFinal(nonce,(short)0,
                 NONCE_SIZE,sha_nonce,(short)0);
  if (Util.arrayCompare(sha_nonce,(short)0,
                        plaintxt,(short)0,SHA_SIZE) == 0) {
      balance = (short)(((plaintxt[SHA_SIZE] & 0x0F) << 8)
                        | (plaintxt[SHA_SIZE+1] & 0xFF));
      counter = 0 };
  else {
     ISOException.throwIt(
        SW_SECURITY_STATUS_NOT_SATISFIED);
  }
}
```

The `respond` method of our phone card applet can only be called (within the `process` method from the previous section) when `state` is `CHARGING` and the incoming APDU contains the `INS_RESPOND` instruction. The method checks whether the received response is appropriate to the challenge previously sent, and if so, it extracts the new value for `balance` from the APDU buffer through some bitwise shifting and masking. (The resulting balance is described in slightly more readable form in the `ensures` clause.) The method then resets the `counter` field to zero. Whether or not the appropriate response is given is expressed in the JML specification, which can be found below.

The verification of the `respond` method is similar to the one for `getChallenge`. The proof cannot be handled automatically with weakest-precondition calculus because there are too many complicated method calls involved. Therefore, we have produced a proof in Hoare logic, which requires substantial user interaction.

JML

```
/*@ behavior
  @   requires state == CHARGING &&
  @            buffer[OFFSET_INS] == INS_RESPOND &&
  @            counter < 5 &&
  @            apdu != null &&
  @            apdu.buffer != null &&
  @            apdu.buffer.length >= CIPHERTEXT_SIZE &&
  @            cipher != null &&
  @            ciphertxt != null &&
  @            digest != null;
  @ assignable state, counter, balance,
  @            ciphertxt[*], plaintxt[*], sha_nonce[*];
  @    ensures state == ISSUED &&
  @            balance == (short)
  @             (256 * (plaintxt[SHA_SIZE] & 0x0F)
  @                       + (plaintxt[SHA_SIZE+1] & 0xFF)) &&
  @            counter == 0 &&
  @            (\forall int i; 0 <= i && i < SHA_SIZE ==>
  @                              sha_nonce[i] == plaintxt[i]);
  @    signals (ISOException e) state == ISSUED &&
  @            balance == \old(balance) &&
  @            counter == \old(counter) &&
  @            (apdu.buffer[OFFSET_LC] != ciphertxt.length
  @            ||
  @            !(\forall int i; 0 <= i && i < SHA_SIZE ==>
  @                       sha_nonce[i] == plaintxt[i]));
  @
  @*/
```

The resulting proof tree in PVS is included in Figure 4.5, and gives an impression of the complexity and number of interactions (each node corresponds to a user-command). The reader is not expected to analyze the information in each node. The bitwise operations involved present no problems because we have a detailed semantics of Java's integral types formalized in PVS [Jac03, Bre06].

4.4 Related work

As far as we know, the paper [JOW04] on which this chapter is based was the first to describe a formal verification of a Java smart card applet which uses cryptographic methods (in the form of a simple challenge-response mechanism). Other approaches, such as in [Mos02], concentrate on the development process, and stop short of the actual verification. Verification of JAVA CARD applets has since been done by a number of researchers, e.g., [BCHJ05, PBB+04, Mos05b]. However these are case-studies showing the progress of

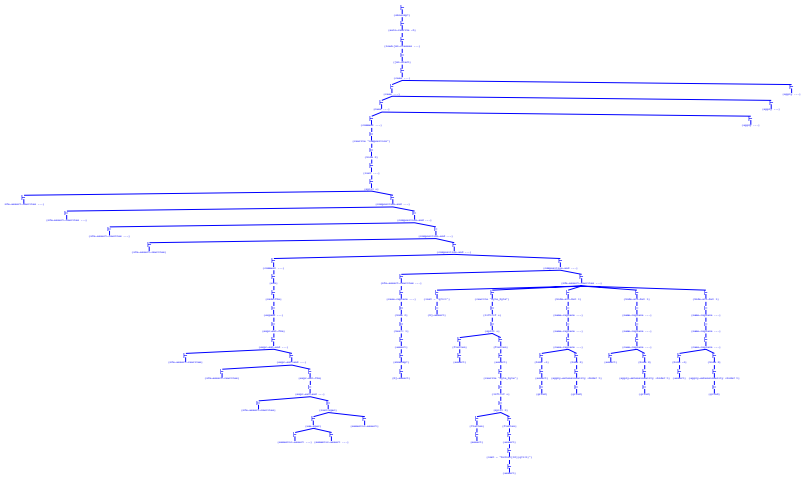

Figure 4.5: Proof tree for the `respond` method.

verification technology and do therefore not focus on the implementation of Java Card applets. The closest work to this chapter we could find is [HRS02]. The applet they develop seems to have been verified in a dynamic logic which abstracts away from certain Java Card details, though.

4.5 Conclusions

In essence, what we have verified is a correspondence between the logical control flow and the flow of the actual Java implementation. The explicit representation of the applet's state makes formulating global properties much easier when writing the JML specification. Usually control flow analysis involves a lot of data abstraction, see e.g., [GD00, HGSC04] for Java Card-related examples. In contrast, our approach uses no data abstraction at all. This ensures that methods are called in an appropriate order, obeying security sensitive restrictions—such as the upper limit on the number of outstanding challenge-responses. Furthermore, it is possible to find actual implementation errors which can prove to be security critical. Low-level implementation issues, such as overflow of data types and unwanted exceptions, simultaneously with control flow properties can thus be detected.

Chapter 5

Specification and verification of non-interference in JML

In the previous chapter we have seen that JML can be used to specify control flow related security properties for JAVA CARD applets. Many other security properties for JAVA CARD applets have been specified and verified in JML, such as the absence of `ISOExceptions` at the top-level [JMR04] or well-formedness of transactions [PBB+04]. All these security properties have in common that they are relatively easy to express for individual methods in JML. Of course, getting the specifications correct if one has to deal with all the low-level details of Java still forms a challenge. However, some program wide security properties such as authentication, confidentiality or integrity are far harder to express in JML.

The focus of this chapter is the specification of confidentiality in JML. Confidentiality is an important security property that is notoriously hard to enforce in computer programs. It can be formalized using the notion of non-interference, first introduced by Goguen and Meseguer [GM82] in the early eighties. In this chapter we primarily focus on *termination-insensitive non-interference*, i.e., we look at all the possible runs of a program that terminate normally and disregard other termination forms such as non-termination (if the program hangs) and abrupt termination (e.g., if the program terminations with an exception).

In Section 2.5 we confidentiality (as non-interference) is discussed in more detail.

Confidentiality can be expressed in JML using the notion of a so-called *specification pattern* for JML. This specification pattern is based on work by Joshi and Leino [JL00]. As far as we know, formalizing confidentiality in this way in JML is novel. Others have either extended JML to be able to specify confidentiality [DFM05] or formalized it directly in a logic of a particular tool, e.g., in dynamic logic [DHS05]. Both these approaches can no longer use the whole range of JML tools [BCC+05] for verification, which is one of *the* attractive features of JML.

The current chapter also serves as a bridge between Chapter 3 and Chapter 4 on the one side and Chapter 6 through 8, on the other side. The former use JML to specify and verify behavioral specifications while the latter deal with dedicated methods for proving non-interference properties. This chapter combines these two and uses JML to specify and

verify non-interference properties for Java programs.

The remainder of this chapter is organized as follows. Section 5.1 introduces the specification pattern for confidentiality. Section 5.2 applies the specification pattern to some examples. Section 5.3 explores the possibility of specifying stronger notions of non-interference in JML. Section 5.4 discusses related work and the chapter ends with conclusions in Section 5.5.

5.1 A specification pattern for confidentiality

In the examples in this chapter we use the simple security lattice $\Sigma = \{\mathsf{High}, \mathsf{Low}\}$ with $\mathsf{Low} \sqsubseteq \mathsf{High}$ from Chapter 2. A secure information flow policy is then given by the function $\mathsf{Sif} : \mathsf{Var} \to \Sigma$ which maps variables to security levels in the simple security lattice. We will abuse notation by identifying security levels and the sets of variables corresponding to those levels, i.e. $\mathsf{High} = \{v \in \mathsf{Var} \mid \mathsf{Sif}(v) = \mathsf{High}\}$ and $\mathsf{Low} = \mathsf{Var} \setminus \mathsf{High}$.

Confidentiality can be formalized using the notion of non-interference [GM82]:

> *Noninterference for programs essentially means that a variation of confidential (high) input does not cause a variation of public (low) output.* [SM03]

In later chapters we will define non-interference more formally, for now this informal definition suffices for our purpose. Notice that confidentiality involves a relation between the pre- and post-state. In JML it is possible to express such relations using the keyword `\old`. If used in a postcondition, variables encapsulated by `\old` are evaluated in the precondition. This makes it possible to use a formulation of confidentiality in JML. This formalization of confidentiality constitutes a form of non-interference that is called *termination-insensitive non-interference*. Confidentiality is only guaranteed for all normally terminating runs of a program, non-termination and abrupt termination (i.e., via an exception) is ignored.

Pattern 1 (specification pattern for confidentiality)**.**

Confidentiality can be expressed in JML as a specification pattern, consisting of clauses of the form:

$$\texttt{ensures low} \;==\; \texttt{\textbackslash old}(\chi_{\mathsf{Low}});$$

For each $\texttt{low} \in \mathsf{Low}$ *the expression* χ_{Low} *should not contain any fields* $\texttt{high} \in \mathsf{High}$*.*

By proving for a Java method that *all* fields $\texttt{low} \in \mathsf{Low}$ in the post-state are independent of the values of fields $\texttt{high} \in \mathsf{High}$ in the pre-state, we have proved confidentiality for that Java method[41]. The meta-expression `low == \old(`χ_{Low}`)` is called a *specification pattern* for confidentiality.

[41] Of course the specification pattern has to be meaningful, e.g., `low == \old((high==1)? low=0: low=0)` does not suggest that secret information is leaked since the pattern is equivalent to the simpler one `low == \old(0)` for which it is clear that it does not break confidentiality.

Integrity, as the formal dual of confidentiality, can easily be expressed using a similar specification pattern:

Pattern 2 (specification pattern for integrity)**.**

$$\texttt{ensures high} \quad == \quad \texttt{\textbackslash old}(\chi_{\mathsf{High}});$$

For each `high` $\in$ High *the expression* χ_{High} *should not contain any fields* `low` $\in$ Low*.*

Pattern 2 expresses that all high fields are independent of low fields. Thus low fields can not alter high fields, thereby ensuring integrity for all high fields in High. This chapter focuses on proving confidentiality, but all techniques described in the sequel are equally applicable to integrity.

The specification pattern for confidentiality can be used to prove one of the most common –and weakest– forms of non-interference know as *termination-insensitive* non-interference. As mentioned before, it can be understood to mean that the non-interference property is only specified if the program terminates normally. If the program terminates with an exception or does not terminate at all (e.g., via a non-terminating loop) the non-interference property is not guaranteed. We can easily strengthen the non-interference property somewhat by also using Pattern 1 inside JML's `signals` clause:

Pattern 3 (extended specification patterns for confidentiality)**.**

Confidentiality can be expressed in JML as two specification patterns:

$$\begin{array}{lll} \texttt{ensures low} & == & \texttt{\textbackslash old}(\chi_{\mathsf{Low}}); \\ \texttt{signals}\ (\epsilon)\ \texttt{low} & == & \texttt{\textbackslash old}(\psi_{\mathsf{Low}}); \end{array}$$

For each `low` $\in$ Low *and exception types* ϵ *the expressions* χ_{Low} *and* ψ_{Low} *should not contain any fields* `high` $\in$ High*.*

In this case non-interference is specified for a method if the method either terminates normally or with an exception ϵ[42]. However, the termination behavior of the method itself can also leak information or information can be leaked in case another exception than the ϵ listed is thrown.

In case the termination behavior itself and the possible termination modes of a program cannot leak any information we obtain the stronger form of non-interference called *termination-sensitive* non-interference (see Chapter 7). Section 5.3 discusses how termination sensitive non-interference can be expressed in JML.

Still stronger forms of non-interference also prevent the use of so-called covert channels [Lam73], such as timing [Aga00a, BRW06], resource consumption or caching, to leak information (see Chapter 8). We will not discuss this matter further here.

In the next section we show how the specification pattern for confidentiality can be applied to an example.

[42] For the most sensitive notion of confidentiality one can take for ϵ `java.lang.Exception`, thereby covering all possibly thrown exceptions.

5.2 Applying the specification pattern

In this section we will show a couple of examples of how the specification pattern for confidentiality can be applied.

5.2.1 A first example

This first example show a Java method `test()` that has the non-interference property:

Java

```
int high,low; // high:H, low:L

  /*@ normal_behavior
    @     requires true;
    @   assignable low;
    @      ensures low == \old(0);
    @*/
  void test() {
      low = high;
      low = low - high;
  }
```

The post condition of this method uses the specification pattern for confidentiality. Proving the correctness of this specification with one of JML's dedicated tools [BCC+05] is straightforward. An since the specification pattern is used *for all low fields* and *no high field occurs inside* `\old` we can conclude that the method is indeed non-interfering and thus maintains confidentiality.

5.2.2 An example involving method calls

The example described below contains two methods: `int decrementhigh(int i)` and `void m()` (the latter calls `decrementhigh`). There are only two fields, `high` and `low` which have security level High and Low respectively.

Java

```
int high,low;  // high:H, low:L

/*@ normal_behavior
  @   requires true;
  @ assignable high;
  @     ensures \result == \old(i) && high == \old(high - 1);
  @*/
int decrementhigh(int i){
    high = high -1;
    return i;
}

/*@ normal_behavior
  @   requires true;
  @ assignable low,high;
  @    ensures low == \old(high) && ensures high == \old(high - 1);
  @*/
void m(){
    low = decrementhigh(high);
}
```

Whether method `decrementhigh` is confidential depends on the parameter `i`. In case `i` is a low field the method does not leak information, otherwise it leaks information of a high field via its return value (which is bound to the special JML variable `\result`. Thus, since `high` is passed along as a parameter to method `decrementhigh` there is a dependency between field `low` and the value of `high` in the pre-state –as indicated by the specification pattern in the `ensures` clause– and method `m()` leaks secret information.

5.2.3 An example with a loop

The next example illustrates JML specification pattern for confidentiality on a method with a loop.

Java

```
int high,low; // high:H low:L


/*@ normal_behavior
  @   requires high > 0;
  @ assignable low,high;
  @    ensures low == \old(high);
  @*/
```

```
void m() {
  low=0;
  /*@ maintaining high + low = \old(high);
    @  decreasing high;
    @*/
  while (high > 0){
    high--;
    low++; }
}
```

Method `m()` above leaks information from `high` to `low` as is indicated by the specification pattern in the `ensures` clause. Notice that in order to prove the correctness of this specification one needs to specify the loop invariant (as given by the JML keyword `maintaining`) and the loop variant (denoted by the JML keyword `decreasing`).

Next we show the application of the specification pattern for confidentiality on a larger example involving a cash register.

5.2.4 A cash register

Most modern cash registers have some sort of logging mechanism to check that no money from the register has disappeared. As an example we have written a small Java program that implements such a secure logging mechanism. The idea is that all transaction are stored in an integer-array in such a way that both the total amount in the register and each individual transaction is logged. The first index (I) of the array will be set to a random value[43] and each consecutive index will contain the value of the previous index plus the new amount added to the register. When the last index is used the register will be locked and no additional money can be added to it until the log is dumped and the whole process can start form the beginning. Figure 5.1 gives a graphical representation of the log.

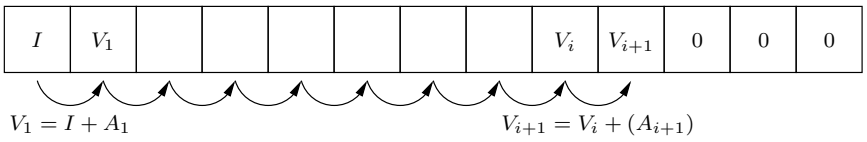

Figure 5.1: The log of the cash register.

Entries that are not used yet have the default value zero. Since all values depend on the previous one, fraud with the log involves changing all values of the log. Moreover the

[43] This is also the reason why we require that `counter > 0` in the specification of method `int add` below.

initial random value at the first index ensures that someone who can read one index has no idea what the total amount of money in the cash is[44], which constitutes the main security objective of the logging mechanism.

One method from the logging mechanism is discussed in detail here, the method `public int add(int amount, int money)` that implements how money is added to the register. The parameter `amount` is the amount that has to be payed by the customer and the `money` parameter is the actual amount of money given by the customer. The return value is the change that has to be returned to the customer. Both the parameters and the return value have security level Low. The public fields `counter` and `size` will also be treated as a variables with security level Low and the (private) reference field `payments` including its array entries has a High security level. The security levels are also indicated in the code in (ordinary) Java comments.

The full program listing can be found in Appendix A.2.

Java

```
class Logger{

     public int counter, size;                    // Low
     private /*@ spec_public @*/ int[] payments;  // High

    /*@ invariant
      @   size > 0 && counter > 0 && counter <= size &&
      @   payments != null && payments.length == size &&
      @   (\forall int i; i > counter && i < size ==> payments[i] == 0) */

    /*@ constraint
      @   size == \old(size) && payments == \old(payments);
      @*/

    /*@
      @ behavior
      @   requires counter > 0;
      @ assignable counter, payments[counter];
      @    ensures amount > 0 && \old(counter) < size &&
      @              counter == \old(counter) + 1 &&
      @              payments[counter -1] ==
      &                       \old(payments[counter - 1] + amount)  &&
```

[44] A more involved implementation of such a secure logging mechanism should use cryptographic hash functions like Sha-1 and MD5. In that case instead of simply adding the next value to the array one should add the hash of the value concatenated with the entry at the previous array index (which is then either a previous hash or the first random entry of the array. We have chosen not to use such functions here since they add some complexity to the Java code in the form a API-calls but do not add any fundamental insights.

```
    @              payments[counter] ==  payments[\old(counter)]  &&
    @              \result == \old(money) - \old(amount) &&
    @              \result >= 0;
    @ signals(NotEnoughMoneyException) \old(amount) > \old(money) &&
    @                                  payments[counter - 1] ==
    @                                  \old(payments[counter -1]) &&
    @                                  counter == \old(counter);
    @ signals(LogFullException) (\old(counter) >= \old(size)  ||
    @                           amount <= 0) &&
    @                           counter == \old(counter) &&
    @                           payments[counter -1] ==
    @                               \old(payments[counter -1]);
    @ signals(OverflowException)
    @   amount > \old(Integer.MAX_VALUE - payments[counter - 1] &&
    @   counter == \old(counter) &&
    @   payments[counter] == \old(payments[counter]);
    @*/
   public /* Low */ int add(/* Low */ int amount,
                                      /* Low */ int money) throws
                                      NotEnoughMoneyException,
                                      LogFullException,
                                      OverflowException{
     if (amount > money)
        NotEnoughMoneyException.throwIt();
     if (amount > 0 && counter < size ) {
        if(payments[counter-1] > Integer.MAX_VALUE - amount)
            OverflowException.throwIt();
        payments[counter] =
            payments[counter-1] + amount;
        counter++; }
      else LogFullException.throwIt();
     return money - amount;
   }
}
```

There are three fields associated with this class: the private array `payments` (the log), a public `counter` that keeps track of where the current index in the private array `payments` is, and a public field `size` that equals the length of this array. The specification has a class invariant which states some obvious properties of the array and asserts that indices that are not used yet have the default value zero. Recall from Section 2.3 that the constraint gives a relation between the pre- and post-state and says that, once created, the length and the reference to the array stay the same. The method specification should be self-explanatory, but notice that Pattern 3 is used in the specification –partly implicit via the constraint.

Three exceptions can be thrown: A `NotEnoughMoneyException` occurs if the `money`

given by the customer is less than the `amount` that has to be paid. An `OverflowException` will be thrown if the value stored at the current index is bigger than `Integer.MAX_VALUE`. Note that this is explicitly implemented in the method, which is necessary because overflow is silent in Java. Finally, a `LogFullException` will be thrown if the log is full, i.e., if `\old(counter)==size` (or if `amount` is below 0 which shouldn't be possible). The post-conditions for each exception can be found in the corresponding `signals` clauses. Both the `NotEnoughMoneyException` and the `LogFullException` are trivial. The specification of the method `add` above has been proved for the given implementation, using the LOOP tool. The examples in this section illustrate that the specification pattern for confidentiality can be used on 'real' Java code and that it can identify potential information leaks.

When the `OverflowException` is thrown the method leaks information: part of this post-condition reads:

```
amount > \old(Integer.MAX_VALUE - payments[counter-1])
```

which leaks partial information about the value stored at the previous index, i.e., at `\old(payments[counter-1])`. Notice that this is not expressible using the specification pattern for confidentiality, since the binary relation used here is a 'bigger then' (`>`) instead of an equality (`==`) relation.

This shows that our method is not complete, i.e., we cannot construct a specification pattern for confidentiality in all cases. Another example of incompleteness occurs when a program performs a complex computation on a low input that sometimes fails to terminate. It might not always be the case that we can construct a specification pattern for confidentiality in such cases. Another disadvantage of this approach is that writing such JML specification patterns for confidentiality is a tedious and error-prone task. Moreover, it does not scale very well. In a preliminary paper the author and Martijn Oostdijk have investigated whether it is possible to automatically generate specification patterns for confidentiality from a secure information flow policy [WO05]. The first results are encouraging, but there is still more research to be done before this question can be answered fully.

However, we believe that the proposed methodology can still be useful in certain cases, especially when one has already constructed (and verified) JML specifications that express other functional or security properties for a particular program. Since in this case one can use one specification language –JML– for the entire verification process.

In cases where confidentiality is of the utmost importance the specification patterns for confidentiality are not enough. In the next chapters we propose methods that ensure a more justified believe in the confidentiality of verified programs.

5.3 Towards termination sensitive non-interference

This chapter mainly focuses on *termination-insensitive* non-interference, meaning that the non-interference property (confidentiality, integrity) is only guaranteed if the analyzed

program terminates normally. If the program terminates with an exception or does not terminate at all (hangs), non-interference is no longer assured.

We have already seen that the specification pattern for confidentiality can easily be used in `signals` clauses. When used in this way, it expresses that when an exception of a certain type is thrown the non-interference property holds. Similarly, it is possible to use the specification pattern for confidentiality in JML's `diverges` clause. Such a specification pattern states for which *precondition* a non-interference result holds provided the method does not terminate.

Termination behavior itself can also leak information, as illustrated by the next code fragment:

Java

```
boolean high; // high:H

public void m(){
     if(high) throw new Exception();
}
```

Assuming the termination behavior of a program is observable, the value of field `high` is leaked. Such cases of information leakage are not covered by the specification pattern for confidentiality. Termination sensitive non-interference can easily be specified in JML using the `normal_behavior` and `exceptional_behavior` keywords, which specify that the method *must* terminate normally or with an exception respectively. In effect this makes termination sensitive and insensitive termination coincide, because only one termination mode (per method) is allowed.

More generally, termination sensitive non-interference in JML can be specified using a JML `ghost` field, as illustrated below:

Java

```
  boolean high;  // high: H;


//@ public ghost int _tmode;


/*@ invariant _tmode == 0 || // normal termination
              _tmode == 1    // exceptional termination
  */

 /*@ behavior
   @   diverges false;
   @   requires true;
```

```
  @ assignable \nothing
  @    ensures \old(high) ==> _tmode == 1 &&
  @           !\old(high) ==> _tmode == 0;
  @    signals (Exception e)
  @            \old(high) ==> _tmode == 1 &&
  @           !\old(high) ==> _tmode == 0;
  @*/
public void m() throws Exception{
    if (high){
        //@ set _tmode = 1;
        throw new Exception();
    }
    //@ set _tmode = 0;
}
```

First of all note that we restrict ourselves to programs that terminate either normally or via an exception. We will always require that the `diverges` clause is `false` if we want to prove termination-sensitive non-interference, thereby ensuring that non-termination of a method cannot leak information.

By encoding the other termination behaviors of a method in the ghost field we can specify that the value of field `high` in the pre-state determines the termination behavior of the method. Thus information is leaked via the termination behavior of method `m()`. Notice that the postconditions in the `ensures` and `signals` clauses are the same and that the invariant is used to guarantee that there are only two termination modes allowed.

In cases where a method is called, e.g., `return someMethod()`, it is the responsibility of the called method (i.e., `someMethod()`) to ensure that the ghost variable is set to the correct termination mode. This ensures that the method is modular and can be used for complete programs.

The specification pattern for confidentiality is no longer used. A specification that has the ambition to specify termination sensitive non-interference in JML should use the specification pattern for confidentiality *and* should specify how the value of each High field in pre-state influences termination behavior of a method.

5.4 Related work

In the context of the *SecSafe* project [Sec] several security properties which are relevant for Java Card applets have been identified [MM01]. They concern amongst others the absence of certain exception types at the top-level, atomicity of updates, no unwanted overflow, only memory allocation during the install phase of the applet and conditional execution points. All these points can either be expressed directly in JML or in the underlying semantic model used by one of the JML tools, as is shown by several researchers [Mos05b, PBB+04, JMR04, JOW04, HOP04]. Expressing non-interference properties such as confidentiality and integrity directly in JML has, to the best of our knowledge, not been done before.

The work of Dufay, Felty and Matwin [DFM05] is probably most related to ours. They add keywords to JML that express confidentiality and have modified the Krakatoa tool [CDF+04] in order to prove non-interference properties for this extended version of JML. The actual non-interference proves are then preformed using the approach described in [BDR04]. The main idea is to compose the program with itself and execute it again *in a different part of memory*. Then a non-interference relation has to hold. This approach has two weak points: first the program needs to be analyzed *twice*, which leads to a considerable overhead (see Chapter 7 for a solution to this problem). Second, as we will show in Chapter 7, it impossible to show a termination-sensitive non-interference result using this approach. Another disadvantages of Dufay *et. al.* 's approach is that only the (modified) Krakatoa tool can be used to prove their extended JML annotations. Other JML tools first need to be altered in order to be used to prove non-interference.

The group behind the KeY-tool [ABB+04] uses a similar approach to ours to prove confidentiality [DHS05]. The main difference is that they do not use a separate specification language but express confidentiality directly in the dynamic logic which they use to reason (interactively) about sequential Java (card) programs.

The Spark programming language [Bar03] has been developed for implementing highly reliable code. The language is basically a subset of Ada[Geh83] with native support for (JML like) annotations. The Spark annotation `derives l from l1,l2,l3` expresses that `l` depends on `l1,l2` and `l3`. In a way this is a minimum annotation for expressing dependencies between fields.

The notion of a specification pattern originated in the work of Dwyer *et. al.* [DAC99]. They noticed that patterns emerge when specifying temporal properties for concurrent systems. As far as we know, specification patterns for JML have not been proposed before.

5.5 Conclusions

We have shown that is possible to apply JML for specifying non-interference properties like confidentiality and integrity using specification patterns for JML. The main advantage of this approach is that one specification language can be used to express a wide range of functional and security properties for Java programs. The main disadvantages are that we cannot construct a specification pattern in all possible cases and the approach does not scale[45]. In the next chapters we therefore look at dedicated methods for proving non-interference properties for Java programs.

[45] We suspect that the approaches proposed in [DFM05] and [DHS05] have similar (if not worse) scalability issues.

Chapter 6

Statically checking termination-insensitive non-interference

In the previous chapter the starting point for specifying and proving non-interference properties was the specification language JML. The main advantage of using JML is its *generality*, which makes it possible to specify and prove other interesting safety and/or security properties. However, this advantage comes at a price. If proving non-interference properties is the sole concern, then the whole process of proving these in JML can be shortened and simplified considerably by a *dedicated* approach that focuses only on proving non-interference properties. The current chapter, based on [JPW05], proposes such a dedicated approach.

Most dedicated methods for statically checking non-interference are based on type-checking, going back to the influential work of Volpano *et. al.* [VSI96, VS97a]. Typically such type-checking methods consider a subset of (the sequential part of) languages like ML [PS03], Java [BN02, Str03, Mye99], Java bytecode [BRB05] or special languages tailored for security such as SLam [HR98]. The object of our study is a simple programming language with side-effects in expressions and the usual programming constructs like composition, assignment, choice and repetition, i.e., WHILE. We will indicate how to extend our results for this simple language to a Java-like language (see Section 6.5).

Type-based approaches have the advantage that they are very easy to employ. A user only has to provide a secure information flow policy and the type-checking algorithm automatically verifies if the program satisfies this policy. However, type-based approaches are also overly restrictive. Many secure programs are rejected as being possibly insecure.

Joshi and Leino [JL00] were the first to show that even in very simple programming languages one can construct provably secure programs that are deemed insecure by type-based approaches. The main idea for constructing such a program is the observation that type-based approaches will stop at the first point where a non-interference property –we again take confidentiality as an example– is broken. Confidentiality is a property that holds for a *complete* program. This means in particular that confidentiality can be *temporarily*

broken, as long as it is *restored* before the program terminates[46]. Interactive techniques for checking non-interference properties, such as those proposed by [JL00, BDR04, CHH02] (also see Chapter 7), are able to handle such temporary breaches of confidentiality. Of course, the main disadvantage of such approaches is that they require interactive reasoning by a specialist.

In this chapter a new approach for statically checking confidentiality is introduced that combines the advantages of type-based approaches and minimizes its disadvantages. It is based on abstract interpretation [Cou96] and allows local breaches of confidentiality. Furthermore, the approach is sound (with respect to some standard program semantics) and has the same level of automation as type-based approaches. The entire framework –including all correctness proofs– has been developed entirely inside the theorem prover PVS [PVS, ORSH95].

Our approach uses a *dynamic labeling function* to track the security levels of program variables. Consider the program fragment $l := h \; ; \; l := 2$, where l has initial security level Low, h has initial level High and $\mathsf{Low} \sqsubseteq \mathsf{High}$. Example 1 below (informally) illustrates how a local breach of confidentiality can be restored. It also shows how the security level of each variable changes during execution of this program fragment.

Example 1.

$$\left\{ \begin{array}{lcl} l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array} \right\} \quad l := h \quad \left\{ \begin{array}{lcl} l & : & \mathsf{High} \\ h & : & \mathsf{High} \end{array} \right\} \quad l := 2 \quad \left\{ \begin{array}{lcl} l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array} \right\}$$

Informally, the assignment $l := h$ breaks confidentiality since a variable with security level High is assigned to a variable of level Low which results in secret information flowing to public (low) variables. However, the assignment $l := 2$ restores confidentiality. Every completed execution of this program will have the same result: the low variable l will have value 2. Thus no secret information is leaked to the low variable.

This use of dynamic labels ensures that it is possible to analyze confidentiality for entire programs. Our main result is that if all dynamic labels for a program are *non-increasing* –which can be checked automatically– then the program maintains confidentiality. In Section 6.3 we will give a formal justification for this result.

In this chapter, like in Chapter 5, we again study termination-insensitive non-interference. Thus, we only assure that a non-interference property holds if the program terminates. Non-interference for all other termination modes –specifically non-termination– is not ensured. Stronger forms of non-interference are studied in Chapter 7 and Chapter 8.

In order to deal with implicit information flow a special variable lenv, called the environment level, is used which stores the security level of 'the context'. An example of implicit information flow is given in the program fragment $\texttt{if}(h > 2)\ \texttt{then}\ l := 0$, where h has

[46] This only works for sequential programs. In a multi-threaded execution model information can be shared amongst multiple threads. Thus, if at some point confidentiality is temporarily broken, other threads can leak information to the outside.

security level High, l level Low and the environment level has initial level Low. Example 2 shows the labels of l and h, and lenv at each point during the execution of this program:

Example 2.

$$\left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{Low} \\ l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\} \ \texttt{if}(h > 2) \ \left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{High} \\ l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\} \ l := 0 \ \left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{High} \\ l & : & \mathsf{High} \\ h & : & \mathsf{High} \end{array}\right\}$$

Under normal conditions the assignment $l := 0$ does not break confidentiality. However, the assignment in this particular context breaks confidentiality because it is carried out under a high guard, thereby implicitly leaking information from a variable with security level High to one with security level Low. The environment variable always has the same security level as the *highest* security level of the conditionals that guard the context. The variable l in Example 2 obtains the same security level as the *maximum* of the security level of 0, which is Low, since it is a constant, and the environment level, which is High since the highest (and only) security level of the conditionals in this context is $h > 2$ which has level High. The new label of the variable l is thus higher than its old label and thus this code fragment breaks confidentiality.

The remainder of this chapter is organized as follows: the next section introduces some notation and basic concepts. In Section 6.2 it is shown how the dynamic labeling transition functions that make up our approach are defined. A formal proof that every program that only has non-increasing labels is indeed confidential (soundness) together with an approach for statically checking confidentiality appears in Section 6.3. Section 6.4 gives some example programs that illustrate our approach. Section 6.5 discusses how our approach can be extended to a more complex object-oriented programming language, such as Java. Section 6.6 discusses related work. The chapter ends with conclusions and suggestions for future work.

6.1 Preliminaries

In this and later sections we abstract away from the specific PVS syntax and formalization. Instead a more logical/mathematical notation is used to present our work as generally as possible.

We assume a finite lattice with carrier type L, bottom element $\bot$, top element $\top$, partial order relation $\leq$ and join $\sqcup$. This lattice is used to represent the security levels. In this chapter we will, in examples, only use the simple security lattice from Definition 2 with values High and Low. The definitions and results however hold for general finite lattices. A special variable conf:L, called the *confidentiality level*, is used to split the lattice L into two parts:

$$\begin{array}{rcl} \Uparrow\mathsf{conf} & = & \{b : \mathsf{L} \mid \mathsf{conf} \leq b\} \\ \Downarrow\mathsf{conf} & = & \{b : \mathsf{L} \mid \mathsf{conf} \not\leq b\} \end{array}$$

The environment level $\mathsf{lenv{:}L}$ has initial value $\bot$ and can only be changed when a conditional statement is analyzed. The formal dual of confidentiality, which is a form of integrity [Bib77], can be analyzed by 'flipping' the lattice. In this case we fix all the high variables and vary the low ones and prove that all high output variables are independent of low input variables. Thus ensuring that public inputs cannot alter the secret (high) data, which enforces integrity.

A location in memory is represented by type Loc. The memory, written as M, consists of mappings from Loc to values. A labeling function $\mathsf{lab} : \mathsf{Loc} \to \mathsf{L}$ then maps memory locations to the security levels given by L.

We consider the small imperative programming language WHILE from Definition 1 with statements and expressions (with side-effects). Statements in WHILE are programs. The semantics of a statement, which can either terminate or hang, has type $\mathsf{M} \to 1 + \mathsf{M}$, where $1 = \{*\}$ and $+$ is disjoint union. The semantics of an expression, which has an additional result value of type Out, has type $\mathsf{M} \to 1 + (\mathsf{M} \times \mathsf{Out})$, where Out can be int, bool etc. We use a denotational semantics here since the work in this chapter has been formalized on top of (a subpart) of the denotational LOOP semantics in PVS. For our purpose any standard denotational semantics suffices (see e.g., [NN92, Sch86]). In the sequel $[\![\ ]\!]$ refers to this semantics.

Next we define an indistinguishability relation $\mathcal{I}$ between memory states, which is parametrized by a labeling function and confidentiality level. This relation is used for defining termination-insensitive non-interference.

Definition 3 (Indistinguishability)**.** The indistinguishability relation $\mathcal{I}$ is defined as

$$\mathcal{I}(\mathsf{conf}, \mathsf{lab}) \subseteq \mathsf{M} \times \mathsf{M} \ = \ \{\ (x, y) \in \mathsf{M} \times \mathsf{M} \mid \forall l : \mathsf{Loc}.\ \mathsf{conf} \not\leq \mathsf{lab}(l) \Rightarrow x(l) = y(l)\ \}$$

Thus $\mathcal{I}(\mathsf{conf}, \mathsf{lab}) \ni (x, y)$ means that memory locations in x and y may only differ for variables in the part of the security lattice given by $\Uparrow \mathsf{conf}$[47]. A semantic definition of confidentiality –as termination-insensitive non-interference– is then given in terms of this indistinguishablity relation.

Definition 4 (Confidentiality as termination-insensitive non-interference)**.** Let p be a program written in WHILE. Then confidentiality for p is defined as

$$\begin{array}{l}\mathsf{Confidential}(\mathsf{p}, \mathsf{lab}) = \\ \quad \forall x, y : \mathsf{M}. \forall \mathsf{conf} : \mathsf{L}. \\ \qquad \mathcal{I}(\mathsf{conf}, \mathsf{lab}) \ni (x, y) \ \wedge\ [\![\mathsf{p}]\!](x) \neq * \ \wedge\ [\![\mathsf{p}]\!](y) \neq * \ \Rightarrow\ \mathcal{I}(\mathsf{conf}, \mathsf{lab}) \ni ([\![\mathsf{p}]\!](x), [\![\mathsf{p}]\!](y))\end{array}$$

[47] Our definition of indistinguishability ($\mathcal{I}$) is not the one usually found in the literature. The standard definition for indistinguishability (which we shall call $\mathcal{I}^*$) is: $\forall l.\mathsf{Loc}.\ \mathsf{lab}(l) \leq \mathsf{conf} \Rightarrow x(l) = y(l)$. We have chosen our own definition because it was more convenient in our modeling in PVS, but since we quantify over all possible security levels the obtained results are equivalent for the standard definition. In particular, our definition ($\mathcal{I}$) rules out flows *from* a level conf unless the flows are to a level above conf, while the standard definition ($\mathcal{I}^*$) rules out flows *to* a level conf unless the flow comes form below. For a two point lattice and a fixed level $\mathcal{I}(\mathsf{High}) = \mathcal{I}^*(\mathsf{Low})$.

Confidentiality, as defined in Definition 4, states that if all memory locations in $\Downarrow$conf are equal for (memory) states x and y, then these same variables should again be equal in the new states obtained by executing program p. This guarantees that $\Downarrow$conf variables are independent of $\Uparrow$conf variables.

Definition 4 clearly concerns termination-insensitive non-interference since it is required that program p does not hang ($[\![\mathsf{p}]\!](x) \neq * \ \wedge \ [\![\mathsf{p}]\!](y) \neq *$), i.e., terminates normally. Furthermore, notice that by quantifying over conf and splitting the lattice in two parts $\Uparrow$ conf and $\Downarrow$ conf, we can reason about all secure information flow policies defined by security levels in the lattice at once. Finally, notice that the labeling function used in the definition of confidentiality is *global* in the sense that both before and after execution of p the same labeling lab is used to distinguish variables in $\Uparrow$conf from those in $\Downarrow$conf.

6.2 Labeling transition functions

We define a function that, given a language construct in WHILE, an initial labeling and an environment level, yields a labeling after execution of the statement or expression. This function gives an abstract interpretation of the statement or expression in terms of modification of security levels. We have two labeling transition functions, one for statements, called labStat, and one for expressions, named labExpr. The function labExpr has an additional result, namely the level of the result of the expression.

The signature for labExpr and labStat, where s is a statement and e an expression, is then given by:

$$\mathsf{labStat(s)} \ : \ ((\mathsf{Loc} \to \mathsf{L}) \times \mathsf{L}) \ \to \ (\mathsf{Loc} \to \mathsf{L})$$

$$\mathsf{labExpr(e)} \ : \ ((\mathsf{Loc} \to \mathsf{L}) \times \mathsf{L}) \ \to \ ((\mathsf{Loc} \to \mathsf{L}) \times \mathsf{L})$$

We can now define the labeling transition function labStat for all statements in WHILE.

Definition 5 (labStat)**.** Let lenv be the environment level. The labeling transition function labStat is then defined as:

$$\begin{aligned}
&\mathsf{labStat}(v := \mathsf{e})(\mathsf{lab}, \mathsf{lenv}) \ = \\
&\qquad \text{let } (\mathsf{lab}', \mathsf{lres}) = \mathsf{labExpr(e)}(\mathsf{lab}, \mathsf{lenv}) \\
&\qquad \text{in } \mathsf{lab}'[(\mathsf{lres} \ \sqcup \ \mathsf{lenv}) \ / \ \mathsf{lab}'(v)]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{labStat}(\mathsf{s1}; \mathsf{s2})(\mathsf{lab}, \mathsf{lenv}) = \\
&\qquad \mathsf{labStat(s2)}(\mathsf{labStat(s1)}(\mathsf{lab}, \mathsf{lenv}), \mathsf{lenv})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{labStat}(\texttt{if-then}(\mathsf{b})(\mathsf{s}))(\mathsf{lab}, \mathsf{lenv}) = \\
&\qquad \text{let} \\
&\qquad\quad (\mathsf{lab}', \mathsf{lres}) \ = \mathsf{labExpr(b)}(\mathsf{lab}, \mathsf{lenv}), \\
&\qquad\quad \mathsf{lenv}' \qquad = \mathsf{lres} \sqcup \mathsf{lenv} \\
&\qquad \text{in } \mathsf{labStat(s)}(\mathsf{lab}', \mathsf{lenv}') \ \sqcup \ \mathsf{lab}'
\end{aligned}$$

$$
\begin{array}{l}
\mathsf{labStat}(\texttt{if-then-else}(\mathsf{b})(\mathsf{s}_1)(\mathsf{s}_2))(\mathsf{lab},\mathsf{lenv}) = \\
\quad \text{let} \\
\quad\quad (\mathsf{lab}',\mathsf{lres}) \;\; = \mathsf{labExpr}(\mathsf{b})(\mathsf{lab},\mathsf{lenv}), \\
\quad\quad \mathsf{lenv}' \;\; = \mathsf{lres} \sqcup \mathsf{lenv} \\
\quad \text{in } \mathsf{labStat}(\mathsf{s}_1)(\mathsf{lab}',\mathsf{lenv}') \;\sqcup\; \mathsf{labStat}(\mathsf{s}_2)(\mathsf{lab}',\mathsf{lenv}') \\
\\
\mathsf{labStat}(\texttt{while}(\mathsf{b})(\mathsf{s}))(\mathsf{lab},\mathsf{lenv}) = \\
\quad \bigsqcup_i \mathsf{iterate}(\mathsf{b})(\mathsf{s})(\mathsf{lab},\mathsf{lenv})(i), \text{where} \\
\quad \mathsf{iterate}(\mathsf{b})(\mathsf{s})(\mathsf{lab},\mathsf{lenv})(n) = \\
\quad\quad \text{let} \\
\quad\quad\quad (\mathsf{lab}',\mathsf{lres}) \;\; = \mathsf{labExpr}(\mathsf{b})(\mathsf{lab},\mathsf{lenv}), \\
\quad\quad\quad \mathsf{lenv}' \;\; = \mathsf{lres} \sqcup \mathsf{lenv}, \\
\quad\quad\quad \mathsf{lab}'' \;\; = \mathsf{labStat}(\mathsf{s})(\mathsf{lab}',\mathsf{lenv}') \\
\quad\quad \text{in} \\
\quad\quad\quad \mathsf{IF} \;\; n = 0 \\
\quad\quad\quad \mathsf{THEN} \;\; \mathsf{lab}' \\
\quad\quad\quad \mathsf{ELSE} \;\; \mathsf{lab}' \sqcup \mathsf{iterate}(\mathsf{b})(\mathsf{s})(\mathsf{lab}'',\mathsf{lenv}')(n-1)
\end{array}
$$

where $n \in \mathbb{N}$, / is function update, and $\leq$ and $\sqcup$ are defined point-wise on labeling functions.

We will explain the rule for assignment in some detail here. To determine the new labeling function that is obtained by abstractly evaluating the statement $v := \mathsf{e}$ we first determine what the result is of evaluating the expression e (this is necessary since e can have a side-effect). This results in a new labeling function lab' and the label of the result of evaluating expression e which is lres. Now, to obtain the new labeling function we substitute the label of variable v in lab‘ with the maximum of the label lres or the environment level, which is denoted with lenv. Thus the environment level in the labeling function for $v := \mathsf{e}$ is used to give v the security level of the result of expressions e or, if we are working in the context of a higher conditional, the level of the environment.

In case a boolean expression b in a statement $\texttt{if-then}(\mathsf{b})(\mathsf{s})$ evaluates to false we need a rule that only updates the labeling function lab with the new labeling function obtained by executing b (we allow b to have side-effects). However, if b evaluates to true, then the statement s is also executed. In this case we (abstractly) evaluate $\mathsf{labStat}(\mathsf{s})$ with as arguments the labeling function obtained by evaluating b and as environment level the maximum of the old environment level lenv and the security level of the result of expression b. Thus for the actual dynamic labeling rule (as found in Definition 5 above) we take the point-wise maximum of the two new labeling functions as the new labeling function. This is necessary since we only have an abstract semantics (given by the labeling function $\mathsf{labStat}$) and thus do not know if b is true or not.

The idea behind the $\mathsf{labStat}$ part for the $\texttt{while}$ is that we calculate a least fixed point for the $\mathsf{iterate}$ function. Since we use a finite lattice, and the join ensures that the $\mathsf{iterate}$ function is increasing, such a fixed point always exists and is reachable in finitely many

iterations. Notice that we do not check (non)-termination of the loop; this should be proved by a semantic evaluation. The other parts of the definition of labStat should hopefully be self-explanatory.

The use of an abstract semantics forms the source of the lack of completeness of our formalization. The static framework is sound but not complete in the sense that some programs which *do* maintain confidentiality will be marked as possibly violating confidentiality. In Example 3 below we show a typical example of such a code fragment:

Example 3.

$$\left\{\begin{array}{lcl} l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\} \quad l := h \quad \left\{\begin{array}{lcl} l & : & \mathsf{High} \\ h & : & \mathsf{High} \end{array}\right\} \quad l := h - l \quad \left\{\begin{array}{lcl} l & : & \mathsf{High} \\ h & : & \mathsf{High} \end{array}\right\}$$

Our approach (and other static approaches) will mark this code as violating confidentiality even though the program does not leak information since variable l will have value zero after each evaluation of this program and is thus independent of the value of variable h. Since the problem of confidentiality is in general undecidable we have to choose between a decidable sound –but incomplete– method or an undecidable but sound *and* complete method. In Chapter 7 we propose a interactive framework for proving non-interference properties. When this approach is used it is possible to prove that the example above is non-interferent.

The function labExpr is defined in a similar fashion for all expressions in WHILE.

Definition 6 (labExpr)**.** Let lab be a labeling function and lenv the environment level, the labeling transition function labExpr is then defined as:

$$\mathsf{labExpr}(c)(\mathsf{lab},\mathsf{lenv}) = (\mathsf{lab},\bot)$$

$$\mathsf{labExpr}(v)(\mathsf{lab},\mathsf{lenv}) = (\mathsf{lab},\mathsf{lab}(v))$$

$$\begin{array}{l}\mathsf{labExpr}(\mathsf{e1\ op\ e2})(\mathsf{lab},\mathsf{lenv}) = \\ \quad \text{let } (\mathsf{lab}',\mathsf{lres}) = \mathsf{labExpr}(\mathsf{e1})(\mathsf{lab},\mathsf{lenv}), \\ \quad\quad (\mathsf{lab}'',\mathsf{lres}') = \mathsf{labExpr}(\mathsf{e2})(\mathsf{lab}',\mathsf{lenv}) \\ \quad \text{in } (\mathsf{lab}'',\mathsf{lres} \sqcup \mathsf{lres}')\end{array}$$

$$\begin{array}{l}\mathsf{labExpr}(v\mathsf{++})(\mathsf{lab},\mathsf{lenv}) \ = \\ \quad (\mathsf{lab}[(\mathsf{lab}(v) \ \sqcup \ \mathsf{lenv}) \ / \ \mathsf{lab}(v)], \mathsf{lab}(v) \sqcup \mathsf{lenv})\end{array}$$

$$\begin{array}{l}\mathsf{labExpr}(v := \mathsf{e})(\mathsf{lab},\mathsf{lenv}) \ = \\ \quad (\mathsf{labStat}(v := \mathsf{e})(\mathsf{lab},\mathsf{lenv}), \pi_2(\mathsf{labExpr}(\mathsf{e})(\mathsf{lab},\mathsf{lenv})))\end{array}$$

where π is projection and op is either $==$, $+$, $-$, $<$ or $>$.

6.3 Correctness of our approach

As we (informally) have seen in the introduction, confidentiality follows if we require that labels, for an entire program, may not increase. We call this property *non-increasingness*. Intuitively, this can be understood as follows: a variable with initial security level High that has level Low after execution of a program does not break confidentiality, since it is impossible to (directly) observe the original value of the high variable. However, a variable with initial security level Low that has level High after execution of a program *may* leak confidential information, since it gives information about the initial value of some higher variable. In this section we only give definitions and proofs for statements. They are analogous for expressions and do not add any new insights.

Definition 7 (NonIncreasing)**.** Let s be a statement and lab a labeling function. The predicate non-increasing on a statement s is then defined as:

$$\mathsf{Decreasing}(\mathsf{s},\mathsf{lab}) = \mathsf{labStat}(\mathsf{s})(\mathsf{lab},\bot) \leq \mathsf{lab}$$

where $\leq$ is point-wise ordering on labeling functions of type $\mathsf{Loc} \to \mathsf{L}$.

Notice that the environment level is initially set to $\bot$. Only when conditionals are analyzed can it change to a higher security level.

The labeling transition functions together with the non-increasing property determine if a program is non-interfering. In order to formally prove that our approach is sound –that is, that if our approach indicates that a program is non-interfering this is indeed the case– we first define a technical property, which we simply call Good.

Definition 8 (Goodness)**.** If s is a statement, then goodness of s is defined as

$$\begin{aligned}
&\mathsf{Good}(\mathsf{s}) \;=\\
&\quad \forall \mathsf{lab} : \mathsf{Loc} \to \mathsf{L}.\forall \mathsf{lenv} : \mathsf{L}.\\
&\qquad (\forall x, y : \mathsf{M}.\forall \mathsf{conf} : \mathsf{L}.\\
&\qquad\; [\![\mathsf{s}]\!](x) \neq * \wedge [\![\mathsf{s}]\!](y) \neq *\wedge\\
&\qquad\; (\mathcal{I}(\mathsf{conf},\mathsf{lab}) \ni (x,y) \Rightarrow \mathcal{I}(\mathsf{conf},\mathsf{labStat}(\mathsf{s})(\mathsf{lab},\mathsf{lenv})) \ni ([\![\mathsf{s}]\!](x),[\![\mathsf{s}]\!](y))))\\
\wedge\\
&\qquad (\forall z : \mathsf{M}.\forall b : \mathsf{Loc}.\\
&\qquad\; [\![\mathsf{s}]\!](z) \neq * \wedge z(b) \neq [\![\mathsf{s}]\!](z)(b) \Rightarrow \mathsf{lenv} \leq \mathsf{labStat}(\mathsf{s})(\mathsf{lab},\mathsf{lenv})(b))
\end{aligned}$$

First of all notice that goodness is only defined for statements that terminate (normally). We do not consider other termination behavior since we study termination-insensitive non-interference. Goodness is then defined by two conjuncts. The first conjunct is almost the same as Definition 4, which defines confidentiality. The only difference is that now the newly obtained labeling function –acquired by application of labStat and labExpr– is used to determine which variables are in $\Downarrow$conf *after* evaluation of s. Thus the labeling function is no longer static, as is the case in the definition of termination-insensitive non-interference.

The second conjunct is an invariant property that we will need to prove that the labeling transition functions for conditional statements are good. It states that if the value of any variable changes by executing s, then the new security level of this variable should at least be the same as the environment level lenv.

Definition 8 also illustrates that our abstract labeling transition functions are related to the underlying (standard) denotational semantics via goodness. Hence our work is a form of abstract interpretation [Cou96].

At this point we can prove our main theorem. It states that our approach is sound, i.e., that goodness together with non-increasingness implies confidentiality.

Theorem 1 (Soundness)**.**

$$\forall \mathsf{s} \in \mathsf{Statements}.\forall \mathsf{lab} : \mathsf{Loc} \rightarrow \mathsf{L.Good}(\mathsf{s}) \wedge \mathsf{Decreasing}(\mathsf{s}, \mathsf{lab}) \Rightarrow \mathsf{Confidential}(\mathsf{s}, \mathsf{lab})$$

Proof. Assume $\mathsf{Good}(\mathsf{s})$, then $[\![\mathsf{s}]\!](x) \neq *$ and $[\![\mathsf{s}]\!](y) \neq *$ and also $\mathcal{I}(\mathsf{conf}, \mathsf{lab}) \ni (x, y) \Rightarrow \mathcal{I}(\mathsf{conf}, \mathsf{labStat}(\mathsf{s}, \bot)) \ni ([\![\mathsf{s}]\!](x), [\![\mathsf{s}]\!](y))$.

Now assume $\mathcal{I}(\mathsf{conf}, \mathsf{lab}) \ni (x, y)$ which gives us

$$\mathcal{I}(\mathsf{conf}, \mathsf{labStat}(\mathsf{s}, \bot)) \ni ([\![\mathsf{s}]\!](x), [\![\mathsf{s}]\!](y))$$

which in turn is equivalent to

$$\forall l : \mathsf{Loc.conf} \not\leq \mathsf{labStat}(s)(\mathsf{lab}, \bot)(l) \Rightarrow [\![\mathsf{s}]\!](x)(l) = [\![\mathsf{s}]\!](y)(l)$$

There are now two possible cases:

case $\mathsf{conf} \leq \mathsf{labStat}(s)(\mathsf{lab}, \bot)(l)$

Assumption $\mathsf{Decreasing}(\mathsf{s}, \mathsf{lab})$ gives us $\mathsf{labStat}(\mathsf{s})(\mathsf{lab}, \bot)(l) \leq \mathsf{lab}$. From this it follows that $\mathsf{conf} \leq \mathsf{lab}$ and thus that $\mathcal{I}(\mathsf{conf}, \mathsf{lab}) \ni ([\![\mathsf{s}]\!](x), [\![\mathsf{s}]\!](y))$.

case $\mathsf{conf} \not\leq \mathsf{labStat}(s)(\mathsf{lab}, \bot)(l)$

Then $[\![\mathsf{s}]\!](x)(l) = [\![\mathsf{s}]\!](y)(l)$ from which $\mathcal{I}(\mathsf{conf}, \mathsf{lab}) \ni ([\![\mathsf{s}]\!](x), [\![\mathsf{s}]\!](y))$ follows directly.

Thus we have shown:

$$\mathcal{I}(\mathsf{conf}, \mathsf{lab}) \ni (x, y) \Rightarrow \mathcal{I}(\mathsf{conf}, \mathsf{lab}) \ni ([\![\mathsf{s}]\!](x), [\![\mathsf{s}]\!](y))$$

and thus $\mathsf{Confidential}(\mathsf{s}, \mathsf{lab})$.

□

We can use Theorem 1 to formalize an approach for statically checking confidentiality. This is accomplished in two steps.

Proposition 1.

$$\forall \mathsf{s} \in \mathsf{Statements}.(\forall x : \mathsf{M}.[\![\mathsf{s}]\!](x) \neq *) \Rightarrow \mathsf{Good}(\mathsf{s})$$

Proof. By induction on the structure of `s`. All cases except for the `while` are straightforward. The `while` case handled is by induction on the number of iterations in memory x and induction loading on the number of iterations in memory y. □

Proving this proposition is where most of the effort of our work has been concentrated. Here it turned out that formalizing our model in the theorem prover PVS was very useful since the many small subtleties we encountered can easily be overlooked if one tries to do these proofs on paper. Proving the `while`-case especially formed a challenge.

Corollary 1 then gives us our main result.

Corollary 1.

$$\begin{aligned}&\forall \mathsf{s} \in \mathsf{Statements}.\forall \mathsf{lab} : \mathsf{Loc} \to \mathsf{L}.\\ &\quad (\forall x : \mathsf{M}.[\![\mathsf{s}]\!](x) \neq *) \ \wedge \ \mathsf{Decreasing}(\mathsf{s}, \mathsf{lab}) \Rightarrow \mathsf{Confidential}(\mathsf{s}, \mathsf{lab})\end{aligned}$$

Statically checking confidentiality then involves computing the labeling transition functions and checking if non-increasingness holds.

6.4 Examples

In this section we illustrate the use of our approach with some simple example programs. We shall first apply it to the statement $l := h; l := 2$ (from the introduction), using the simple security lattice as representation for the security policy. The initial lab is $\{l : \mathsf{Low}, h : \mathsf{High}\}$ and the environment level is initially Low:

$$\begin{aligned}&\mathsf{labStat}(l := h; l := 2)(\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{Low}) &&= \\ &\mathsf{labStat}(l := 2)(\mathsf{labStat}(l := h)(\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{Low}), \mathsf{Low}) &&= \\ &\mathsf{labStat}(l := 2)(\{l : \mathsf{High}, h : \mathsf{High}\}, \mathsf{Low}) &&= \\ &\{l : \mathsf{Low}, h : \mathsf{High}\}\end{aligned}$$

We did not show the trivial steps of applying $\mathsf{labExpr}$ to constants or variables. Since the labeling stays the same, non-increasingness holds and thus we conclude by corollary 1 that this 'program' maintains confidentiality.

To show the use of the environment level in implicit secure information flow we look at the program fragment `if-then-else`$(h == 1)(l = 1)(l = 2); l = 0$ that branches over a variable with security level High. Again, the security level of h is High and the level of l is Low. Using the informal notation from the introduction the analysis of this program works as follows:

Example 4.

$$\left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{Low} \\ l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\} \quad (h \ \begin{array}{c}\texttt{if}\\ ==\end{array}\ 1) \quad \left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{High} \\ l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\} \quad \begin{array}{lcl} l & := & 1 \\ & or & \\ l & := & 2 \end{array} \quad ; \quad \left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{Low} \\ l & : & \mathsf{High} \\ h & : & \mathsf{High} \end{array}\right\}$$

$$l := 0 \qquad \left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{Low} \\ l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\}$$

Notice how the environment levels becomes High inside the branches of the `if` statement and is has security level Low again outside the scope of the `if`.

Figure 6.1 shows how our the abstract labeling functions are applied to Example 4. We again use $\{l : \mathsf{Low}, h : \mathsf{High}\}$ for an initial lab; the environment level is initially Low.

$$\begin{array}{lll}
\mathsf{labStat}(\texttt{if-then-else}(h == 1)(l := 1)(l := 2); l := 0)(\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{Low}) & = \\
\mathsf{labStat}(l := 0)(\mathsf{labStat}(\texttt{if-then-else} \quad (h == 1)(l := 1)(l := 2) & \\
\qquad\qquad (\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{Low}), \mathsf{Low}) & = \\
\mathsf{labStat}(l := 0)(\quad \mathsf{labStat}(l := 1)(\quad \{l : \mathsf{Low}, h : \mathsf{High}\} \sqcup & \\
\qquad\qquad \pi_1(\mathsf{labExpr}(h == 1)), & \\
\qquad\qquad \pi_2(\mathsf{labExpr}(h == 1)) \sqcup \mathsf{lenv}) \sqcup & \\
\qquad\qquad \mathsf{labStat}(l := 2)(\quad \{l : \mathsf{Low}, h : \mathsf{High}\} \sqcup & = \\
\qquad\qquad \pi_1(\mathsf{labExpr}(h == 1)), & \\
\qquad\qquad \pi_2(\mathsf{labExpr}(h == 1)) \sqcup \mathsf{lenv}), \mathsf{Low}) & \\
\mathsf{labStat}(l := 0)(\quad \mathsf{labStat}(l := 1)(\quad \{l : \mathsf{Low}, h : \mathsf{High}\} \sqcup & \\
\qquad\qquad \{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{High} \sqcup \mathsf{lenv}) \sqcup & \\
\qquad\qquad \mathsf{labStat}(l := 2)(\quad \{l : \mathsf{Low}, h : \mathsf{High}\} \sqcup & = \\
\qquad\qquad \{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{High} \sqcup \mathsf{lenv}), \mathsf{Low}) & \\
\mathsf{labStat}(l := 0)(\quad \mathsf{labStat}(l := 1)(\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{High}) \sqcup & \\
\qquad\qquad \mathsf{labStat}(l := 2)(\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{High}), \mathsf{Low}) & = \\
\mathsf{labStat}(l := 0)(\{l : \mathsf{High}, h : \mathsf{High}\} \sqcup \{l : \mathsf{High}, h : \mathsf{High}\}, \mathsf{Low}) & = \\
\mathsf{labStat}(l := 0)(\{l : \mathsf{High}, h : \mathsf{High}\}, \mathsf{Low}) & = \\
\{l : \mathsf{Low}, h : \mathsf{High}\} &
\end{array}$$

where π_1 and π_2 are first and second projection.

Figure 6.1: Application of abstract labeling functions to Example 4.

NonIncreasingness holds again for this example since the labeling before and after applying labStat and labExpr is exactly the same. We conclude by corollary 1 that the program maintains confidentiality. This example also illustrates that statically checking confidentiality with the abstract labeling functions is possible. The different steps in applying the labeling transition functions only involve substitutions and calculating maximums. We have run

this example in PVS where we loaded labStat en labExpr as automatic rewrite rules. It took PVS a fraction of a second to automatically prove confidentiality for this example.

The following program fragment shows –in an abstract way– how we process a `while`. In this piece of code `h` initially has security level High, `l1,l2` and `l3` start with security level Low and the environment level lenv has initial level Low.

Example 5. `while (l1 > (l1 := l2)) {l1 := h; l3++}`

The table in Figure 6.2 shows the security levels of the different variables after each iteration, where zero iterations means that only the conditional of the `while` is evaluated, one iteration means that the conditional is evaluated, then the body and then the conditional again, etc. Notice that after two iterations a fixed point is reached. The application

Iterations	`h`	`l1`	`l2`	`l3`	lenv
0	High	Low	Low	Low	Low
1	High	Low	Low	Low	High
2	High	High	Low	High	High
3	High	High	Low	High	High

Figure 6.2: Security levels of variables after each `while` iteration.

of the abstract labeling functions will stop at this point. Both `l1` and `l3` have a higher security level than before execution of this program fragment, hence it does not maintain confidentiality. We will not show the application of the rewrite rules here. Using PVS we evaluated this example –using automatic rewrite rules– in about 5 seconds, after which the program was identified as (possibly) breaking confidentiality. The time needed to check this small example is of course too long for a practical tool. However, an optimized dedicated tool should shorten this time considerably. If such a tool will be fast enough to be of any practical value remains to be seen. Further research and an actual implementation is needed for this.

6.5 Possible Extensions

We ultimately wish to extend our approach to a programming language like (sequential) Java. This leads to a number of additional challenges which are briefly discussed in this section.

6.5.1 Indistinguishable objects and heaps

Adding objects is, in principle, straightforward but, unlike the primitive types (such as the booleans and integers which are used in this chapter), simply comparing objects by

looking at reference equality will not be enough to establish confidentiality. One particular problem arises when we create new objects[48]. Consider the next code fragment:

Example 6. `if(high > 0) then Object o := new Object();`

Since the guard has security level High in this example, for some values of memory states the `then` part will be evaluated and for other memory states this will not be the case. So after this statement the heaps can be *unbalanced*. In our current work we only compare memory locations, but in this situation this will not be enough. If the heaps are unbalanced, the same location in the two heaps can refer to different objects. In essence, we want that objects created under a high context are indistinguishable from the outside for an attacker.

An indistinguishablity relation for objects and heaps can fix this. Following [BN05] we then have to define indistinguishablity relative to a partial bijection on reference locations. We do this in the context of *time-sensitive* non-interference for an object-oriented language in Definition 24 in Appendix B. Also see the next chapter for a different solution to this problem.

6.5.2 Exceptions

In languages like Java, programs cannot only terminate normally or hang, they can also throw exceptions. Taking this into account complicates our model considerably. In case of exceptional termination, temporary breaches of confidentiality cannot occur, since each exception propagates until its catch clause, without restoring confidentiality along the way. Since every statement or expression can possibly throw an exception, and we do not want to exclude temporary breaches of confidentiality completely, we have to calculate *two* labeling functions, one for normal termination and one for exceptional termination. Moreover, if an exception may be thrown by a certain statement, the statements after this one should have as environment level the level of the condition under which the exception is thrown, since their execution depends on this condition[49].

6.5.3 Method calls

(Non-recursive) methods calls can easily be added to our language. The main idea is to just propagate the labeling function lab and the environment level lenv. So if at a certain point inside a program a method is called we simply compute the labeling function that results from this method call by analyzing the method starting with the labeling function and environment level at the point the method is called. After analysis of the method call

[48] Creating a new object (in Java) may also possibly throw an `OutOfMemoryException`. Since the memory model we use (from the LOOP semantics) is modeled by *infinite* lists (in PVS), such memory violations are not considered here. As an aside, these kind of memory errors can also leak secret information –via a so called covert channel– thereby breaking confidentiality. Also see Chapter 8.

[49] The situation becomes even more complicated if we take the different internal termination modes into account. In Java, statements can terminate abnormally via an exception, a break, a continue or a return.

the new labeling function and environment level are used for the remainder of the analysis of the original program.

Recursive method calls are more elaborate and require a treatment similar to the `while` statement.

6.5.4 Assertions

Related to the problem of multiple termination modes is the following example:

Example 7. `MyObject o := new MyObject(1); low := o.f`

Here `o` is some high object with field `f` that has value 1. If we do abstract interpretation, and consider the second statement separately, we do not know if the object is a `null` reference or not. Therefore, the statement can throw a `NullPointerException`. This means that even if we include exceptional termination in our approach we will only be able to confirm confidentiality if this exception is caught in a surrounding try-catch block. Otherwise, we will not be able to verify that the termination mode does not depend on high variables.

A possible solution here is the use of *assertions*. Looking back at Example 7, if we know that object `o` is never `null` at the start of the second statement, we only have to consider normal termination. The assertion then needs to be proved separately using a tool like ESC/JAVA2 [CK04] or LOOP [JP04].

6.5.5 Completeness

Our approach for checking confidentiality is not complete in the sense that some programs which are confidential are identified as (possibly) leaking information (see Section 6.2). An example of a program fragment that is considered to be insecure by our approach is given below:

Example 8. `low := high; low := low - high;`

This code fragment does not leak any information from the high variable `high` to the low variable `low`, because after complete evaluation the variable `low` will always have value zero. However, the abstract labeling functions will assign the security level High to the variable `low`: the assignment `low := high` will assign security level High to `low` and the next assignment `low := low - high` will again assign security level High to `low`, because the labeling function for minus involves calculating the maximum of the security levels of `low` and `high` is High (in fact, both variables are high here). The problem with this example is that we need more information on the *semantic* level, which we do not have in our abstract semantics. Other automatic approaches, such as those based on type-checking will also identify this example as possibly leaking information.

Assertions can also be useful when dealing with these situations were we need more semantic information. In Example 8 we can add the assertion that variable `low` will always have value zero at the end of this code fragment. If this assertion is true, then we can treat variable `low` from this point onwards again as a variable with security level Low.

6.5.6 Aliasing

Even if all the proposed extensions work, there still remain a couple of open problems.The main one being *aliasing*. Automatic analysis of aliasing of objects is a notoriously hard problem by itself. In combination with variable security levels this problem becomes even harder –the security levels in the automatic secure information flow analyzer JIF [Mye99] are static because of aliasing[50].

At the moment we have no idea how the handle aliasing in the approach discussed here. Several solutions for this problem have been proposed, e.g., in [ABB06] and in Chapter 7 of this thesis, but it is unclear if and to what extend these approaches can be integrated in our current work.

6.6 Related work

In this section we focus on closely related work on abstract interpretation. We refer to Section 2.5 for a general overview of the field.

Applying abstract interpretation [Cou96, CC77] to confidentiality is not new. Giacobazzi and Mastroeni [GM04] have formalized a notion of confidentiality based on abstract interpretation. Their main idea is to formalize an explicit attacker and define non-interference in terms of what this attacker can observe. The language they consider is similar to ours except that they do not allow side-effects in expressions.

Avvenuti et al [ABDF03] formalized an algorithm for assuring confidentiality for Java bytecode based on abstract interpretation. The main difference with our work is that the secrecy labels associated with variables are *static*, i.e., do not change during the abstract evaluation. This means that Avvenuti et al cannot check temporary breaches of confidentiality.

Zanotti [Zan02] uses abstract interpretation in a way related to ours. However instead of applying abstract label transition functions and then afterwards checking if non-increasingness holds, Zanotti constructs at each assignment a set of allowed assignments, i.e., those that do not violate confidentiality, and checks if the assigned variable is in this set. We suspect that for larger programs this will not work as well as our approach, since the constructed set of assignable variables can become very large.

6.7 Conclusions

We have presented a new approach for automatically proving termination-insensitive non-interference. The framework is completely formalized and proved to be sound within the higher-order theorem prover PVS. Based on this model a static approach for checking confidentiality is given, which we have illustrated via rewriting in PVS. We argue that this approach can easily be integrated in existing (rewriting) tools for static program verification.

[50] Thanks are due to David Sands for pointing this out.

Chapter 7

Interactively proving termination-sensitive non-interference

Chapter 6 concerned a method for automatically checking termination insensitive non-interference for WHILE, which is only a small subset of sequential Java. In contrast, this chapter focuses on interactively proving termination sensitive non-interference for full sequential Java.

Our approach for proving non-interference extends the Java semantics developed for the LOOP project (briefly discussed in Section 2.2). Instead of proving the correctness of predicates on the state space using LOOP's (ordinary) Hoare logic, one can use a novel Hoare logic on relations to prove bisimulations on the state space. Such bisimulations can be used to express non-interference properties, where we shall focus on confidentiality.

As far as we are aware, the extension on the LOOP project that is presented in this chapter is the first that provides a provably sound verification framework for full sequential Java. The JIF [Mye99, Jif] system also covers all of sequential Java –moreover, it can verify (termination-insensitive) non-interference fully automatically– but a soundness result has never been presented.

This chapter is also the first work that systematically studies non-interference for all termination modes of a language like Java: so called *termination-sensitive* non-interference. Others have studied exceptions [ML97] and (limited forms of) non-termination [VS97b], but the work presented in this chapter also considers (labeled and unlabeled) break and continue statements.

The remainder of this chapter is organized as follows: the next section explains how bisimulation for a (Java) class can be used to express non-interference. Section 7.2 then introduces a relational Hoare logic for a simple WHILE-like language that can be used to prove non-interference as bisimulation. Section 7.3 discusses how the current LOOP framework can be extended to prove non-interference for full sequential Java. A relational Hoare logic for sequential Java is then introduced in Section 7.4, it can be used to interactively prove termination-sensitive non-interference. Section 7.5 shows a number of (verified) ex-

amples, Section 7.6 discusses related work and we end with conclusions and suggestions for future work in section 7.7.

7.1 Non-interference through bisimulation

The notion of invariant for a class is well-established in object-oriented programming. It is a predicate on the state space State that is maintained by all methods[51], i.e., a subset $P \subseteq \mathsf{State}$ such that:

Definition 9 (Invariants for (Java) classes)**.**
P is an invariant for all methods m of the class iff

$$\forall x \in \mathsf{State}.\, P(x) \Longrightarrow P([\![m]\!](x))$$

Additionally, one should require that P is established by the constructors. Invariants form a fundamental construct in JML; they are typically used to make assumptions explicit that the programmer has in the back of his/her mind. Invariants in JML are so called *weak* invariants, because they can be broken inside a method body as long as they are restored before the method terminates (either normally or via an abnormal termination). In contrast, a *strong* invariant is an invariant that must never be broken.

A bisimulation is a relation between states of different runs[52], they form a standard for showing indistinguishability. Bisimulations appeared in the context of transition systems and process calculi, see e.g., [Par81, Mil89]. They are studied in abstract form in the context of state-based computation provided by the theory of coalgebras [Rut00, Jac02, RTJ01]. This forms a source of inspiration for the current work, since the underlying (LOOP) semantics of Java has a coalgebraic formulation (see Section 2.2 and [JP03] for a discussion).

Bisimulations for a class are less familiar in object-oriented programming. The basic idea is, like in Definition 9, that they are relations that are maintained by all methods. This can be expressed as: a relation $R \subseteq \mathsf{State} \times \mathsf{State}$ that satisfies

Definition 10 (Bisimulations for (Java) classes)**.**
R is a bisimulation for all methods m of the class iff

$$\forall x, y \in \mathsf{State}.\, R(x, y) \Longrightarrow R([\![m]\!](x), [\![m]\!](y))$$

Bisimilarity between classes is then defined as the union of all bisimulations. It is itself a bisimulation—and an equivalence relation—and is the formalization of the notion of indistinguishability of states. This means that we regard two states x, y as "equal as far as we can see from the outside" in case we can find a bisimulation R with $R(x, y)$. Similar

[51] For now, methods can simply be seen as functions on the state space, i.e., $m : \mathsf{State} \to \mathsf{State}$.

[52] Recall that JML also supports the notion of a (history) `constraint`. This is a relation (or binary predicate) between pre- and post-states of method calls. It is thus a relation between states *in the same run*.

to invariants we also distinguish between *weak* bisimulations[53], that can be broken inside a method body as long they are re-established when the method terminates, and *strong* bisimulations that can never be broken.

7.1.1 Confidentiality as bisimulations in classes

Bisimulations can be used to formulate non-interference for a class (in an object-oriented language). In the sequel, the simple security lattice $\Sigma = \{\mathsf{Low}, \mathsf{High}\}$ with $\mathsf{Low} \sqsubseteq \mathsf{High}$ is again assumed, where Low denotes 'low' or public information and High represents 'high' or secret information. We assume a function $\mathsf{sl} : \mathsf{Loc} \to \Sigma$ maps memory locations (containing fields) to elements in the security lattice. We overload notation and assume that the set Field of all fields of the class is partitioned into two sets: $\mathsf{High} = \{\ l \in \mathsf{Loc} \mid \mathsf{sl}(l) = \mathsf{High}\}$ and $\mathsf{Low} = \mathsf{Field} \setminus \mathsf{High}$.

We wish to express that the information that is held in the high fields remains confidential. If we adapt the notion of confidentiality as non-interference to the current setting than confidentiality can be defined as:

Definition 11 (Confidentiality)**.**
A (Java) class C maintains confidentiality iff the relation $L \subseteq \mathsf{State} \times \mathsf{State}$ with

$$L = \{(x, y) \mid \ell(x) = \ell(y), \text{ for all fields } \ell \in \mathsf{Low}\}$$

is a bisimulation for class C[54].

This (weak) bisimulation property expresses confidentiality because if we have two states x, y where we know that low fields are equal, but where we have no information about the high fields, then after each method call these possibly different high values of x and y should not interfere with the low fields: the lows should still be equal. Hence confidentiality says that equality of low fields should be maintained.

Dually, one may formulate integrity as: equality of high fields should be maintained. Hence differences in low fields should not be visible in the high fields. Since this dual property of integrity is can be proved in a similar fashion as confidentiality, we shall not elaborate it in this chapter.

7.2 A relational Hoare logic for WHILE

In order to prove that a program is non-interfering (confidential) we thus need to prove that a (non-interference) relation is maintained by all methods of a class. The relational formulation of confidentiality in Definition 10 requires that each method m is evaluated twice, once in state x and once in state y. This is very inconvenient in proofs.

[53] Usually, i.e., in the context of transition systems and process calculi, weak bisimulation means that (silent) τ-steps are allowed.

[54] The observant reader might notice that –using Definition 3 from the precious chapter– the following is equivalent to Definition 11: $L = I(\mathsf{High}, \mathsf{sl})$

The Hoare logic for relations that we introduce in this chapter is designed precisely to handle this problem. It uses relations instead of predicates in its classical Hoare triples, and uses structural rules to prove relational program properties. As far as we are aware, using a Hoare logic for relations for proving non-interference properties is new. See Section 7.6 for a discussion on other relational Hoare logics and their uses.

Here in Section 7.2, we first explain the basic idea of a relational Hoare logic for WHILE from Definition 1. In Section 7.3 this basic idea is then extended to full sequential Java.

As explained in Section 7.1 the relational approach to confidentiality requires proving that specific relations are bisimulations, i.e., are maintained by all methods. We will denote statements s, s_1, s_2 and expressions e, e_1, e_2 with small case letters and use capital letters R, T, S for predicates and relations. For a predicate C we use a short hand notation C^2 ("C squared") for the relation given by $C^2(x, y) = C(x) \wedge C(y)$, with $x, y \in \mathsf{State}$. The partial relational Hoare triple $\{S\}\ s\ \{R\}$ can be interpreted as: 'if the relation S holds in the pre-state of statement s and if statement s terminates, then the relation R holds in the post-state of statement s'.

The main rules for the simple relational Hoare logic are displayed in Figure 7.1.

$$\frac{S \Rightarrow R[a/v(x), a/v(y)]}{\{S\}\ v := a\ \{R\}}\ \text{assignment}$$

$$\frac{\{R\}\ s_1\ \{T\} \qquad \{T\}\ s_2\ \{S\}}{\{R\}\ s_1; s_2\ \{S\}}\ \text{composition}$$

$$\frac{R \Rightarrow C^2 \vee (\neg C)^2 \qquad \{R \wedge C^2\}\ s_1\ \{S\} \qquad \{R \wedge (\neg C)^2\}\ s_2\ \{S\}}{\{R\}\ \texttt{if-then-else}(c)(s_1)(s_2)\ \{S\}}\ \text{if-then-else}$$

$$\frac{R \Rightarrow C^2 \vee (\neg C)^2 \qquad \{R \wedge C^2\}\ s\ \{S\}}{\{R\}\ \texttt{while}(c)(s)\ \{(\neg C)^2 \wedge S\}}\ \text{while (partial)}$$

Figure 7.1: Relational Hoare logic for WHILE.

Since the Hoare triples are partial correctness triples, i.e., do not require that a statement terminates, the relational Hoare logic can only be used to prove termination-insensitive non-interference properties for programs written in WHILE. Notice that the logic is sound with respect to WHILE's semantics (given in Section 2.2.1):

Lemma 1. *The relational Hoare logic defined in Figure 7.1 is sound, i.e.,*

$$\forall s : \mathsf{Statement}.\ \textit{if}\ \vdash \{R\}\ s\ \{T\}\ \textit{then}\ \forall q_1, q_2 : \mathsf{M}.\ R(q_1, q_2) \Rightarrow T([\![s]\!](q_1), [\![s]\!](q_2))$$

Proof. Straightforward structural induction. □

The following is also obvious:

Remark 1. *The relational Hoare logic defined in Figure 7.1 is* not *complete, i.e.,*

$$\begin{array}{l}\forall s : \mathsf{Statement}.\\ \quad \textit{if } \forall q_1, q_2 : \mathsf{M}.\ R(q_1, q_2) \Rightarrow T([\![s]\!](q_1), [\![s]\!](q_2))\\ \quad\quad \textit{then it is not } \text{always } \textit{the case that } \vdash \{R\}\ s\ \{T\}\end{array}$$

This follows from a simple inspection of the rules for the `if-then-else` and `while`. These rules all use the conditional C^2 which forces an evaluation of the same branch (or the same number of iterations for a `while` statement) in both states. Formally there are four different possibilities here : $C(x) \wedge C(y)$, $\neg C(x) \wedge \neg C(y)$, $\neg C(x) \wedge C(y)$ and $C(x) \wedge \neg C(y)$. Excluding the latter two cases ensures that the relational Hoare logic is not complete.

We *have* to exclude these cases if we want to restrict the relational Hoare rules to evaluation of the same (one) statement or expression at the time. In cases where the boolean values of $C(x)$ and $C(y)$ are different confidentiality is (possibly locally) broken. Recall that only fields in set High can have different values in memory x and y. A similar argument can be made for the rules for while statements.

If we want to be able to prove termination-sensitive non-interference we have to add a variant[55] to the rule for `while` statements. Figure 7.2 below shows the total relational Hoare rule for `while` statements.

$$\frac{R \Rightarrow C^2 \vee (\neg C)^2 \qquad [R \wedge C^2 \wedge (\text{variant} = n)^2]\ s\ [S \wedge (\text{variant} < n)^2]}{[R]\ \texttt{while}(c)(s)\ [(\neg C)^2 \wedge S]}\ \text{while (total)}$$

Figure 7.2: Total relational Hoare logic for `while` statements.

Notice how we simply duplicate the variant in the total correctness rule for while statements, basically requiring that the number of iterations is equal for all states.

Using these rules we can thus only prove strong bisimulation properties, not the more desirable weak one. In the next sections we will show how we solved this problem and how we have extended these rules to full sequential Java.

[55] A variant gives a mapping from the underlying state space to some well-founded set. Termination may be proved by showing that, for each execution of the body of the while-loop, this mapping decreases. Formally, we have to define two mappings, but, for our approach, one mapping suffices. Again if the variant differs for separate states confidentiality will (at least locally) be broken.

7.3 Extension to sequential Java

When one actually tries to prove that a particular relation is a bisimulation for a class in a concrete language like Java, it quickly becomes clear that the formulation of confidentiality given in Definition 11 is an oversimplification, and that several additional aspects have to be taken into account.

1. Methods in Java have different termination modes: they may hang, terminate normally, or throw an exception[56]. This means that instead of "$R(m(x), m(y))$" in 10 we should write: "$m(x)$ and $m(y)$ should have the same termination mode, and in that mode the result states (if any) should be related again by R".

2. Parameters of methods need to be included. So, if a method m has parameters $\overrightarrow{a}$ we use quantification to fix their values. So instead of requiring $R(m(x), m(y))$ as conclusion, we should write $\forall \overrightarrow{a}\,.\, R(m(x)(\overrightarrow{a}), m(y)(\overrightarrow{a}))$.

3. Methods in Java can also return a result (when they have a non-void return type). We should require in our adaptation of Definition 10 that these results are again 'indistinguishable', where indistinguishable can –for the moment– be interpreted as Java's `equals` methods. Indistinguishability depends in particular on the type of the result: thus if the results of $m(x)$ and $m(y)$ are of

   - primitive type, then they should be equal;
   - reference, but not an array, type, then they should be either both null, or both be references to 'indistinguishable' objects;
   - an array type, they should either be both null, or both be references to arrays with the same length and with 'indistinguishable' entries at each position.

   We define this indistinguishability notion formally in Definition 17.

4. For each (public) field f of the class, we should have that if $R(x, y)$, then $f(x)$ and $f(y)$ should be indistinguishable, like we just described for results of methods.

Within the Java verification work centered around the LOOP tool [BJ01, JP04] there is a well-developed semantics for sequential Java formalized in the theorem prover PVS. The relational Hoare logic for Java will be formalized on top of this semantics. In the next section we briefly explain a relevant subset of this Java semantics.

[56] This only describes the situation from the outside. Inside method bodies we may also have abrupt termination because of a return, break or continue statements.

7.3.1 Interlude: Java semantics in the LOOP project

The Java semantics that has been formalized in the LOOP project has been modeled in the higher order theorem prover PVS. We will not use the standard PVS notation here, but will continue to use a more general logical/mathematical notation.

LOOP's Java memory model, consisting of a heap and a stack, is denoted by the type M. Typical getter and setter functions (in PVS) manipulate it, see [BHJP00] and [Hui01, Chapter 2] for details. Java statements and expressions are represented as functions $\mathsf{M} \to \mathsf{StatResult}$ and $\mathsf{M} \to \mathsf{ExprResult}[\mathsf{Out}]$ for an output type Out (which –in Java– can be either a primitive or a reference type). The result types are defined as labeled coproduct types:

$$\begin{array}{l} \mathsf{StatResult} \ : \ \mathsf{TYPE} \equiv \ \{ \\ \quad \mathsf{hang} \ : \ \mathsf{unit} \\ \mid \ \mathsf{norm} \ : \ \mathsf{M} \\ \mid \ \mathsf{abnorm} \ : \ \mathsf{StatAbn} \ \} \end{array} \qquad \begin{array}{l} \mathsf{ExprResult}[\mathsf{Out}] \ : \ \mathsf{TYPE} \equiv \ \{ \\ \quad \mathsf{hang} \ : \ \mathsf{unit} \\ \mid \ \mathsf{norm} \ : \ [\mathsf{ns} : \mathsf{M}, \mathsf{res} : \mathsf{Out}] \\ \mid \ \mathsf{abnorm} \ : \ \mathsf{ExprAbn} \ \} \end{array}$$

where the labels hang, norm and abnorm represent non-termination, normal termination and abnormal termination in Java, respectively. An expression that terminates normally returns both a new memory state and an output type (of type Out). The abnormal termination types StatAbn and ExprAbn formalize the different abrupt termination forms of statements and expressions:

$$\begin{array}{l} \mathsf{StatAbn} \ : \ \mathsf{TYPE} \equiv \ \{ \\ \quad \mathsf{exp} \ : \ [\mathsf{es} : \mathsf{M}, \mathsf{ex} : \mathsf{RefType}] \\ \mid \ \mathsf{return} \ : \ \mathsf{M} \\ \mid \ \mathsf{break} \ : \ [\mathsf{bs} : \mathsf{M}, \mathsf{blab} : \mathsf{lift}[\mathsf{string}]] \\ \mid \ \mathsf{continue} \ : \ [\mathsf{cs} : \mathsf{M}, \mathsf{clab} : \mathsf{lift}[\mathsf{string}]] \ \} \end{array} \qquad \begin{array}{l} \mathsf{ExprAbn} \ : \ \mathsf{TYPE} \equiv \\ \quad [\mathsf{es} : \mathsf{M}, \mathsf{ex} : \mathsf{RefType}] \end{array}$$

The type RefType represents references. The RefType above denotes a reference to an exception object ex. Java's `break` and `continue` constructs can be used with and without label, this is indicated in the labeled product types above, where both bs and cs indicate the resulting state (that incorporates the side-effect) and blab and clab are labels. The lifted type lift[string] adds a bottom element to an arbitrary type (in this case string) which ensures that labeled and unlabeled breaks and continues can be represented by the same type.

These types for Java statements and expressions form the basis for the formalization of the semantics of (sequential) Java in the theorem prover PVS. All statements and expressions in Java have to be formalized separately, as an example we give the semantics for composition in Definition 12.

Definition 12 (Composition)**.** Let $p, q : [\mathsf{M} \to \mathsf{StatResult}]$, composition in the LOOP Java semantics is then defined as:

```
(p ; q) : [M → StatResult] =
  λ(x : M) :
    CASES p(x) OF {
        hang          ↦  hang,
      | norm(y)       ↦  q(y),
      | abnorm(a)     ↦  abnorm(a) }
```

A sound Hoare logic has been formalized 'on top of' this Java semantics. The extended Hoare n-tuples take all the different termination modes of Java into account. Definition 13 defines the types (for statements and expression) of these Hoare tuples, where the boolean type is denoted with $\mathbb{B}$. Notice the similarity with JML method specifications:

Definition 13 (Hoare n-tuples for Java)**.**

```
StatBehaviorSpec : TYPE ≡
 [  diverges : M → 𝔹,
    requires : M → 𝔹,
    statement : M → StatResult,
    ensures : M → 𝔹,
    signals : M → RefType → 𝔹,
    return : M → 𝔹,
    break : M → lift[string] → 𝔹,
    continue : M → lift[string] → 𝔹, ]
```

```
ExprBehaviorSpec[Out] : TYPE ≡
 [  diverges : M → 𝔹,
    requires : M → 𝔹,
    expression : M → ExprResult[Out],
    ensures : M → Out → 𝔹,
    signals : M → RefType → 𝔹 ]
```

The functions SB and EB then relate the Hoare tuples sbs : StatBehaviorSpec and ebs : ExprBehaviorSpec to the underlying Java semantics. The precise interpretation of these functions can be found in Definition 14.

Definition 14 (Semantic interpretation functions SB and EB)**.**

```
sbs : StatBehaviorSpec ⊢ SB · sbs : 𝔹 ≡
  ∀x ∈ M : sbs.requires · x ⇒
    CASES sbs.statement · x OF {
        hang ↦ sbs.diverges · x
      | norm y ↦ sbs.ensures · y
      | abnorm a ↦ CASES a OF {
                     excp e ↦ sbs.signals · (e.es) · (e.ex)
                   | return z ↦ sbs.return · z
                   | break b ↦ sbs.break · (b.bs) · (b.blab)
                   | continue c ↦ sbs.continue · (c.cs) · (c.clab) } }
ebs : ExprBehaviorSpec[Out] ⊢ EB · ebs : 𝔹 ≡
  ∀x ∈ M : ebs.requires · x ⇒
    CASES ebs.statement · x OF {
        hang ↦ ebs.diverges · x
      | norm y ↦ ebs.ensures · (y.ns) · (y.res)
      | abnorm a ↦ ebs.signals · (a.es) · (a.ex) } }
```

The semantic functions SB and EB then allow us to interpret (classical) Hoare triples. Thus, the partial Hoare triple $\{P\}\ s\ \{Q\}$ is related to an extended Hoare n-tuple as follows:

$$\{P\}\ s\ \{Q\}\ =\ \mathsf{SB}\cdot\left(\begin{array}{l}\mathsf{diverges} = \lambda x : \mathsf{M.true},\\ \mathsf{requires} = P\\ \mathsf{statement} = s\\ \mathsf{ensures} = Q\\ \mathsf{signals} = \lambda x : \mathsf{M}.\lambda e : \mathsf{RefType.true}\\ \mathsf{return} = \lambda x : \mathsf{M.true}\\ \mathsf{break} = \lambda x : \mathsf{M}.\lambda l : \mathsf{lift[string].true}\\ \mathsf{continue} = \lambda x : \mathsf{M}.\lambda l : \mathsf{lift[string].true}\end{array}\right)$$

The partial Hoare tuple above should be interpreted in the same way as classical partial Hoare triples: "assuming that predicate P holds in the pre-state of statements s, then *if* s *terminates*, predicate Q should hold in the post-state of s". If all true values above are changed into false, then the n-tuple becomes a *total* correctness tuple, meaning that statement s *must* terminate normally.

The Hoare tuples can then be used to formulate (total) Hoare rules[57]. In the actual PVS formalization these rules are specified as provable lemmas. Lemma 2 gives the Hoare rule for composition, as an example.

Lemma 2 (Composition Hoare rule in LOOP)**.**

$$\exists T : \mathsf{M} \to \mathbb{B}.$$

$$\mathsf{SB}\cdot\left(\begin{array}{l}\mathsf{diverges} = \lambda x : \mathsf{M}.b,\\ \mathsf{requires} = P,\\ \mathsf{statement} = s_1,\\ \mathsf{ensures} = T,\\ \mathsf{signals} = S,\\ \mathsf{return} = R,\\ \mathsf{break} = B,\\ \mathsf{continue} = C\end{array}\right)\ \wedge\ \mathsf{SB}\cdot\left(\begin{array}{l}\mathsf{diverges} = \lambda x : \mathsf{M}.b,\\ \mathsf{requires} = T,\\ \mathsf{statement} = s_2,\\ \mathsf{ensures} = Q,\\ \mathsf{signals} = S,\\ \mathsf{return} = R,\\ \mathsf{break} = B,\\ \mathsf{continue} = C\end{array}\right)$$

$$\Longrightarrow$$

[57]Notice that we use a boolean constant instead of a predicate in the diverges clause of the Hoare n-tuples in Lemma 2. In most actual (to be verified) programs this will indeed be a constant value, but there is a special dedicated rule that turns an arbitrary diverges predicate into two cases: one true and one false.

$$\mathsf{SB} \cdot \begin{pmatrix} \mathsf{diverges} = \lambda x : \mathsf{M}.b, \\ \mathsf{requires} = P, \\ \mathsf{statement} = s_1; s_2, \\ \mathsf{ensures} = Q, \\ \mathsf{signals} = S, \\ \mathsf{return} = R, \\ \mathsf{break} = B, \\ \mathsf{continue} = C \end{pmatrix}$$

Proof (sketch). Three cases can be distinguished:

- Case s_1 diverges, then by the Definition 12 the composition also diverges.
- Case s_1 terminates normally, then there are again three cases:
  - Case s_2 diverges, then by the Definition 12 the composition also diverges.
  - Case s_2 terminates normally, then we have to prove that $s_1; s_2(x) = s_2(s_1(x))$ for all states x where s_1 followed by s_2 terminate normally. This follows from Definition 12.
  - Case s_2 terminates abnormal, then again by Definition 12 we know that the composition also terminates abnormally.
- Case s_1 terminates abnormal, then again by Definition 12 we know that the composition also terminates abnormally.

□

This ends the interlude. In the next section we will show how the LOOP Hoare rules can be extended to a relational setting and can be used to interactively prove non-interference properties for non-trivial Java programs.

7.3.2 Relational Hoare n-tuples

The Hoare n-tuples from the previous section can be extended to a relational setting: all predicates are replaced by a (binary) relation which allows us to reason about the relation between (memory) states in all possible different runs of a program. Once these relational n-tuples are properly defined we widen the relational Hoare logic from Figure 7.1 to full (sequential) Java in Section 7.4.

The Hoare n-tuples from Definition 13 can be extended to relational Hoare n tuples:

Definition 15 (Relational Hoare n-tuples for Java)**.**

$$
\begin{array}{l}
\mathsf{RelStatBehaviorSpec} \;:\; \mathsf{TYPE} \equiv \\
\quad [\;\; \mathsf{reldiverges} : (\mathsf{M} \times \mathsf{M}) \to \mathbb{B}, \\
\quad\quad \mathsf{relrequires} : (\mathsf{M} \times \mathsf{M}) \to \mathbb{B}, \\
\quad\quad \mathsf{statement} : \mathsf{M} \to \mathsf{StatResult}, \\
\quad\quad \mathsf{relensures} : (\mathsf{M} \times \mathsf{M}) \to \mathbb{B}, \\
\quad\quad \mathsf{relsignals} : (\mathsf{M} \times \mathsf{M}) \to (\mathsf{RefType} \times \mathsf{RefType}) \to \mathbb{B}, \\
\quad\quad \mathsf{relreturn} : (\mathsf{M} \times \mathsf{M}) \to \mathbb{B}, \\
\quad\quad \mathsf{relbreak} : (\mathsf{M} \times \mathsf{M}) \to (\mathsf{lift[string]} \times \mathsf{lift[string]}) \to \mathbb{B}, \\
\quad\quad \mathsf{relcontinue} : (\mathsf{M} \times \mathsf{M}) \to (\mathsf{lift[string]} \times \mathsf{lift[string]}) \to \mathbb{B}, \;\;]
\end{array}
$$

$$
\begin{array}{l}
\mathsf{RelExprBehaviorSpec[Out]} \;:\; \mathsf{TYPE} \equiv \\
\quad [\;\; \mathsf{reldiverges} : (\mathsf{M} \times \mathsf{M}) \to \mathbb{B}, \\
\quad\quad \mathsf{relrequires} : (\mathsf{M} \times \mathsf{M}) \to \mathbb{B}, \\
\quad\quad \mathsf{expression} : \mathsf{M} \to \mathsf{ExprResult[Out]}, \\
\quad\quad \mathsf{relensures} : (\mathsf{M} \times \mathsf{M}) \to (\mathsf{Out} \times \mathsf{Out}) \to \mathbb{B}, \\
\quad\quad \mathsf{relsignals} : (\mathsf{M} \times \mathsf{M}) \to (\mathsf{RefType} \times \mathsf{RefType}) \to \mathbb{B} \;\;]
\end{array}
$$

The functions RSB and REB, as given in Definition 16, relate the relational Hoare tuples $rsbs$: RelStatBehaviorSpec and $rebs$: RelExprBehaviorSpec to the underlying semantics. Notice that we do not uses a case-expression here (as in Definition 14) since this becomes unreadable due to the large number of nested cases.

Definition 16 (Semantic interpretation functions REB and RSB)**.**

$$
\begin{array}{l}
rebs : \mathsf{RelExprBehaviorSpec[Out]} \;\vdash\; \mathsf{REB} \cdot rebs : \mathbb{B} \;\equiv \\
\quad \forall x, y \in \mathsf{M} : rebs.\mathsf{relrequires} \cdot (x, y) \Rightarrow \\
\quad\quad \left(\begin{array}{ll}
\mathsf{hang?}(rebs.\mathsf{statement}(x)) & \wedge \\
\mathsf{hang?}(rebs.\mathsf{statement}(y)) & \wedge \\
rebs.\mathsf{reldiverges} \cdot (x, y) &
\end{array} \right) \\
\quad\quad \vee \\
\quad\quad \left(\begin{array}{ll}
\mathsf{norm?}(rebs.\mathsf{statement}(x)) & \wedge \\
\mathsf{norm?}(rebs.\mathsf{statement}(y)) & \wedge \\
rsbs.\mathsf{relensures} \cdot (x.\mathsf{ns}, y.\mathsf{ns}) \cdot (x.\mathsf{res}, y.\mathsf{res}) &
\end{array} \right) \\
\quad\quad \vee \\
\quad\quad \left(\begin{array}{ll}
\mathsf{abnorm?}((rsbs).\mathsf{statement}(x)) & \wedge \\
\mathsf{abnorm?}((rsbs).\mathsf{statement}(y)) & \wedge \\
\mathsf{excp?}(\mathsf{dev?}((rsbs).\mathsf{statement}(x))) & \wedge \\
\mathsf{excp?}(\mathsf{dev?}((rsbs).\mathsf{statement}(y))) & \wedge \\
rsbs.\mathsf{relsignals} \cdot (x.\mathsf{es}, y.\mathsf{es}) \cdot (x.\mathsf{ex}, y.\mathsf{ex}) &
\end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
rsbs : \mathsf{RelStatBehaviorSpec} \;\vdash\; \mathsf{RSB} \cdot rsbs : \mathbb{B} \;\equiv \\
\quad \forall x, y \in \mathsf{M} : rsbs.\mathsf{relrequires} \cdot (x, y) \Rightarrow
\end{array}
$$

$$
\begin{array}{l}
\left(\begin{array}{ll}
\mathsf{hang?}(rsbs.\mathsf{statement}(x)) & \wedge \\
\mathsf{hang?}(rsbs.\mathsf{statement}(y)) & \wedge \\
rsbs.\mathsf{reldiverges} \cdot (x, y) &
\end{array}\right) \\
\vee \\
\left(\begin{array}{ll}
\mathsf{norm?}(rsbs.\mathsf{statement}(x)) & \wedge \\
\mathsf{norm?}(rsbs.\mathsf{statement}(y)) & \wedge \\
rsbs.\mathsf{relensures} \cdot (x.\mathsf{ns}, y.\mathsf{ns}) &
\end{array}\right) \\
\vee \\
\left(\begin{array}{ll}
\mathsf{abnorm?}((rsbs).\mathsf{statement}(x)) & \wedge \\
\mathsf{abnorm?}((rsbs).\mathsf{statement}(y)) & \wedge \\
\mathsf{excp?}(\mathsf{dev?}((rsbs).\mathsf{statement}(x))) & \wedge \\
\mathsf{excp?}(\mathsf{dev?}((rsbs).\mathsf{statement}(y))) & \wedge \\
rsbs.\mathsf{relsignals} \cdot (x.\mathsf{es}, y.\mathsf{es}) \cdot (x.\mathsf{ex}, y.\mathsf{ex}) &
\end{array}\right) \\
\vee \\
\left(\begin{array}{ll}
\mathsf{abnorm?}((rsbs).\mathsf{statement}(x)) & \wedge \\
\mathsf{abnorm?}((rsbs).\mathsf{statement}(y)) & \wedge \\
\mathsf{return?}(\mathsf{dev?}((rsbs).\mathsf{statement}(x))) & \wedge \\
\mathsf{return?}(\mathsf{dev?}((rsbs).\mathsf{statement}(y))) & \wedge \\
rsbs.\mathsf{relreturn} \cdot (x.\mathsf{rs}, y.\mathsf{rs}) &
\end{array}\right) \\
\vee \\
\left(\begin{array}{ll}
\mathsf{abnorm?}((rsbs).\mathsf{statement}(x)) & \wedge \\
\mathsf{abnorm?}((rsbs).\mathsf{statement}(y)) & \wedge \\
\mathsf{break?}(\mathsf{dev?}((rsbs).\mathsf{statement}(x))) & \wedge \\
\mathsf{break?}(\mathsf{dev?}((rsbs).\mathsf{statement}(y))) & \wedge \\
rsbs.\mathsf{relbreak} \cdot (x.\mathsf{bs}, y.\mathsf{bs}) \cdot (x.\mathsf{blab}, y.\mathsf{blab}) &
\end{array}\right) \\
\vee \\
\left(\begin{array}{ll}
\mathsf{abnorm?}((rsbs).\mathsf{statement}(x)) & \wedge \\
\mathsf{abnorm?}((rsbs).\mathsf{statement}(y)) & \wedge \\
\mathsf{cont?}(\mathsf{dev?}((rsbs).\mathsf{statement}(x))) & \wedge \\
\mathsf{cont?}(\mathsf{dev?}((rsbs).\mathsf{statement}(y))) & \wedge \\
rsbs.\mathsf{relcontinue} \cdot (x.\mathsf{cs}, y.\mathsf{cs}) \cdot (x.\mathsf{clab}, y.\mathsf{clab}) &
\end{array}\right)
\end{array}
$$

We can use the functions RSB and REB to interpret the relational Hoare triples from Section 7.2. Thus if R and T are relations then in the extended setting the (partial) relational Hoare triple $\{R\}\ s\ \{T\}$ is equivalent to:

$$
\{R\}\ s\ \{T\} \ = \ \mathsf{RSB} \cdot \left(\begin{array}{l}
\mathsf{reldiverges} = \lambda x, y : \mathsf{M}.\mathsf{true} \\
\mathsf{relrequires} = R \\
\mathsf{statement} = s \\
\mathsf{relensures} = T \\
\mathsf{relsignals} = \lambda x, y : \mathsf{M}.\lambda e_1, e_2 : \mathsf{RefType}.\mathsf{true} \\
\mathsf{relreturn} = \lambda x, y : \mathsf{M}.\mathsf{true} \\
\mathsf{relbreak} = \lambda x, y : \mathsf{M}.\lambda l_1, l_2 : \mathsf{lift}[\mathsf{string}].\mathsf{true} \\
\mathsf{relcontinue} = \lambda x, y : \mathsf{M}.\lambda l_1, l_2 : \mathsf{lift}[\mathsf{string}].\mathsf{true}
\end{array}\right)
$$

Again, by changing all true values in false the partial correctness tuple above becomes a total correctness tuple. These relational Hoare tuples can also be used to express termination-sensitive non-interference for (full) sequential Java.

7.3.3 Termination sensitive non-interference as bisimulation

In order to be able to define termination-sensitive non-interference for sequential Java we give a coinductive definition of indistinguishability on memory locations:

Definition 17 (Indistinguishability)**.** I is the largest relation such that:

$$
\begin{array}{l}
\mathcal{I}(l_1, l_2) = \\
\quad \textsc{cases}\ \mathsf{Jtypeof}(l_1)\ \textsc{of}\ \{ \\
\quad\quad \mathsf{PrimType} \mapsto \textsc{cases}\ \mathsf{Jtypeof}(l_2)\ \textsc{of}\ \{ \\
\quad\quad\quad\quad \mathsf{PrimType} \mapsto l_1 = l_2 \\
\quad\quad\quad\quad \mid \mathsf{RefType} \mapsto \mathsf{false} \\
\quad\quad\quad\quad \mid \mathsf{ArrayType} \mapsto \mathsf{false}\ \} \\
\quad\quad \mid \mathsf{RefType} \mapsto \textsc{cases}\ \mathsf{Jtypeof}(l_2)\ \textsc{of}\ \{ \\
\quad\quad\quad\quad \mathsf{PrimType} \mapsto \mathsf{false} \\
\quad\quad\quad\quad \mid \mathsf{RefType} \mapsto \mathsf{class}(l_1) = \mathsf{class}(l_2) \wedge \\
\quad\quad\quad\quad\quad\quad \mathcal{I}(\mathsf{Jfieldsof}(l_1), \mathsf{Jfieldsof}(l_2)) \\
\quad\quad\quad\quad \mid \mathsf{ArrayType} \mapsto \mathsf{false}\ \} \\
\quad\quad \mid \mathsf{ArrayType} \mapsto \textsc{cases}\ \mathsf{Jtypeof}(l_2)\ \textsc{of}\ \{ \\
\quad\quad\quad\quad \mathsf{PrimType} \mapsto \mathsf{false} \\
\quad\quad\quad\quad \mid \mathsf{RefType} \mapsto \mathsf{false} \\
\quad\quad\quad\quad \mid \mathsf{ArrayType} \mapsto \mathsf{class}(l_1) = \mathsf{class}(l_2) \wedge \\
\quad\quad\quad\quad\quad\quad \mathsf{length}(l_1) = \mathsf{length}(l_2) \wedge \\
\quad\quad\quad\quad\quad\quad \mathsf{Jdim}(l_1) = \mathsf{Jdim}(l_2) \wedge \\
\quad\quad\quad\quad\quad\quad \mathcal{I}(\mathsf{Jindices}(l_1), \mathsf{Jindices}(l_2))\ \}\quad\quad \}
\end{array}
$$

Where the function Jtypeof *takes as argument a location in memory and returns a boolean. It distinguishes either* PrimType*,* RefType *or* ArrayType *depending on if the memory location was a primitive type, a (non-array) reference type or an array reference type. The function* class *returns the static class type of a reference, i.e., the same as Java's* `getClass()` *method, and it returns* `null` *if the reference is a null reference.* Jfieldsof *lists all fields of a reference. The function* length *returns the length,* Jdim *the dimension and* Jindices *lists all memory locations of the indexes of an array.*

Notice that the indistinguishability operator $\mathcal{I}$ is overloaded, as it is point-wise defined for all fields of an object and all array entries.

Termination-sensitive non-interference is then defined as:

Definition 18 (Termination-sensitive non-interference)**.** A program P is termination-sensitive non-interfering if the following Hoare n-tuple holds:

$$\begin{array}{l}\forall l : \mathsf{Loc}.\forall x, y : \mathsf{M}. \\ \quad \mathsf{sl}(l) = \mathsf{Low} \wedge \\ \quad \mathsf{RSB} \cdot \left(\begin{array}{lcl} \mathsf{reldiverges} & = & \mathcal{I}(x(l), y(l)) \\ \mathsf{relrequires} & = & \mathsf{true}, \\ \mathsf{statement} & = & P, \\ \mathsf{relensures} & = & \mathcal{I}((P.\mathsf{ns} \cdot x)(l), (P.\mathsf{ns} \cdot y)(l)) \\ \mathsf{relsignals} & = & \mathcal{I}((P.\mathsf{es} \cdot x)(l), (P.\mathsf{es} \cdot y)(l)) \\ \mathsf{relreturn} & = & \mathcal{I}((P.\mathsf{rs} \cdot x)(l), (P.\mathsf{rs} \cdot y)(l)) \\ \mathsf{relbreak} & = & \mathcal{I}((P.\mathsf{bs} \cdot x)(l), (P.\mathsf{bs} \cdot y)(l)) \\ \mathsf{relcontinue} & = & \mathcal{I}((P.\mathsf{cs} \cdot x)(l), (P.\mathsf{cs} \cdot y)(l)) \end{array} \right) \end{array}$$

In words: a program P is termination-sensitive non-interfering if, when the program terminates (either normally or abruptly), all memory locations with security level Low are indistinguishable. The same holds if the program does not terminate (loops), but then all Low memory locations have to be indistinguishable in the pre-state. If low memory locations are not indistinguishable for a certain termination mode, then the program may not terminate via that termination mode if non-interference is to be established.

We do not require that a relation (like R below) has to hold in the pre-state.

$$R \equiv \forall l : \mathsf{Loc}.\forall x, y : \mathsf{M}.\ \mathsf{sl}(l) = \mathsf{Low} \ \wedge \ \mathcal{I}(x(l), y(l))$$

Though this will usually be the case for a program that is 'freshly started'.

7.3.4 JML for relations: JMLrel

The Hoare logic n-tuples from Definition 13 directly correspond with JML's method specifications (e.g., such as the default one on page 12 in Chapter 2). The relational Hoare n-tuples from Definition 15 similarly correspond directly with a JML for relations. Below it is shown how an (interfering) JMLrel specification is related to a relational Hoare n-tuple. Notice that the default specification for one of the relational clauses is false[58], which ensures that (proved) specifications are total.

[58] Which differs from JML's default (in heavyweight specifications) in which the default spec is true, except for the diverges clause which is also false [LPC+05, §9.9].

JMLrel

```
int h,l;

//@ relrequires l(x) == l(y);
//@ relensures  l(x) != l(y);
public void m(){
     ...
 }
```

$$\mathsf{RSB} \cdot (\ \begin{array}{l} \mathsf{reldiverges} = \lambda x, y : \mathsf{M}.\ \mathsf{false}, \\ \mathsf{relrequires} = \lambda x, y : \mathsf{M}.\ x(l) = y(l), \\ \mathsf{statement} = \ldots, \\ \mathsf{relensures} = \lambda x, y : \mathsf{M}.\ x(l) \neq y(l), \\ \mathsf{relsignals} = \lambda x, y : \mathsf{M}. \lambda r_1, r_2 : \mathsf{RefType}.\ \mathsf{false}, \\ \mathsf{relreturn} = \lambda x, y : \mathsf{M}.\ \mathsf{false}, \\ \mathsf{relbreak} = \lambda x, y : \mathsf{M}. l_1, l_2 : \mathsf{lift}[\mathsf{string}].\ \mathsf{false}, \\ \mathsf{relcontinue} = \lambda x, y : \mathsf{M}. l_1, l_2 : \mathsf{lift}[\mathsf{string}].\ \mathsf{false}\ ) \end{array}$$

Where *field* `l` is stored in *memory location* l.

JMLrel should be considered as a convenient way of specifying relations for Java methods that is understandable for anybody who can understand JML specifications. We are not proposing to introduce a new specification language with its own tools, semantics and community backing. Its goal is to give a readable specification for the examples in Section 7.5.

7.4 A relational Hoare logic for Java

In this section the relational Hoare logic is illustrated by showing some example rules[59]. Note that all relational Hoare logic rules are actually (provable) lemma's, just like the (ordinary) Hoare rules described in the previous section.

7.4.1 Composition

The relational composition rule is provided here for easy comparison with the 'ordinary' Hoare logic rule from Lemma 2 and to illustrate that the rules are 'inherently' incomplete. Applying such a rule is similar to the use of ordinary Hoare rules. In ordinary Hoare rules the main challenge is to provide a suitable intermediate predicate (on the state space). The challenge now becomes to provide the correct intermediate relation (T in Lemma 3 below).

Lemma 3 (Relational Hoare rule for composition)**.**

[59]Due to space constraints we cannot show all (120+) rules. The relational Hoare logic rules in this section should be enough to appreciate the complexities involved.

$$\forall x, y : \mathsf{M}.\ \texttt{IF}\ (s_1 \cdot x).\mathsf{norm}\ \wedge\ (s_1 \cdot y).\mathsf{norm}\ \texttt{THEN}\ D((s_1 \cdot x).\mathsf{ns}, (s_1 \cdot y).\mathsf{ns}) \Rightarrow D(x, y)$$

$$\bigwedge$$

$$\exists T : \mathsf{M} \times \mathsf{M} \to \mathbb{B}.$$

$$\left[\ \mathsf{RSB} \cdot \begin{pmatrix} \mathsf{reldiverges} = D, \\ \mathsf{relrequires} = P, \\ \mathsf{statement} = s_1, \\ \mathsf{relensures} = T, \\ \mathsf{relsignals} = Qs, \\ \mathsf{relreturn} = Qr, \\ \mathsf{relbreak} = Qb, \\ \mathsf{relcontinue} = Qc \end{pmatrix} \wedge \mathsf{RSB} \cdot \begin{pmatrix} \mathsf{reldiverges} = D, \\ \mathsf{relrequires} = T, \\ \mathsf{statement} = s_2, \\ \mathsf{relensures} = Q, \\ \mathsf{relsignals} = Qs, \\ \mathsf{relreturn} = Qr, \\ \mathsf{relbreak} = Qb, \\ \mathsf{relcontinue} = Qc \end{pmatrix} \right]$$

$$\Longrightarrow$$

$$\left[\ \mathsf{RSB} \cdot \begin{pmatrix} \mathsf{reldiverges} = D, \\ \mathsf{relrequires} = P, \\ \mathsf{statement} = s_1; s_2, \\ \mathsf{relensures} = Q, \\ \mathsf{relsignals} = Qs, \\ \mathsf{relreturn} = Qr, \\ \mathsf{relbreak} = Qb, \\ \mathsf{relcontinue} = Qc \end{pmatrix} \right]$$

Proof sketch. Case distinctions on the termination modes of s_1 and s_2 using Definition 12. □

The composition rule above is a total correctness rule. First of all, notice the resemblance to the 'ordinary' Hoare logic rule from Lemma 2. The predicates are simply replaced by relations. This will actually work for most ordinary Hoare rules. Only in cases where a form of branching is possible we need to do something more involved, which is illustrated in subsequent sections.

On a more technical note, the first line of the rule above is needed because the reldiverges clause is evaluated in the pre-condition. In later sections we will replace the relation D with a boolean constant. In most actual cases this will indeed be a constant value, however it is possible (using a dedicated rule) to replace the constant back into a relation.

In general, the relational Hoare logic rules presented in this section are total and sound, but not complete:

Remark 2. *The relational Hoare logic is* not *complete for sequential Java.*

Completeness is not possible due to the different number of termination modes that can occur. Suppose that s_1 terminates normally in memory x and with an exception in y, if s_2 then throws the same exception in memory x then the termination mode of the

composition of s_1 and s_2 does not leak information. It is easy to see that one can construct a non-interfering program that uses this observation:

Java

```
int h; \\ High
int l; \\ Low

public void m() throws Exception {
   if (h > 1) {                          } s1
     l++; throw new Exception() }        }
   l++;                                  } s2
   throw new Exception();                }
}
```

This method is non-interfering, but the composition rule from Lemma 3 (or any of the other relational Hoare logic rules) cannot be used to prove this, hence the logic is not complete.

At first glance it might seem problematic that our rules are not applicable in all cases. Though completeness is hard to obtain in this context, it seems that the rules are somewhat too restrictive, especially if one considers that the (interactive) relational Hoare rules have a 'similar level of completeness' as the (automatic) type system [VS97a] of Volpano and Smith[60]. It turns out that both approaches can only prove non-interference in terms of strong bisimulations. Volpano and Smith's type system cannot type programs that only break the non-interference property (temporarily) inside a method body. Similarly a relational Hoare logic rule, like the one in Lemma 3, cannot be applied in cases where non-interference is temporarily broken.

In a sequential setting, weak bisimulations (as defined in Section 7.1) are more appropriate for expressing non-interference[61], since these allow temporary breaches of confidentiality (also described in Chapter 6). However, in practice, rules that can be applied in all situations are not workable. Such rules require keeping track of all possible *coupled* termination modes. Indeed, such a rule is required if we want to prove *at a syntactical level* that the example above is non-interfering. It turns out that it is far easier to mix syntactic reasoning with reasoning at a semantic level where necessary. This means that we can –temporarily– drop the relational Hoare logic entirely and fall back on the underlying semantics. Since the LOOP semantics in PVS form a shallow embedding, this is possible (and straightforward).

Of course, mixing syntactic reasoning with semantic reasoning has its own problems. For one thing, this restricts the use of the relational Hoare logic to a setting –such as the

[60] Though, the relational Hoare logic rules cover a much larger language (sequential Java) compared to Volpano and Smith's (while-like language).

[61] In a multi-threaded environment strong bisimulations are more natural (for expressing non-interference). Different threads may execute the same method at the same time which can leak (secret) information from inside a method body.

LOOP framework– where it is possible to reason a semantical level. Tools like ESC/Java2 or Jive [MMPH00], that only use purely syntactic Hoare logics, thus cannot prove non-interference as a weak bisimulation. The example in Section 7.5.1 illustrates switching between the syntactic level of the relational Hoare logic and the semantic level of the actual Java semantics.

If the termination behavior is restricted –by disallowing abrupt or non-termination– then a sound and complete relational composition rule can be constructed. Where soundness means that *if* confidentiality can be proved using the rule *then* confidentiality is indeed maintained, and completeness means that if confidentiality *cannot* be established using the rule then the analyzed program indeed does not maintain confidentiality.

Lemma 4 gives such a rule[62]:

Lemma 4 (Complete relational Hoare rule for composition)**.**

$$\exists T : \mathsf{M} \times \mathsf{M} \to \mathbb{B}.$$

$$\mathsf{RSB}\cdot\left(\begin{array}{l}\mathsf{reldiverges} = \lambda x, y : \mathsf{M}.\mathsf{false},\\ \mathsf{relrequires} = P,\\ \mathsf{statement} = s_1,\\ \mathsf{relensures} = T,\\ \mathsf{relsignals} = \lambda x, y : \mathsf{M}.\lambda e_1, e_2 : \mathsf{RefType}.\mathsf{false},\\ \mathsf{relreturn} = \lambda x, y : \mathsf{M}.\mathsf{false},\\ \mathsf{relbreak} = \lambda x, y : \mathsf{M}.\lambda l_1, l_2 : \mathsf{lift}[\mathsf{string}].\mathsf{false},\\ \mathsf{relcontinue} = \lambda x, y.\lambda l_1, l_2 : \mathsf{lift}[\mathsf{string}].\mathsf{false}\end{array}\right) \wedge \mathsf{RSB}\cdot\left(\begin{array}{l}\mathsf{reldiverges} = \lambda x, y : \mathsf{M}.\mathsf{false},\\ \mathsf{relrequires} = T,\\ \mathsf{statement} = s_2,\\ \mathsf{relensures} = Q,\\ \mathsf{relsignals} = \lambda x, y : \mathsf{M}.\lambda e_1, e_2 : \mathsf{RefType}.\mathsf{false},\\ \mathsf{relreturn} = \lambda x, y : \mathsf{M}.\mathsf{false},\\ \mathsf{relbreak} = \lambda x, y : \mathsf{M}.\lambda l_1, l_2 : \mathsf{lift}[\mathsf{string}].\mathsf{false},\\ \mathsf{relcontinue} = \lambda x, y : \mathsf{M}.\lambda l_1, l_2 : \mathsf{lift}[\mathsf{string}].\mathsf{false}\end{array}\right)$$

$$\iff$$

$$\mathsf{RSB}\cdot\left(\begin{array}{l}\mathsf{reldiverges} = \lambda x, y : \mathsf{M}.\mathsf{false},\\ \mathsf{relrequires} = P,\\ \mathsf{statement} = s_1; s_2,\\ \mathsf{relensures} = Q,\\ \mathsf{relsignals} = \lambda x, y : \mathsf{M}.\lambda e_1, e_2 : \mathsf{RefType}.\mathsf{false},\\ \mathsf{relreturn} = \lambda x, y : \mathsf{M}.\mathsf{false},\\ \mathsf{relbreak} = \lambda x, y : \mathsf{M}.\lambda l_1, l_2 : \mathsf{lift}[\mathsf{string}].\mathsf{false},\\ \mathsf{relcontinue} = \lambda x, y : \mathsf{M}.\lambda l_1, l_2 : \mathsf{lift}[\mathsf{string}].\mathsf{false}\end{array}\right)$$

Proof.

($\Rightarrow$) Trivial by Lemma 3.

($\Leftarrow$) Trivial by Definition 12.

□

In principal one can define a relational Hoare logic that uses relational Hoare logic rules that restrict the number of termination modes as in Lemma 4. Such a logic would be sound and complete, but it can still not be used to prove non-interference of the example on the previous page. In Section 7.5 it is shown how confidentiality can be proved for this example.

[62] Note that similar complete rules can be constructed for other termination modes as well.

7.4.2 If-then-else

The rule for `if-then-else` statements uses the same trick as in the simple relational Hoare logic from Section 7.2: only one of the branches is evaluated. Thus, in cases where both branches need to be analyzed, e.g., if the conditional depends on a variable that has security level High, the rule cannot be applied.

The actual rule is then given by the following lemma:

Lemma 5 (Relational Hoare rule for if-then-else statements)**.**

$$\begin{aligned}
&\forall x, y : \mathsf{M}.(\ (c \cdot x).\mathsf{norm}\ \wedge\ (c \cdot y).\mathsf{norm}\ \wedge\ P(x,y)\ ) \Rightarrow (c \cdot x).\mathsf{res} = (c \cdot y).\mathsf{res} \\
&\qquad\qquad \bigwedge \\
&\qquad \exists Rc : (\mathsf{M} \times \mathsf{M}) \to (\mathbb{B} \times \mathbb{B}) \to \mathbb{B}. \\
&\left[\ \mathsf{REB} \cdot \left(\begin{array}{l} \mathsf{reldiverges} = \lambda x, y : \mathsf{M}.b, \\ \mathsf{relrequires} = P, \\ \mathsf{expression} = c, \\ \mathsf{relensures} = \lambda x, y; \mathsf{M}. \lambda b_1, b_2 : \mathsf{bool}. Rc(x,y)(b_1,b_2), \\ \mathsf{relsignals} = Qs \end{array} \right) \right. \\
&\qquad\qquad \bigwedge \\
&\mathsf{RSB} \cdot \left(\begin{array}{l} \mathsf{reldiverges} = \lambda x, y : \mathsf{M}.b, \\ \mathsf{relrequires} = \lambda x, y : \mathsf{M}. Rc(x,y)(\mathsf{true},\mathsf{true}), \\ \mathsf{statement} = s_1, \\ \mathsf{relensures} = Q, \\ \mathsf{relsignals} = Qs, \\ \mathsf{relreturn} = Qr, \\ \mathsf{relbreak} = Qb, \\ \mathsf{relcontinue} = Qc \end{array} \right) \wedge \mathsf{RSB} \cdot \left(\begin{array}{l} \mathsf{reldiverges} = \lambda x, y : \mathsf{M}.b, \\ \mathsf{relrequires} = \lambda x, y : \mathsf{M}. Rc(x,y)(\mathsf{false},\mathsf{false}), \\ \mathsf{statement} = s_2, \\ \mathsf{relensures} = Q, \\ \mathsf{relsignals} = Qs, \\ \mathsf{relreturn} = Qr, \\ \mathsf{relbreak} = Qb, \\ \mathsf{relcontinue} = Qc \end{array} \right) \\
&\qquad\qquad \Longrightarrow \\
&\qquad\qquad \left. \mathsf{RSB} \cdot \left(\begin{array}{l} \mathsf{reldiverges} = \lambda x, y : \mathsf{M}.b, \\ \mathsf{relrequires} = P, \\ \mathsf{statement} = \texttt{if } c \texttt{ then } s_1 \texttt{ else } s_2, \\ \mathsf{relensures} = Q, \\ \mathsf{relsignals} = Qs, \\ \mathsf{relreturn} = Qr, \\ \mathsf{relbreak} = Qb, \\ \mathsf{relcontinue} = Qc \end{array} \right) \right]
\end{aligned}$$

Proof sketch. We distinguish two cases:

Case $(c \cdot x).\mathsf{res} = (c \cdot y).\mathsf{res} = \mathsf{true}$ We take as intermediate relation $Rc = P(x, y)$ and conclude this branch by the semantics of `if-then-else`.

Case $(c \cdot x).\mathsf{res} = (c \cdot y).\mathsf{res} = \mathsf{false}$ Analogous to the previous case.

□

This rule is only applicable in cases where the termination behavior of the conditional c is the same in memory x and y. Moreover, if c terminates normally then it has to evaluate to the same value in memory x and y, which will only be in cases where the conditional does

not depend on High variables. Notice that if this requirement is not met then (termination sensitive) non-interference is –at least locally– broken (and we can thus not prove a strong bisimulation property).

Also notice that in case the if-then-else statement terminates abnormally via an exception then –in order to prove termination-sensitive non-interference– we also have to prove that the raised exception objects are indistinguishable (as defined by the indistinguishability operator $\mathcal{I}$ from Definition 17).

7.4.3 Integer division

The rule for division of integers is interesting because a (new) exception object is thrown when a division by zero occurs. This implies that one can never naively divide by a variable of high security level, since this can leak if the high variable has value zero (or not). The rule is given in Lemma 6 below:

Lemma 6 (Relational Hoare rule for integer division)**.**

$$\exists T : (\mathsf{M} \times \mathsf{M}) \to (\mathsf{int} \times \mathsf{int}) \to \mathbb{B}.$$

$$\mathsf{REB} \cdot \left(\begin{array}{l} \mathsf{reldiverges} = \lambda x, y : \mathsf{M}.b, \\ \mathsf{relrequires} = P, \\ \mathsf{expression} = e_1, \\ \mathsf{relensures} = \lambda x, y : \mathsf{M}.\lambda j_1, j_2 : \mathsf{int}.T(x,y)(j_1,j_2), \\ \mathsf{relsignals} = Qs \end{array} \right)$$

$$\bigwedge$$

$$\forall i_1, i_2 : \mathsf{int}.\ \mathsf{REB} \cdot \left(\begin{array}{l} \mathsf{reldiverges} = \lambda x, y : \mathsf{M}.b, \\ \mathsf{relrequires} = \lambda x, y : \mathsf{M}.T(x,y)(i_1,i_2) \\ \mathsf{expression} = e_2, \\ \mathsf{relensures} = \quad \lambda x, y : \mathsf{M}, \lambda j_1, j_2 : \mathsf{int}. \\ \qquad \texttt{IF}\ (j_1 \neq 0 \wedge j_2 \neq 0) \\ \qquad \texttt{THEN}\ Qi(x,y)(\mathsf{div}(i_1,j_1) \wedge \mathsf{div}(i_2,j_2)) \\ \qquad \texttt{ELSE IF}\ (j_1 = 0 \wedge j_2 = 0) \\ \qquad\qquad \texttt{THEN} \\ \qquad\qquad\ \texttt{LET}\ \mathsf{ex}_1 : [\mathsf{M}, \mathsf{RefType}] = \\ \qquad\qquad\qquad \mathsf{make?ex}(\texttt{ArithmeticException})(x), \\ \qquad\qquad\quad \mathsf{ex}_2 : [\mathsf{M}, \mathsf{RefType}] = \\ \qquad\qquad\qquad \mathsf{make?ex}(\texttt{ArithmeticException})(y) \\ \qquad\qquad\ \texttt{IN}\ Qs(\pi_1(\mathsf{ex}_1), \pi_1(\mathsf{ex}_2))(\pi_2(\mathsf{ex}_1), \pi_2(\mathsf{ex}_2)) \\ \qquad\qquad \texttt{ELSE}\ \ \mathsf{false}, \\ \mathsf{relsignals} = Qs \end{array} \right)$$

$$\Longrightarrow$$

$$\mathsf{REB} \cdot \begin{pmatrix} \mathsf{reldiverges} = \lambda x, y : \mathsf{M}.b, \\ \mathsf{relrequires} = P, \\ \mathsf{expression} = e_1/e_2, \\ \mathsf{relensures} = Qi, \\ \mathsf{relsignals} = Qs \end{pmatrix}$$

Proof sketch. Figure 7.3 provides the proof tree (generated in PVS) of this proof. It is easy to recognize the main points:

- Starting at the root, the first branching point represent the termination behavior of expression e_1, with non-termination (left), abnormal termination (right) and normal termination (middle) of e_1.
- following the main branch, the next branching then represents the termination behavior of expression e_2, again non-termination (left), abnormal termination (right) and normal termination (middle).
- the next place the main branch splits is a case distinction on the result of expression e_2 being zero or non-zero (in both memory x and y).
- the remaining parts are all (semi) automatically handled by PVS.

□

Like before (in Section 4.3.2), the proof tree is displayed here to give the reader an impression of the complexity and number of interactions (each node corresponds to a user-command). The reader is not expected to analyze the information in each node in detail here.

The overall pattern should be familiar by now: (integer) division constitutes a branching rule of some kind, i.e., if one divides by zero or not. Thus the rule is constructed in such a way that it is only possible to apply it if at all times only one of 'both branches' is evaluated. In the context of non-interference this means that we cannot divide by a high variable since this can leak information to the outside (since high variables can have different values in memory x and y, and in particular, one could be zero and the other one not).

The rule also shows some details of the actual PVS formalization. The function make?ex takes a string as argument and returns a tuple that contains the new state and a reference to a (new) exception object (of the type indicated by the string argument). Furthermore, notice that the rule uses two different division operators: the Java operator "/" and the native PVS devision operator "div". The Java operator throws appropriate exceptions when necessary, i.e., an `ArithmeticException` exception when a division by zero occurs, and uses the native PVS operator otherwise. Overloading PVS's div operator allows us to use multiple representations of (Java) integers, such as bitvectors or PVS's internal (unbounded) integer type, see [Jac03] and [Bre06, Chapter 4] for details.

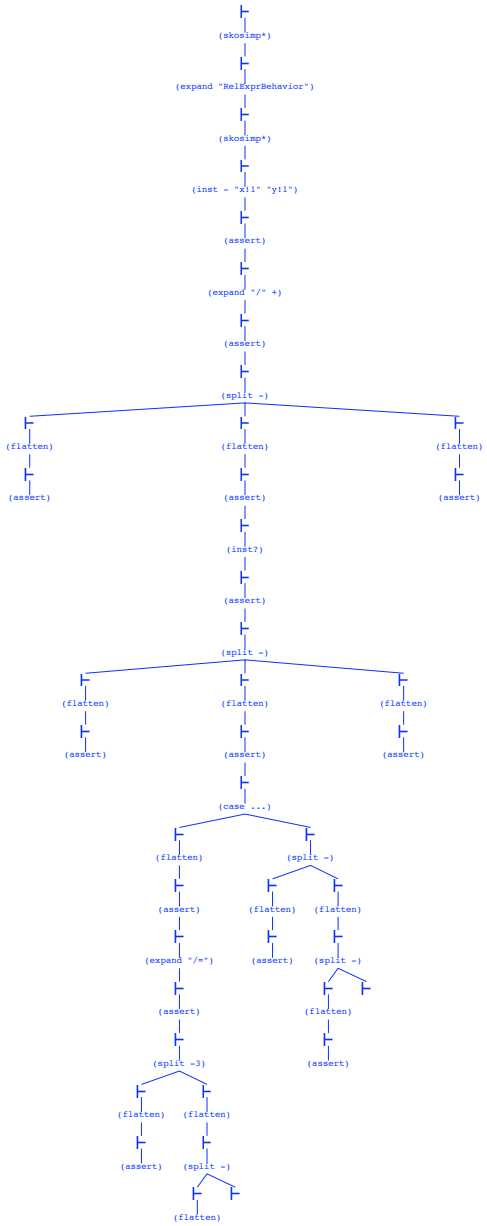

Figure 7.3: Soundness prove in PVS of the relational Hoare logic rule for integer division.

Another point to note is that the heaps in x and y both grow with one object, thus if the heaps were indistinguishable from the outside with respect to low memory locations in the pre-state of the method, then after application of this rule the heaps are again indistinguishable. In actual proofs we will often (implicitly) assume that the heaps are indistinguishable for all low memory locations. Since high memory locations are supposed

to be unobservable, these can be different. Indeed, the heaps do not need to have the same size[63] in both locations, i.e., in cases where an object is created under a high guard. We refer to this as *unbalanced heaps* (this issue has briefly been discussed in Section 6).

7.4.4 Throwing exceptions explicitly

There are two basic options when one designs a relational Hoare logic like the one explained in this section:

1. Make the logic as *general* as possible, this ensures that it can be used to prove a number of different properties.

2. Make the logic as *specific* to the task at hand as possible.

We have not chosen for either of these extremes, but the logic clearly leans more towards option 2 than to option 1 above. The rule for `throw` statements illustrates this[64]:

Lemma 7 (Throw relational Hoare rule)**.**

$$\forall x, y : \mathsf{M}.((e \cdot x).\mathsf{norm} \ \wedge \ (e \cdot y).\mathsf{norm} \ \wedge \ P(x,y)) \Rightarrow \mathcal{I}((e \cdot x).\mathsf{res}, (e \cdot y).\mathsf{res})$$

$$\bigwedge$$

$$\forall x, y : \mathsf{M}.((e \cdot x).\mathsf{abnorm} \ \wedge \ (e \cdot y).\mathsf{abnorm} \ \wedge \ P(x,y)) \Rightarrow \mathcal{I}((e \cdot x).\mathsf{ex}, (e \cdot y).\mathsf{ex})$$

$$\bigwedge$$

$$\mathsf{REB} \cdot \left(\begin{array}{l} \mathsf{reldiverges} = \lambda x, y : \mathsf{M}.b, \\ \mathsf{relrequires} = P, \\ \mathsf{expression} = e, \\ \mathsf{relensures} = \lambda x, y : \mathsf{M}.\lambda r_1, r_2 : \mathsf{RefType}. \\ \quad \mathtt{LET} \ \ \mathsf{ex}_1 \ = \ \mathsf{make?ex}(\mathtt{NullPointerException})(x), \\ \qquad\quad \mathsf{ex}_2 \ = \ \mathsf{make?ex}(\mathtt{NullPointerException})(y) \\ \quad \mathsf{IN} \\ \quad \textsc{cases}\ a\ \textsc{of}\ \{ \\ \qquad \mathtt{null} \qquad\quad \mapsto \ Qs(\pi_1(\mathsf{ex}_1), \pi_1(\mathsf{ex}_2))(\pi_2(\mathsf{ex}_1), \pi_2(\mathsf{ex}_2)) \\ \quad \mid \mathtt{reference}\ c \ \mapsto \ \textsc{cases}\ b\ \textsc{of}\ \{ \\ \qquad\qquad\qquad\qquad \mathtt{null} \qquad\quad \mapsto \ \mathsf{false}^{65} \\ \qquad\qquad\qquad\qquad \mid \mathtt{reference}\ d \ \mapsto \ Qs(x,y)(a,b)\ \}\ \}, \\ \mathsf{relsignals} = Qs\ \end{array} \right)$$

[63] Actually, the correct word here is *length* since the heap model in PVS is implemented as a list of lists of native Java data types (references and primitive types). Garbage collection has not been modeled in PVS, thus these (sub) lists can only grow longer.

[64] The composition rule from Lemma 3 is good example of one of the more general rules in our logic.

$$\Longrightarrow$$

$$\mathsf{RSB}\cdot\left(\begin{array}{l}\mathsf{reldiverges} = \lambda x, y : \mathsf{M}.b,\\ \mathsf{relrequires} = P,\\ \mathsf{statement} = \texttt{throw}\ e,\\ \mathsf{relensures} = Q,\\ \mathsf{relsignals} = Qs,\\ \mathsf{relreturn} = Qr,\\ \mathsf{relbreak} = Qb,\\ \mathsf{relcontinue} = Qc\ )\end{array}\right)$$

Proof. The proof is straightforward and has been omitted. □

The rule requires that both the return value of the expression e in memories x and y and the actual thrown object in memories x and y should be equivalent (indistinguishable) references for memories x and y. The indistinguishability relation from Definition 17 is used to ensure this. Customizing this rule for proving (termination-sensitive) non-interference simplifies the rule considerably.

Another point to notice is that this rule will always end in an exceptional state (as is required by the semantics of `throw` statements), the only difference is given by the arguments of the relation Qs.

7.4.5 Other rules

We have formulated many more relational Hoare rules in PVS and proved their correctness relative to LOOP's Java semantics. We cannot show them here, mainly because they are too involved –such as the rule for `try-catch-finally` and array access– or because they are not interesting –such as the many rules for integer arithmetic (`+,-,*`)– that are all almost the same, yet slightly different.

We especially want to mention the rule for `while` statements here, since it is very restrictive in a setting where termination-sensitive non-interference is important. It can only be applied in cases where one can find a strong bisimulation, i.e., a bisimulation that is valid in both memories x and y at the same time, which we believe will almost never be the case in practice. It is not surprising that in cases where even stronger forms of non-interference have to be proved (such as time-sensitive non-interference discussed in Chapter 8) it is standard to ignore loops altogether and simply require that the conditional of a `while` statement is typed minimally[66] (to borrow a notion from [VS97b]).

We believe that approximate approaches, such as discussed in Chapter 6 or in [AB05, Zan02], are better suited for dealing with loops. Another option is to simply use a standard type-checking algorithm that will indicate a possible breach of confidentiality if something is assigned to a Low variable when the context (the conditional of the loop) has security level High. This is equivalent to (dis)proving appropriate strong bisimulations for loops.

[65] Notice that this point is not reachable.

[66] An expression or statement is typed minimally (in our setting) if it has security level Low.

7.5 Examples

In this section we apply our framework to examples from the literature and of our own. These should illustrate the use of the relational Hoare logic. We will use JML on relations as specification language

7.5.1 A (partial) semantic proof

This example (from [DHS05]) illustrates mixing syntactic and semantic reasoning. Notice that relational JML clauses that are not explicitly specified default to false.

Java

```
int h,l;

//@ relrequires l(x) == l(y);
//@  relensures l(x) == l(y);
public void m(){
    if (h== 1) l=1; else l=0;
    //@ relassert true;
    l=0;
}
```

The idea here is to apply the composition rule with intermediate relation true and prove the two (trivial) remaining branches on a semantic level. Examples likes these can typically not be proved by type-based approaches, which will mark this method as possibly insecure. Notice that a strong bisimulation property cannot be proved here.

7.5.2 An example revisited

Recall the example form the previous section:

Java

```
class Test{

    int h,l;

    //@ relrequires l(x) == l(y);
    //@  relsignals (Exception e) l(x) == l(y) &&
    //@            l == \old(l) + 1 && I(e(x),e(y));
    public void m() {
        if (h > 1) {
          l++; throw new Exception() }
```

```
            //@ relassert \old(h) <= 1) ==>
            //@           l(x) == \old(l(x)) && l(y) == \old(l(y));
            l++;
            throw new Exception()
            }
    }
```

The proof starts with an application of the relational composition rule with the intermediate relation as indicated by the relational assertion. The first branch is than proved on a semantic level, since we cannot apply the rule for `if-then` which requires that the guard is minimally typed. The second branch can be proved by application of another composition rule and the throw rule. Notice that we have to prove that the exception objects are indistinguishable, which is trivial (if we assume that the (low) observable part of the heap was indistinguishable before method `m()` was called).

7.5.3 Simple arithmetic

This example (from [BDR04]) is non-interfering. It shows a perfect example of a –temporary– breach of confidentiality on a semantical level.

Java

```
int h,l;

//@ relrequires l(x) == l(y);
//@  relensures l(x) == l(y);
public void m() {
    l = l + h;
    //@ relassert l(x) == \old(l(x)) + \old(h(x)) &&
    //@           l(y) == \old(l(y)) + \old(h(y));
    l = l - h;
}
```

The thing to notice for this example is how the intermediate relation (indicated by the `relassert` keyword) no longer asserts how the low field `l` is related in both memories. From here relational Hoare rules for assignment and integer subtraction are applied which leads to the following relation:

```
l(x) == (\old(l(x)) + \old(h(x))) - \old(h(x)) &&
l(y) == (\old(l(y)) + \old(h(y))) - \old(h(y));
```

which is indeed equivalent to `l(x) == l(y)`.

This examples illustrates what we call a (temporary) breach of confidentiality and subsequent restoration of confidentiality *on a semantical level*. Program fragments like

these are very difficult to analyze automatically since they involve some semantical fact (here that `\old(h(y)) - \old(h(y)) == 0`) which is hard to check at a more abstract level. As a consequence, type-based approaches cannot prove that this program does not leak secret information and the same holds for approaches based on abstract interpretation (such as discussed in the Chapter 6).

7.5.4 Mixed termination modes

Java

```
int h,l;

//@ relrequires l(x) == l(y);
//@  relensures l(x) == l(y) && l = 5;
//@  relsignals (ArithmeticException ae) l(x) == l(y) &&
//@            l == \old(l) && I(ae(x),ae(y));
public void m() throws ArithmeticException{
    l = h / l;
    //@ relassert true;;
    l = 5;
}
```

This program is non-interfering, since no information about field `h` is leaked. Its behavior should be self-explanatory.

7.6 Related work

A lot of research has focused on secure information flow and non-interference. We will concentrate on closely related work here and refer to Section 2.5 for a general overview of the field.

Numerous papers on non-interference have been published that use some Hoare-like logic. We discuss the most important ones here.

Andrews and Reitman [AR80] were probably the first to use a Hoare-like logic for secure information flow. They consider a small concurrent while-language and give Hoare rules to prove a non-interference property. However they do not prove soundness of their rules.

Joshi and Leino [JL00] give an elegant definition of non-interference in terms of arbitrary assignments. Every statement S is non-interfering if the following equality holds: $\mathsf{S};\mathsf{HH} = \mathsf{HH};\mathsf{S};\mathsf{HH}$, where HH expresses that arbitrary values are assigned to all high variables. Their weakest pre-condition calculus can prove both termination sensitive (without abnormal termination) and termination insensitive forms of non-interference. However, they only consider a small while-like language.

Darvas, Hähnle and Sands [DHS05] use dynamic logic to prove secure information flow properties for (a subset of) JAVA CARD programs. They can prove both termination

insensitive and termination sensitive non-interference (though the latter is not illustrated on examples), but do not consider abnormal termination modes. They do give an example that involves objects.

Barthe, D'Argenio and Rezk [BDR04] propose a number of logics, including Hoare logic, Separation logic and Linear Temporal Logic, for proving non-interference. The main idea is to compose the program with itself and execute it again *in a different part of memory*. Then a non-interference relation has to hold. They claim that their Hoare logic is both sound and complete, which is true because they only consider termination insensitive non-interference. As we have shown, completeness is practically impossible in a setting with multiple termination modes where the goal is proving termination-sensitive non-interference.

Amtoft and Banerjee [AB05] present a Hoare logic for a small imperative language. Their approach is based on abstract interpretation [Cou96] and is less precise than the work cited above, but the logic can be applied automaticly and can provide counterexamples in cases where their approach encounters problems (either because the program is interfering or their analysis is too course).

More recently, Amtoft, Bandhakavi and Banerjee [ABB06] propose a Hoare-like logic for checking termination-insensitive non-interference for object-oriented programs. Under certain assumptions they can use an (automatic) algorithm for computing the strongest possible post-condition that ensures non-interference. JML-style (intermediate) assertions can also be used that allow a more fine-scaled interactive verification framework.

Benton [Ben04] gives a simple analysis and transformation system for while-programs that can be used to prove non-interference. He also gives a relational Hoare logic that can be used to prove that a pair of statements maps a given pre-relation into a post-relation. His logic only deals with a while-like language, but the logic is more general than ours, because he considers two statements. Of course, this last point does introduce a (form of) dual evaluation of statements, which is precisely something we wanted to prevent in the design of our relational Hoare logic.

Another relational logic we encountered in the literature is Yang's relational separation logic [Yan05]. Like Benton's logic, this one can prove relations for pairs of programs. Yang's logic works for a while language which includes lists and sets as basic data types. The paper does not show how to prove non-interference properties, but we expect that this is straightforward.

7.6.1 Relation to the LOOP framework

The relational Hoare logic has been formulated 'on top of' LOOP's Java semantics in PVS. This has a number of advantages:

- Proving soundness of the rules is easier since we work in a familiar environment (PVS) and have a 'stress tested' semantics for sequential Java.
- Aliasing and inheritance, which are handled by the LOOP compiler, are obtained 'for free'.

– The LOOP compiler can translate Java code into PVS theories. The (JMLRel) specifications still have to be written by hand, which –for now– is done by manually rewriting translated (standard) JML annotations. Implementing an automatic translation is left for future work.

For an overview of the LOOP project we refer to [JP04]. It suffices to note that the research presented in this chapter was only possible because of this pre-existing work.

7.7 Conclusions

We explained how non-interference can be expressed in terms of bisimulations for Java classes. A novel Hoare logic on relations has been introduced for proving confidentiality properties in the form of termination-sensitive non-interference. This logic is formalized in the theorem prover PVS on top the LOOP verification framework. Soundness of the logic has been proved as well and it is shown how completeness in actual verifications can be obtained by mixing (syntactic) reasoning on the relational Hoare logic level with semantic reasoning.

Besides the aforementioned automatic translation of JMLrel specifications there are a number of other challenges remaining for future work. A start has been made on automating the interactive proofs by developing powerful proof strategies (tactics) in PVS. This preliminary work can be extended further. Another question is if a relational weakest precondition calculus can be constructed (by lifting the 'ordinary' WP-calculus in the LOOP-framework [Jac04b]). It seems that this might be difficult, since mixing semantic and syntactic reasoning is less convenient in WP-calculi and formulating rules that are applicable in all situations does not look promising. Finally, to really 'stress-test' the logic a bigger verification case-study is required.

Chapter 8

Enforcing time-sensitive non-interference

This chapter extends the previous one, in the sense that it deals with termination-sensitive non-interference and adds one covert channel: timing behavior. The resulting form of non-interference is called *time-sensitive termination-sensitive non-interference*, and ensures termination-sensitive non-interference and the absence of timing channels. Such timing channels can be used to leak sensitive information: for example, Kocher [Koc96] and Bernstein [Ber04] showed that certain implementations of encryption algorithms can leak information about the used key via timing behavior. Unfortunately, timing leaks are hard to avoid by design, in particular because even the slightest difference in execution time can be made observable by putting it inside a loop, thereby potentially leaking secret information. They are also very hard to detect, and static enforcement mechanisms for non-interference do not consider timing channels.

The main trend in avoiding timing channels is to use a program transformation that transforms termination-insensitive non-interfering programs into time-sensitive termination-sensitive non-interfering programs. Supposedly, the following is a standard trick that is used to avoid timing leaks in (the smart card) industry:

Java

```
if (c) { a = a+b; }   ⟹   int[] x = new int[2];
                          x[0] = a;
                          x[1] = a+b;
                          a = x[c];   // boolean is 0 or 1 in c
```

Notice that the expressions `a` and `b` cannot have side-effects. In principle one can remove all side-effects from a program and use the transformation above to remove all timing leaks, but such (manual) transformations stay very ad hoc, and do not scale in practice.

More elaborate methodologies –most notably by Agat [Aga00b]– have been proposed for removing timing leaks in a more structured manner, however these are limited to programs

that only exploit a limited set of features.

The main result of this chapter,which is based on [BRW06], is a program transformation method that eliminates timing leaks in sequential object-oriented languages with exceptions. Our method for enforcing non-interference is based on (nested) transaction mechanisms [LMWF94]. The basic idea is to transform conditionals that depend on high expressions, i.e., expressions that depend on high variables, into conditionals in which each branch performs two transactions (one transaction for each branch in the original conditional statement), one of which is committed, namely the one that would have been executed in the original statement. Formally, the idea is to transform a branching statement of the form

$$\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2$$

into the statement

$$\begin{array}{l}\texttt{if } e \quad \texttt{then beginT; } s_2'\texttt{; abortT; beginT; } s_1'\texttt{; commitT}\\ \qquad\texttt{else beginT; } s_1'\texttt{; abortT; beginT; } s_2'\texttt{; commitT}\end{array}$$

where `beginT` starts a new transaction and `abortT` and `commitT` respectively aborts and commits a transaction, and s_1' and s_2' are respectively the statements s_1 and s_2 transformed in the same manner[67].

The proposed transformation offers several advantages. First of all, it is correct in the sense that termination-insensitive non-interfering programs are transformed into time-sensitive termination-sensitive non-interfering programs.

Second of all, the method is applicable to sequential object-oriented languages with exceptions and method calls, and therefore handles a fragment of the language that is significantly more expressive than the languages considered in previous works, see below. In particular, this is the first work considering dynamic object creation[68]. Furthermore, the transformation is applicable to structured languages and intermediate or low-level languages (but for the sake of clarity, we choose to present the transformation at source code level).

Third, the transformation is independent of the technique used to enforce termination-insensitive non-interference. More specifically, the transformation does not rely on the fact that programs are verified with an information flow type system as the one pioneered by Volpano and Smith's work [VS97b], with a specification pattern such as the one in Chapter 5, a static algorithm based on abstract interpretation as proposed in Chapter 6 or using a program logic as described in Chapter 7. The only requirement is that the methodology can prove termination-sensitive non-interference.

On a more negative side our translation raises several questions, which are discussed in Section 8.4.

[67] Notice that it is crucial that in both branches *first* the aborted block is executed and then the committed block. If one would first preform a committed code block and then the aborted one then the committed block might influence the aborted one via a side-effect and hence might influence execution time.

[68] Hedin and Sands [HS05] published, *at the same time as us*, another approach for removing timing leaks in object-oriented programs. It basically extends Agat [Aga00a] results to an object-oriented setting. However, for now, they do not support exceptions.

8.1 Language

We again use the sequential imperative language WHILE from Definition 1 (this time without side-effects in expressions) to explain the basic idea behind the timing-leaks removing program transformation. In Section 8.3 we extend this approach to an object-oriented programming language.

The operational semantics of the programs we use here is given by a small step operational semantics that captures one step execution of the program, and relates states, execution time and results. Notice that we need a small step operational semantics here because we want to model timing behavior of programs. In our setting results are simply values and we let Res denote the set of results. The set State of states is defined as the set of pairs of the form $\langle c, \rho\rangle$ where c is a statement, ρ is a mapping from local variables to values. We distinguish a special variable res to store results of execution of programs, as is defined below. Finally, the execution time of the program is modeled using a commutative monoid $(T, +, 0)$.

Formally, the operational semantics is defined through a relation $s \rightsquigarrow_t r$, where $s \in \mathsf{State}$, $r \in \mathsf{Res} \cup \mathsf{State}$ and $t \in T$, with intuitive meaning that s evaluates to r in time t. The closure $\rightsquigarrow_t^*$ is then defined inductively by the clauses:

- if $s \rightsquigarrow_t s'$ then $s \rightsquigarrow_t^* s'$;
- if $s \rightsquigarrow_t^* s'$ and $s' \rightsquigarrow_{t'}^* s''$ then $s \rightsquigarrow_{t+t'}^* s''$.

Finally, we define an evaluation relation $\Downarrow$ between states, results and execution time and set $\langle c, \rho\rangle \Downarrow_t v$ iff $\langle c, \rho\rangle \rightsquigarrow_t^* \langle \mathtt{skip}, \rho'\rangle$ with $\rho'(res) = v$ and $v \in \mathsf{Res}$. In the sequel, we often write $\langle P, \rho\rangle \Downarrow_t$ when the result of the evaluation is irrelevant, i.e., as a shorthand for $\exists r.\ \langle P, \rho\rangle \rightsquigarrow_t^* \langle \mathtt{skip}, \rho'\rangle \ \wedge\ \rho'(res) = r$. Further, for every function $f \in A \to B$, $x \in A$ and $v \in B$, we let $f \oplus \{x \mapsto y\}$ denote the unique function f' such that $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = v$.

The rules of the operational semantics are given in Figure 8.1. The rules are standard except for execution time, for which there are several possible models. In our model, each statement has its own execution time; for example, the execution time of $x := e$ is equal to the sum of the execution time of e and of some constant $t_{:=}$. More refined models allow the execution time of each instruction to be parametrized by the state, or even by execution history; for example, in the above example $t_{:=}$ would become a function. It is possible to extend our results to such execution models, by imposing suitable equational constraints on these functions.

The operational semantics of transactions is given in Figure 8.2. We assume that a beginT statement evaluates in time t_b, abortT evaluates in time t_a and commitT evaluates in time t_c. Note that transactions can be nested arbitrarily deep.

$$\frac{}{\langle x, \rho\rangle \rightsquigarrow_{t_{R_1}} \langle \rho(x), \rho\rangle}$$

$$\frac{v \ op \ v' = v''}{\langle v \ op \ v', \rho\rangle \rightsquigarrow_{t_{OP}} \langle v'', \rho\rangle}$$

$$\frac{\langle e_1, \rho\rangle \rightsquigarrow_t \langle e_1', \rho\rangle}{\langle e_1 \ op \ e_2, \rho\rangle \rightsquigarrow_t \langle e_1' \ op \ e_2, \rho\rangle}$$

$$\frac{\langle e_2, \rho\rangle \rightsquigarrow_t \langle e_2', \rho\rangle}{\langle v \ op \ e_2, \rho\rangle \rightsquigarrow_t \langle v \ op \ e_2', \rho\rangle}$$

$$\frac{}{\langle x := v, \rho\rangle \rightsquigarrow_{t_:=} \langle \texttt{skip}, \rho \oplus \{x \mapsto v\}\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle e', \rho\rangle}{\langle x := e, \rho\rangle \rightsquigarrow_t \langle x := e', \rho\rangle}$$

$$\frac{\langle s_1, \rho\rangle \rightsquigarrow_t \langle s_1', \rho'\rangle}{\langle s_1; s_2, \rho\rangle \rightsquigarrow_t \langle s_1'; s_2, \rho'\rangle}$$

$$\frac{}{\langle \texttt{skip}; s_2, \rho\rangle \rightsquigarrow_{t_{\texttt{skip}}} \langle s_2, \rho'\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle true, \rho\rangle}{\langle \texttt{while}(e)(s), \rho\rangle \rightsquigarrow_t \langle s; \texttt{while}(e)(s), \rho\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle false, \rho\rangle}{\langle \texttt{while}(e)(s), \rho\rangle \rightsquigarrow_t \langle \texttt{skip}, \rho\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle true, \rho\rangle}{\langle \texttt{if-then-else}(e)(s_1)(s_2), \rho\rangle \rightsquigarrow_t \langle s_1, \rho\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle false, \rho\rangle}{\langle \texttt{if-then-else}(e)(s_1)(s_2), \rho\rangle \rightsquigarrow_t \langle s_2, \rho\rangle}$$

$$\frac{\langle e, \rho\rangle \rightsquigarrow_t \langle e', \rho\rangle}{\langle \textsf{return } e, \rho\rangle \rightsquigarrow_t \langle \textsf{return } e', \rho\rangle}$$

$$\frac{}{\langle \textsf{return } v, \rho\rangle \rightsquigarrow_{t_R} \langle \texttt{skip}, \rho \oplus \{res \mapsto v\}\rangle}$$

Figure 8.1: Small step operational semantics for WHILE.

8.2 Transforming out timing leaks

Our method for transforming out timing leaks is based on (nested) transaction mechanisms [LMWF94]. Transactions allow a programmer to view code blocks as atomic and perform all updates inside such a transaction block as conditional. Only after an explicit *commit* statement is an update actually carried out. If the programmer desires so it is also possible to perform a roll-back to the state before the beginning of the transaction block via an explicit *abort* statement.

8.2.1 Problem statement and hypotheses

Since we are interested in ensuring (time-sensitive) non-interference, we assume a secure information flow policy that maps expressions, including program variables, to security levels High (secret) or Low (public). Formally, we assume a function sl that maps variables to security levels in the standard security lattice from Definition 2. A mapping from expressions to the security lattice is calculated by looking at the security level of all the

$$\frac{\langle c, \rho\rangle \rightsquigarrow^*_{t'} \langle \texttt{skip}, \rho'\rangle}{\langle \mathsf{beginT}; c; \mathsf{abortT}, \rho\rangle \rightsquigarrow_{t_{ba}+t'} \langle \texttt{skip}, \rho\rangle}$$

$$\frac{\langle c, \rho\rangle \rightsquigarrow^*_{t'} \langle \texttt{skip}, \rho'\rangle}{\langle \mathsf{beginT}; c; \mathsf{commitT}, \rho\rangle \rightsquigarrow_{t_{bc}+t'} \langle \texttt{skip}, \rho'\rangle}$$

Where $t_{ba} = t_b + t_a$ and $t_{bc} = t_b + t_c$ and we assume $t_a = t_c$

Figure 8.2: Operational semantics for transaction mechanisms.

individual variables. The maximum of the individual variables is then the security level of the whole expression[69].

Timing leaks in a program might occur when the branches of a conditional statement take different execution times. In our language, branches in execution are caused either by `if-then-else` or `while` statements. Due to our execution model, timing leaks can only occur when the conditional statement makes its test on a high expression. We only address the case of `if-then-else` statements that branch over high expressions, and restrict ourselves to low-recursive programs, where a program P is low-recursive iff it does not contain a statement of the form $\texttt{while}(e)(s)$ with $\mathsf{sl}(e) = \mathsf{High}$. While the restriction to low-recursive programs is rather severe, it is a standard assumption in the literature, going back to the works of Volpano and Smith [VS97b] and Agat [Aga00a, Aga00c].

8.2.2 The transformation

In order to avoid timing leaks caused by `if-then-else` statements, we use transactions to execute both branches of the statement. Thanks to an appropriate use of committing and aborting transactions, the transformation is semantics-preserving up to termination.

The transformation of `if-then-else` statements is given in Definition 19. The capitalized if-then-else in the translation –**IF**, **THEN** and **ELSE**– belongs to the translation and not to the programming language, and is introduced so that we only use transactions for `if-then-else` blocks with a high conditional. For all other cases, the transformation is defined by the obvious recursive clause[70].

[69] Note that this is an over-approximation. An expression like $h - h$, which should have security level Low (since the result is always zero), is thus classified as High. However, this naive approach will suffice for our purpose.

[70] Notice that –for reasons of readability— the syntax of `if-then-else` statements in WHILE has changed in the sequel, i.e., $\texttt{if-then-else}(e)(s_1)(s_2)$ equals `if` e `then` s_1 `else` s_2.

Definition 19 (Transformation $\mathcal{T}$ for removing timing leaks)**.**

$$
\begin{array}{l}
\mathcal{T}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2) \;= \\
\quad \textbf{IF } \mathsf{sl}(e) = \mathsf{Low} \textbf{ THEN } \texttt{if } e \texttt{ then } \mathcal{T}(s_1) \texttt{ else } \mathcal{T}(s_2) \\
\quad \textbf{ELSE } \texttt{if } e \texttt{ then} \\
\qquad \texttt{beginT;}\mathcal{T}(s_2)\texttt{;abortT;beginT;}\mathcal{T}(s_1)\texttt{;commitT; else} \\
\qquad \texttt{beginT;}\mathcal{T}(s_1)\texttt{;abortT;beginT;}\mathcal{T}(s_2)\texttt{;commitT;}
\end{array}
$$

The transformation is semantics preserving up to termination of the transformed program. Indeed, the transformation may introduce non-termination, as illustrated by the program fragments below (where h$\in$ High):

Java

```
if ( h > 0 ) {
  if ( h < 0 ) {
    while (true);
  } else ;
} else ;
```

```
if (h != h) {
      while(true);
} else;
```

Nevertheless we can prove some interesting properties concerning our translation. We first introduce an equality $\simeq$ on memories, and say $\rho \simeq \rho'$ holds iff $\rho(x) = \rho'(x)$ for all variables x where $\mathsf{sl}(x) = \mathsf{Low}$:

Lemma 8. *For every program P, memory ρ, results $r, r' \in \mathsf{Res}$, and times $t, t' \in T$ such that $\langle P, \rho\rangle \Downarrow_t r$ and $\langle \mathcal{T}(P), \rho\rangle \Downarrow_{t'} r'$, we have $r = r'$.*

Proof. We prove that for every statement s, memory ρ, results $r, r' \in \mathsf{Res}$, and times $t, t' \in T$ such that $\langle s, \rho\rangle \Downarrow_t r$ and $\langle \mathcal{T}(c), \rho\rangle \Downarrow_{t'} r'$, we have $r = r'$.

The proof proceeds by induction on the structure of c. It is straightforward and omitted. □

8.2.3 Application to non-interference

This section shows that our transformation maps low-recursive non-interfering programs into time-sensitive termination-sensitive non-interfering programs.

Before establishing these results, we give a –by now familiar– definition of non-interference. The definition is again adapted to the current context; in the sequel $\Uparrow$ denotes non-termination.

Definition 20 (Non-interference)**.**

1. A program P is termination-insensitive non-interfering if

$$
\begin{array}{l}
\forall \rho, \rho' : \mathcal{X} \to \mathcal{V}.\ \forall v, v' : \mathcal{V}.\ \forall t, t' : \mathbb{N}. \\
\quad (\langle P, \rho\rangle \Downarrow_t v \ \wedge\ \rho \simeq \rho') \Rightarrow ((\langle P, \rho'\rangle \Downarrow_{t'} v' \ \wedge\ v = v') \ \vee\ (\langle P, \rho'\rangle \Uparrow))
\end{array}
$$

2. A program P is termination-sensitive non-interfering if

$$\forall \rho, \rho' : \mathcal{X} \to \mathcal{V}.\ \forall v, v' : \mathcal{V}.\ \forall t, t' : \mathbb{N}.\\ (\langle P, \rho\rangle \Downarrow_t v \ \wedge\ \rho \simeq \rho') \Rightarrow (\langle P, \rho'\rangle \Downarrow_{t'} v' \ \wedge\ v = v')$$

3. A program P is time-sensitive termination-sensitive non-interfering if

$$\forall \rho, \rho' : \mathcal{X} \to \mathcal{V}.\ \forall v, v' : \mathcal{V}.\ \forall t : \mathbb{N}.\\ (\langle P, \rho\rangle \Downarrow_t v \ \wedge\ \rho \simeq \rho') \Rightarrow (\langle P, \rho'\rangle \Downarrow_t v' \ \wedge\ v = v')$$

The difference between termination-insensitive and termination-sensitive non-interference is that the former only compares execution traces that terminate while the latter requires that the termination of the program is uniform in the high part of the memory. Note that time-sensitive termination-sensitive non-interference implies termination-sensitive non-interference, time-sensitive termination-sensitive non-interference imposes moreover that the execution time is uniform in the high part of the memory.

Since the transformation is semantic-preserving up to termination, we prove that our transformation preserves termination-insensitive non-interference.

Corollary 2. *For every low-recursive program P, if P is termination insensitive non-interfering, then $\mathcal{T}(P)$ is also termination insensitive non-interfering.*

Proof. Straightforward by Lemma 8. □

The transformation eliminates from programs timing leaks due to high if-then-else statements, provided that the program is non-interfering and low-recursive.

Theorem 2. *For all low-recursive non-interfering programs P, and memories ρ and ρ' such that $\rho \simeq \rho'$, we have $\langle \mathcal{T}(P), \rho\rangle \Downarrow_t$ iff $\langle \mathcal{T}(P), \rho'\rangle \Downarrow_t$.*

Proof. We prove that for all low-recursive statements s, and memories ρ and ρ' such that $\rho \simeq \rho'$, we have $\langle \mathcal{T}(s), \rho\rangle \rightsquigarrow_t^* \langle \texttt{skip}, \rho_1\rangle$ iff $\langle \mathcal{T}(s), \rho'\rangle \rightsquigarrow_t^* \langle \texttt{skip}, \rho_1'\rangle$. The proof proceeds by structural induction on statements.

Case s = if e then s_1 else s_2

– Suppose $\mathsf{sl}(e) = \mathsf{Low}$ then

$$\mathcal{T}(s) \ =\ \text{if } e \text{ then } \mathcal{T}(s_1) \text{ else } \mathcal{T}(s_2)$$

and by the IH we have that there exists a t such that, $\langle \mathcal{T}(s_1), \rho\rangle \rightsquigarrow_t^* \langle \texttt{skip}, \rho_1\rangle$ iff $\langle \mathcal{T}(s_1), \rho'\rangle \rightsquigarrow_t^* \langle \texttt{skip}, \rho_1'\rangle$ and the same holds for s_2. Since $\mathsf{sl}(e) = \mathsf{Low}$ and $\rho \simeq \rho'$, by our operational semantics e evaluates to the same value in memory ρ and ρ' and we conclude.

– Suppose $\mathsf{sl}(e) \neq \mathsf{Low}$ then

$$\begin{aligned}\mathcal{T}(s) = \ & \mathsf{if}\ e\ \mathsf{then} \\ & \quad \texttt{beginT;}\mathcal{T}(s_2)\texttt{;abortT;beginT;}\mathcal{T}(s_1)\texttt{;commitT;} \\ & \mathsf{else} \\ & \quad \texttt{beginT;}\mathcal{T}(s_1)\texttt{;abortT;beginT;}\mathcal{T}(s_2)\texttt{;commitT;}\end{aligned}$$

There are again two cases:

– $\langle e, \rho\rangle \Downarrow_{te} v \ \wedge\ \langle e, \rho'\rangle \Downarrow_{te} v' \ \wedge\ v = v'$ which follows directly from the IH.
– $\langle e, \rho\rangle \Downarrow_{te} v \ \wedge\ \langle e, \rho'\rangle \Downarrow_{te} v' \ \wedge\ v \neq v'$ in which case we have to prove that:

$$\begin{aligned}&\langle \texttt{beginT;}\mathcal{T}(s_2)\texttt{;abortT;beginT;}\mathcal{T}(s_1)\texttt{;commitT;}, \rho\rangle \rightsquigarrow_t^* \langle \texttt{skip}, \rho_1\rangle \\ &\qquad\qquad \Leftrightarrow \\ &\langle \texttt{beginT;}\mathcal{T}(s_1)\texttt{;abortT;beginT;}\mathcal{T}(s_2)\texttt{;commitT;}, \rho'\rangle \rightsquigarrow_t^* \langle \texttt{skip}, \rho_1'\rangle\end{aligned}$$

(⇒) Suppose that

$$\langle \texttt{beginT;}\mathcal{T}(s_2)\texttt{;abortT;beginT;}\mathcal{T}(s_1)\texttt{;commitT;}, \rho\rangle \rightsquigarrow_t^* \langle \texttt{skip}, \rho_1\rangle$$

where $\langle \mathcal{T}(s_2), \rho\rangle \rightsquigarrow_{t_2}^* \langle \texttt{skip}, \rho_2\rangle$ and $\langle \mathcal{T}(s_1), \rho\rangle \rightsquigarrow_{t_1}^* \langle \texttt{skip}, \rho_2'\rangle$ and $t = t_1 + t_2 + t_{ba} + t_{bc}$. By the IH $\langle \mathcal{T}(s_2), \rho'\rangle \rightsquigarrow_{t_2}^* \langle \texttt{skip}, \rho_3\rangle$ and $\langle \mathcal{T}(s_1), \rho'\rangle \rightsquigarrow_{t_1}^* \langle \texttt{skip}, \rho_3'\rangle$ hence we are done.

(⇐) Analogous to previous case.

Case $s = \mathsf{while}\ e\ \mathsf{do}\ s_1$ Because the program is low-recursive, we know that $\mathsf{sl}(e) = \mathsf{Low}$ and since $\rho \simeq \rho'$, by our operational semantics e evaluates to the same value in ρ and ρ'. By the IH $\langle \mathcal{T}(s_1), \rho\rangle \rightsquigarrow_{t_1}^* \langle \texttt{skip}, \rho_1\rangle$ iff $\langle \mathcal{T}(s_1), \rho'\rangle \rightsquigarrow_{t_1}^* \langle \texttt{skip}, \rho_1'\rangle$ holds. Since the program is non-interfering, we have that $\rho_1 \simeq \rho_1'$ and we can conclude.

Assignment, return and skip statements are trivial. The case for $s_1; s_2$ is straightforward by the IH.

□

As a corollary of this result we prove that our transformation also transforms termination-insensitive non-interfering programs into (time-sensitive) termination-sensitive non-interfering programs.

Corollary 3. *For all low-recursive and termination-insensitive non-interfering programs P we have that $\mathcal{T}(P)$ is time-sensitive termination-insensitive non-interfering.*

Proof. By Corollary 2 we have that $\mathcal{T}(P)$ is termination-insensitive non-interfering. And by Theorem 2, for all low-recursive and termination-insensitive non-interfering programs P, and memories ρ and ρ' such that $\rho \simeq \rho'$, we have $\langle \mathcal{T}(P), \rho\rangle \Downarrow_t$ iff $\langle \mathcal{T}(P), \rho'\rangle \Downarrow_t$. □

8.2.4 Enforcing termination-sensitive non-interference

Our transformation yields programs that are time-sensitive termination-sensitive non-interfering provided the input programs are termination-insensitive non-interfering programs and low-recursive. In this section, we examine how the latter properties may be enforced on programs.

Volpano and Smith [VS97b] provide a sound information flow type system for a simple imperative language. They use minimal typing for loop-conditionals and expressions which can (possibly) throw exceptions to enforce the termination sensitive form of non-interference, i.e., if the security types of these conditionals and expressions are minimal, in terms of the security level, then the termination behavior will only depend on low variables, thereby ensuring that termination behavior of a program cannot leak information. Volpano and Smith prove an additional lemma which states that if all conditionals and expressions that can throw exceptions are typed minimally then the program will be time-sensitive termination-sensitive non-interfering.

The main disadvantage of this approach is that information flow type systems are very restrictive, and reject many secure programs, even for simple programming languages. The techniques discussed in the previous chapters allow more precise analysis of programs.

8.3 Adding objects, methods and exceptions

8.3.1 Language

We extend our simple imperative language with objects, methods (without dynamic dispatch) and an exception mechanism as in Java, and we call the resulting language OO. The sets Expr of *expressions* and Stat of *statements* are given by the syntaxes in Definition 21.

An OO program P comes equipped with a set $\mathcal{C}$ of class names, including a class Throwable of exceptions, a set $\mathcal{F}$ of field names, and a set of method declarations of the form $m(\vec{x}) := c$, where m is a method name, $\vec{x}$ is a vector of variables (method formal parameters), and c is a statement in Stat. For every program in OO, we distinguish a main method namely main.

Definition 21 (The language OO)**.**

$$\begin{array}{rcl} e & ::= & x \mid n \mid e_1 \; op \; e_2 \mid e.f \mid \texttt{new}\; C \mid e.m(\vec{e}) \mid (C)e \\ c & ::= & x := e \mid s_1; s_2 \mid \texttt{while}\; e\; \texttt{do}\; c \mid \texttt{if}\; e\; \texttt{then}\; s_1\; \texttt{else}\; s_2 \mid \\ & & e.f := e \mid \texttt{return}\; e \mid \texttt{skip} \mid \texttt{try}\; s_1\; \texttt{catch(Exception x)}\; \texttt{s}_2 \mid \texttt{throw e} \end{array}$$

where *op* is either a primitive operation $+, \times$, a comparison operation $<, \leq, =$, or the (unconditional) boolean connectives `|` and `&` and `skip` is the empty program (skip).

The operational semantics of the language is given in the appendix[71] in Section B.1; due

[71] The semantics of OO is presented in the appendix for completeness sake. This technical detail is not strictly necessary to understand the current chapter and we believe that it would only blur the main issues presented here.

to the presence of exceptions, performing one-step execution of a statement may either lead to a normal state, or to an exceptional state. For the sake of simplicity, we assume that the only statements that may raise exceptions in our language are $x.f := e$, $x := e.f$ (where e cannot throw exceptions), and an explicit `throw` statement.

The operational semantics is used to define an evaluation relation that relates programs, memories, results, and execution times.

8.3.2 Problem statement

In our extended language, timing leaks may occur due to explicitly or implicitly thrown exceptions.

We require that all exceptions that are thrown under a high security level are surrounded by a `try-catch` block. In order to write a correct transformation, we also need to assume that high exceptions (i.e., exceptions thrown by a statement containing $e.f$ where $e.f$ is a high expression) are handled in the same method where the exceptions are thrown, i.e., high exceptions cannot be propagated. The same applies to `throw` statements inside influence of high conditionals.

8.3.3 The transformation

Definition 22 shows how we extend the translation $\mathcal{T}$ from Definition 19 to allow for objects and exceptions. The transformation $\mathcal{T}$ uses a tail recursive function $\mathcal{T}_1$, shown[72] in Definition 23. Intuitively $\mathcal{T}_1$ transforms every high statement that might throw an exception into a set of statements without timing leaks.

Definition 22 (Transformation $\mathcal{T}$ for OO)**.**

$$
\begin{array}{lll}
\mathcal{T}(\texttt{try } s_1 \texttt{ catch(Exception } x\texttt{) } s_2) & = \mathcal{T}_1(\texttt{skip}, s_1, s_2, x) & (1)\\
\mathcal{T}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2) & = & \\
 & \quad \textbf{IF } \mathsf{sl}(e) = \mathsf{Low} \textbf{ THEN } \texttt{if } e \texttt{ then } \mathcal{T}(s_1) \texttt{ else } \mathcal{T}(s_2) & \\
 & \quad \textbf{ELSE } \texttt{ if } e \texttt{ then beginT;}\mathcal{T}(s_2)\texttt{;abortT;} & \\
 & \qquad\qquad\qquad \texttt{beginT;}\mathcal{T}(s_1)\texttt{;commitT;} & \\
 & \qquad\qquad \texttt{else beginT;}\mathcal{T}(s_1)\texttt{;abortT;} & \\
 & \qquad\qquad\qquad \texttt{beginT;}\mathcal{T}(s_2)\texttt{;commitT;} & \\
\mathcal{T}(\texttt{while } e \texttt{ do } c) & = \texttt{ while } e \texttt{ do } \mathcal{T}(c) & \\
\mathcal{T}(s_1; s_2) & = \mathcal{T}(s_1); \mathcal{T}(s_2) &
\end{array}
$$

Transformation $\mathcal{T}$ for other statements is defined as the identity.

In the transformation $\mathcal{T}_1(s_0, s_1, s_2, x)$, argument s_0 is a partial result of sequence of statements that either do not depend of high variables or that depend of high variables but that have been transformed in a sequence of timing-leaks free statements; its second argument s_1 represents statements in the original sequence of statements of a try part of

[72] Both definitions contain numbers between parentheses on the right, these are used in an example program transformation (in Figure 8.3) and can be ignored for now.

a `try-catch` statement that has to be transformed into a sequence of timing-leaks free statements; the third argument s_2 corresponds to the original statement in the catch part of a `try-catch` statement, and x is its variable.

Definition 23 (Function $\mathcal{T}_1$)**.**

$$\mathcal{T}_1(s_0, \texttt{skip}, s_3, x) \;=\; \texttt{try}\ s_0\ \texttt{catch(Exception}\ x)\ s_3 \tag{4}$$

$$\begin{array}{l}
\mathcal{T}_1(s_0, x' := e.f; s_2, s_3, x) \;= \\
\quad \textbf{IF}\ \mathsf{sl}(x') \;=\; \mathsf{Low} \\
\quad \textbf{THEN}\ \mathcal{T}_1(s_0; x' := e.f, s_2, s_3, x) \\
\quad \textbf{ELSE}\ \mathcal{T}_1(s_0; s'_1, s_2, s_3, x) \\
\quad\ \ \textit{where}\ s'_1 = \\
\qquad \texttt{if}\ e == \texttt{null then} \\
\qquad\quad \texttt{beginT}; s_3; \texttt{commitT}; \\
\qquad\quad \texttt{beginT}; x' := e; x' := (\texttt{new}\ C).f; \texttt{abortT}; \\
\qquad \texttt{else} \\
\qquad\quad \texttt{beginT}; s_3; \texttt{abortT}; \\
\qquad\quad \texttt{beginT}; x' := \texttt{new}\ C; x' := e.f; \texttt{commitT};
\end{array} \tag{3}$$

$$\mathcal{T}_1(s_0, (\texttt{while}\ e\ \texttt{do}\ s'_1); s_2, s_3, x) \;=\; \mathcal{T}_1(s_0; \texttt{while}\ e\ \texttt{do}\ \mathcal{T}_1(\texttt{skip}, s'_1, s_3, x), s_2, s_3, x)$$

$$\begin{array}{l}
\mathcal{T}_1(s_0, (\texttt{if}\ e\ \texttt{then}\ s'_1\ \texttt{else}\ s'_2); s_2, s_3, x) \;= \\
\quad \textbf{IF}\ \mathsf{sl}(e) = \mathsf{Low} \\
\quad \textbf{THEN}\ \mathcal{T}_1(s_0; \texttt{if}\ e\ \texttt{then}\ \mathcal{T}_1(\texttt{skip}, s'_1, s_3, x) \\
\qquad\qquad\qquad \texttt{else}\ \mathcal{T}_1(\texttt{skip}, s'_2, s_3, x), s_2, s_3, x) \\
\quad \textbf{ELSE}\ \mathcal{T}_1(s_0;\ \texttt{if}\ e\ \texttt{then} \\
\qquad\qquad\qquad\ \ \texttt{beginT;}\mathcal{T}_1(\texttt{skip}, s'_2, s_3, x)\texttt{;abortT;} \\
\qquad\qquad\qquad\ \ \texttt{beginT;}\mathcal{T}_1(\texttt{skip}, s'_1, s_3, x)\texttt{;commitT;} \\
\qquad\qquad\qquad \texttt{else} \\
\qquad\qquad\qquad\ \ \texttt{beginT;}\mathcal{T}_1(\texttt{skip}, s'_1, s_3, x)\texttt{;abortT;} \\
\qquad\qquad\qquad\ \ \texttt{beginT;}\mathcal{T}_1(\texttt{skip}, s'_2, s_3, x)\texttt{;commitT;}, s_2, s_3, x)
\end{array}$$

$$\begin{array}{l}
\mathcal{T}_1(s_0, x'.f := e; s_2, , s_3, x) \;= \\
\quad \textbf{IF}\ \mathsf{sl}(x'.f) = \mathsf{Low} \\
\quad \textbf{THEN}\ \mathcal{T}_1(\texttt{skip}, x'.f := e, s_3, x) \\
\quad \textbf{ELSE}\ \mathcal{T}_1(s_0; s'_1, s_2, s_3, x) \\
\quad\ \ \textit{where}\ s'_1 = \\
\qquad \texttt{if}\ e == \texttt{null then} \\
\qquad\quad \texttt{beginT}; s_3; \texttt{commitT}; \\
\qquad\quad \texttt{beginT}; \texttt{beginT}; x' := \texttt{new}\ C; \texttt{commitT}; x'.f := e; \texttt{abortT}; \\
\qquad \texttt{else} \\
\qquad\quad \texttt{beginT}; \texttt{beginT}; x' := \texttt{new}\ C; \texttt{abortT}; s_3; \texttt{abortT}; \\
\qquad\quad \texttt{beginT}; x'.f := e; \texttt{commitT}
\end{array}$$

$$\mathcal{T}_1(s_0, e.f := e'; s_2, s_3, x) \;=\; \mathcal{T}_1(s_0; e.f := e', s_2, s_3, x) \qquad \text{where } e \text{ is not a variable}$$

$$\mathcal{T}_1(s_0, x' := e; s_2, s_3, x) \;=\; \mathcal{T}_1(s_0; x' := e, s_2, s_3, x) \qquad \text{where } e \text{ is not } e'.f \tag{2}$$

$$\mathcal{T}_1(s_0, \texttt{return}\ e; s_2, s_3, x) \;=\; \mathcal{T}_1(s_0; \texttt{return}\ e, s_2, s_3, x)$$

$$\mathcal{T}_1(s_0, \texttt{skip}; s_2, s_3, x) \;=\; \mathcal{T}_1(s_0, s_2, s_3, x)$$

$$\mathcal{T}_1(s_0, \texttt{throw}; s_2, s_3, x) \;=\; \mathcal{T}_1(s_0; \texttt{throw}, \texttt{skip}, s_3, x)$$

$$\mathcal{T}_1(s_0, \texttt{try}\ s'_1\ \texttt{catch(Exception}\ x')\ s'_2; s_2, s_3, x) \;=\; \mathcal{T}_1(s_0; \mathcal{T}_1(\texttt{skip}, s'_1, s'_2, x'), s_2, s_3, x)$$

For example, in the case where the first statement in s_2 is $x'.f := e$, the 'dummy' object in its transformation, that is created if variable x' holds a null reference, is used to perform the assignment $x'.f := e$, which is then aborted. Symmetrically we also create a dummy object within a transaction-block which is then aborted directly, so that the assignment can be performed on the initial (non-null) object. Note that the *static* type of an object is used to create a corresponding dummy object. In this way we ensure that fields of dummy objects are the same as the fields of the original one (either directly or via inheritance).

$$\mathcal{T}(\texttt{try}\{x := 1; y.f := v; z.f := v'\}\ \texttt{catch}(\texttt{Exception}\ x')\ \{s_3\}) \xrightarrow{(1)}$$

$$\mathcal{T}_1(\texttt{skip}, x := 1; y.f := v; z.f := v', s_3, x') \xrightarrow{(2)}$$

$$\mathcal{T}_1(x := 1, y.f := v; z.f := v', s_3, x') \xrightarrow{(3)}$$

$$\mathcal{T}_1(x := 1; c_1', z.f := v', s_3, x') \equiv$$

$$\mathcal{T}_1(x := 1; c_1', z.f := v'; \texttt{skip}, s_3, x') \xrightarrow{(2)}$$

$$\mathcal{T}_1(x := 1; c_1'; z.f := v', \texttt{skip}, s_3, x') \xrightarrow{(4)}$$

```
try {
  x := 1;
   if(y == null) {                          }
     beginT; T(s3); commitT; }              }
     beginT;                                }
       beginT; y := new C; commitT;         }
       y.f := v;                            } c1'
     abortT;                                }
   else {                                   }
     beginT;                                }
       beginT; y := new C; abortT;          }
       T(s3);                               }
     abortT;                                }
     beginT; y.f := v; commitT; };          }
  z.f := v'; }
catch(Exception x') { s3 }
```

Figure 8.3: An example program transformation.

In Figure 8.3 an example transformation is presented. It is assumed that x and $z.f$ are expressions with security level Low and that $y.f$ is an expression with security level High. The labels above the arrows refer to the (parts of the) rules in Definitions 22 and 23 that are used for the program transformation.

One can extend the results of the previous section and show that the transformation removes timing leaks from non-interfering programs. The proofs are similar to those of the previous section, but they are more involved, because of the increased complexity of the semantics and definitions of non-interference[73].

8.3.4 Enforcing termination-sensitive non-interference

In order to yield time-sensitive termination-sensitive non-interfering programs, the transformation must take as inputs non-interfering low-recursive programs. The framework discussed in Chapter 7 can be used to prove such properties for sequential Java programs.

8.4 Some observations

This section describes some general observations about our approach, as well as some problems and possible solutions when it is applied in practice. We describe these practical concerns in the context of the (sequential part of the) programming language Java. However we want to stress that the code translation can be applied to any object-oriented language as long as (nested) transaction mechanisms are supported or can be implemented.

8.4.1 Total execution time

The main problem with our transformation is that the total execution time of a program can dramatically increase. Since we always execute both branches of an if-then-else statement the executing time of these increases to the sum of the execution time of both branches. Obviously, for larger programs this is completely impractical. We therefore propose to only use the program transformation in those parts of a program that handle secret information. Examples of these include password checking routines and cryptographic operations.

8.4.2 Time-outs

We do not want to use transactions which can time-out, i.e., which have a maximum bound on the time they can take, then time-out and consequently perform an abort. Either one of two things can happen in such a scenario both of which are undesirable:

1. Information can be leaked.

2. The semantics of our language is no longer preserved.

[73] See Section B.2 in the appendix for a formal definition of time-sensitive non-interference in the current setting.

Consider the translation $\mathcal{T}$ from Definition 19 again. Suppose that each transaction-block has a fixed time-out t_{to}. Then if the statement `c1` hangs and the statement `c2` terminates normally in a time $t_{c2} < t_{to}$ the following situation occurs:

- if `c` evaluates to true both transactions will be aborted, the first (`c2`) explicitly via a call to `abortT` the second (`c1`) implicitly via a time-out (because `c1` does not terminate),
- however if `c` evaluates to false then the first transaction-block will again time-out and thus abort, but the second block will be executed.

So in this scenario information can be leaked (about the value of `c`).

We can prevent this undesirable behavior by putting the complete `if-then -else` in a transaction block, i.e., if `body` is the (translated) if-then-else from Definition 19, we would put this again inside a transaction block, thereby obtaining program fragment `beginT; body; commitT;`. Information is now no longer leaked because if one of the inner transaction blocks time-outs the outer block will obviously also time-out and thus the whole `if-then-else` will no longer be executed. However, in this case the semantics of our language is no longer preserved.

8.4.3 Termination

Our transformation may turn a terminating program into a program that hangs, which is clearly undesirable. Non-termination arises in the transformation when we consider program fragments like `if` e `then` $s1$ `else` $s2$ (a similar argument applies to statements that may throw exceptions). If statement `c1` terminates normally and `c2` hangs then the translated program will always hang. This behavior results from the fact that the termination mode of a statement in a way *overrules* the transaction mechanism. In order to minimize the cases of non-termination and the overhead caused by our transformation, the translation of `if-then-else` blocks with a low conditional is given by the obvious recursive clause. Finer grained approaches that do not introduce non-termination are left for future work.

8.4.4 Timing model

Our operational semantics adopts a very simple model of time. In particular, the execution time of arithmetic expressions is constant and independent of the values assigned to variables. Furthermore, our model of execution time does not reflect the fact that the execution time for a given state may vary depending on the execution history of the program. Thus mechanisms such as caching interfere with our approach and could allow timing leaks.

One drastic solution is to turn off all caching features, such as data-cache, instruction-cache and virtual memory. A better solution would be to impose appropriate interactions between caching and transaction mechanisms. This is left for future work.

8.4.5 Preventing code explosion

Automatic program transformations may lead to a substantial increase in the size of code, or even to code explosion. Our approach is no exception to this, since the size of the transformed code is exponential in the number of nested ifs. In order to avoid code explosion, one can implement the transaction mechanism at the *bytecode* level using subroutines. The basic idea there is that conditional branching statements, which are usually compiled as,

Java bytecode

1	ifeq i
..	s_2
$i-1$	goto j
i	s_1
	..
j	return

are compiled, if they branch over a high expression, into

1	ifeq 9	13	jsr k_2
2	beginT	14	abortT
3	jsr k_1	15	goto 18
4	abortT	16	return
5	beginT	k_1	store x
6	jsr k_2		s'_1
7	commitT		...
8	goto 18		ret x
9	beginT	k_2	store x
10	jsr k_1		s'_2
11	commitT		...
12	beginT		ret x

where the instruction jsr k calls the subroutine starting at k.

Another method that can be used to minimize code explosion is the use of *assertions*. The method is complementary, in that it aims at reducing the branching at instructions that may raise an exception, but in fact do not. Consider the following code fragment:

```
//@ assert o1!=null
    o1.f = c;
```

The assertion states that the variable `o1` is not a null reference. Since this prevents a branching instruction it means in particular that the assignment does not need to be encapsulated inside a `try-catch` block, which simplifies the program transformation considerably. That is, provided the assertion is true, which has to be checked with a separate tool such as ESC/Java2 [CK04]. Note that assertions can also be used to improve the precision of information flow type systems and other approaches for checking non-interference such those discussed in previous chapters.

8.4.6 Optimizing and JIT-compiling

Although we prove that the transformation at the source code level removes all timing leaks we need to be careful when compiling and running the transformed program. In the case of Java we have to ensure that no optimizations are performed when compiling to bytecode. A 'smart' compiler probably removes code fragments like `beginT; ... ; abortT` since semantically these are equivalent to a `skip` statement. In our approach it is crucial that these code-blocks are also present in the compiled bytecode program. Other forms of optimization can have similar undesirable results.

Java bytecode is interpreted by a virtual machine. All modern Java virtual machines use, so called, just-in-time (JIT) compilation of bytecode. This means that code is compiled to native machine language 'on the fly' and that calls to the same method with the same parameters can take different execution times[74] (if the code needs to be compiled on the fly or not). If no timing leaks are allowed then this JIT-compilation has to be disabled and the bytecode simply needs to be interpreted without optimization.

8.4.7 Nested transactions in Java

As far as we know there is no API that implements nested transactions for the Java Standard Edition. Both the Java Enterprise Edition and the Java tailored for smart cards, Java card [Che00], have transaction mechanisms. The former in the form of the Java Transaction API[75] and the latter has transactions as a core language feature. However both transaction mechanisms only allow non-nested transactions.

The one actual implementation of a nested transaction mechanism (for a language of the Java family) we are aware of is the implementation by Lecomte, Grimaud and Donsez [LGD99] for Java card. In order to empirically establish how well our proposed approach works we need to implement a nested transaction mechanism [Mos81]. We leave this as future work.

8.5 Related work

Agat [Aga00a, Aga00c] suggests an approach for removing timing leaks based on program transformation. In essence his approach involves dummy assignments and branching instructions which take exactly the same time as normal assignments and branching instructions. By padding a program with these dummy instructions he can prove that the resulting program will always execute in a time which only depends on non-secret variables, thereby removing all timing leaks. An additional type system then enforces that well-typed padded programs are time-sensitive termination-sensitive non-interfering, under the restriction that all guards of while statements are typed minimal. The programming language Agat considers is an imperative language with arrays, but without objects or exceptions. In [Aga00b] Agat also implemented his approach, using (part of) Java bytecode as his programming language.

Recently, Hedin and Sands [HS05] have extended Agat's work towards an object-oriented language. However, for now, exceptions are not supported.

[74] Assuming the method is evaluated in the same state.

[75] http://java.sun.com/products/jta/

Köpf and Mantel [KM04] exposed ideas of how to improve type systems to eliminate timing leaks by incorporating unification. They look at a simple imperative language without objects.

Di Pierro, Hankin and Wiklicky [DPHW05] have proposed a probabilistic padding algorithm for removing timing leaks in the context of probabilistic process algebra.

A completely different approach is to use some kind of probabilistic reasoning applicable to worst execution time [EES+03]. Since these techniques only address the same problem, but work inherently different from our approach, we will not discuss it further here.

8.6 Conclusions

We show that, under certain assumptions, it is possible to transform termination-sensitive non-interfering object-oriented programs (which can throw exceptions) into object-oriented programs that maintain the stronger form of time-sensitive non-interference.

Chapter 9

Conclusions

> *"It's important perhaps to point out here that secure programs, reliable programs and correct programs are all different things. Knowing how to write provably secure programs is very different from saying we know how to write reliable or correct programs."*
>
> – Alan Cox [Dum05]

In this final chapter we present the conclusions from the previous chapters and give some general comments on the research preformed in the thesis as a whole. The work presented in this thesis can broadly be divided into two parts both of which study security properties for programs written in Java:

> Chapter 3, 4 & 5 explore what can be done regarding security properties using state-of-the-art program verification of traditional functional specifications, consisting of preconditions, postconditions and invariants.

> Chapter 6, 7 & 8 focus on dedicated methods for proving non-interference properties, ranging from automatic checks for a weak form of non-interference to interactive theorem proving and program transformations for verification of stronger forms of non-interference.

As a whole the thesis shows what can be done using the latest specification and verification techniques. At least for small programs –small in lines of code, *not* in terms of complexity– the results are encouraging. We show that it is possible to specify and verify complex behavioral specifications and security properties, specifically confidentiality as non-interference, for Java programs, including programs that include complicated language features (such as exceptions).

On a by chapter basis we conclude the following:

Chapter 3: Specification and verification of Java programs

This chapter shows some of the semantical problems associated with (Java) program verification. A number of canonical examples illustrates the semantical challenges involved and the LOOP verification framework and ESC/Java2 were used to verify these examples. The following are conclusions of the chapter:

- It is possible and feasible to verify the correctness of JML specifications of small, but semantically interesting, programs in a real language like Java.
- ESC/Java2 can prove the correctness of fewer programs than the LOOP tool.
- For most users the assurances a tool like ESC/Java2 (currently) can give are high enough. Its automation and scalability will be far more important features than the occasional soundness bug (which become rarer and rarer). Only for small core parts of programs where the highest assurance levels are required the LOOP tool would still be better suited.
- On a more general note, part of the work is switching to the appropriate 'verification mindset', which requires an analytical view on ones own code. This is already necessary when using JML as a specification language.

We especially want to remark that ESC/Java2 has made a lot of progress compared to its previous incarnation. Semi-automatic tools that do not have the overhead of a general theorem prover such as ESC/Java2 (or tools such as Key [ABB+04], or Jive [MMPH00]) seem to be the future of program verification.

This does not mean that we think that approaches which have formalized Java's semantics entirely inside a theorem prover are not useful. On the contrary, since we know that all rules are provably sound with respect to the Java semantics we have a higher level of assurance that the correctness proofs of (JML) specifications are also correct. As an aside we want to remark that it is possible to prove the correctness of the logic separately from the tool, which has been done for part of the dynamic logic used in the KeY project [Tre05]. In certain small *niches* the added control and flexibility provided by a tool like LOOP are a tremendous advantage. But it seems clear that for large(r) scale verification projects an interactive theorem proving approach is simply too labor intensive. Of course, in such large scale verification projects one might want to use both ESC/Java2 *and* the LOOP tool. The former for the bulk of the work and the initial specification and verification and the latter for a more in-depth look at certain key areas.

Chapter 4: Specification and verification of control flow properties

The chapter discusses a case study in design, development and formal verification of secure smart card applications. JML is used to specify some crucial security properties –especially related to control flow properties– and the LOOP verification framework is used to prove the correctness of these specifications. The following conclusions stem from this chapter:

- It is possible –at least for small JAVA CARD applets– to analyze control flow *directly* at the level of source code. This in contrast to approaches that use some kind of data abstraction [GD00, HGSC04] –typically involving the construction of a call graph– and which thus loose details about the actual implementation.

– This also ensures that it is possible to find actual implementation errors which can prove to be security critical. Low level implementation issues, such as overflow of data types and unwanted exceptions can be detected simultaneously with control flow properties.

There are several aspects left for future research –such as transactions, authentication, links with abstract protocol descriptions, verification with a more complete API specification, more complex JAVA CARD applications– which provide ample material for future work.

Finally, the research in this chapter was inspired by the question if it is possible to bridge the gap between security analysis at the abstract level of e.g., security protocols and the low level of actual source code. This research attempted to close this gap from below –and has some partial success on this topic. Another question for future work is if it is possible to bridge this gap from above e.g., by code-generation techniques.

Chapter 5: Specification and verification of non-interference in JML

In this chapter the notion of a specification pattern for JML is introduced and it is shown how such patterns can be used to specify non-interference properties such as confidentiality and integrity in JML. Any of the JML tools can then be used to verify if the Java program is indeed non-interfering. The main conclusions of this chapter are:

– It is possible to use JML for specifying non-interference properties like confidentiality and integrity using specification patterns for JML.

– With some extra effort it is even possible to specify termination-sensitive non-interference in JML.

– The main disadvantages are that this approach is not complete and that it does not scale. Hence the work in the next chapters that use dedicated methods for proving non-interference properties for Java programs.

There is plenty room for future work. One of the questions that remains is whether it is possible to generate the specification patterns for confidentiality automatically from a secure information flow policy.

Chapter 6: Statically checking termination-insensitive non-interference

The chapter introduces another verification framework for checking termination-insensitive non-interference –the most common notion of confidentiality found in the literature. The framework is formally developed in the theorem prover PVS. It uses dynamic labeling functions that abstractly interpret a simple programming language via modification of security levels of variables. The conclusions for this chapter are:

– The new approach for proving termination-sensitive non-interference is sound and does not require any user interaction.

– Our method generates fewer false positives than classical type-checking based approaches.

– The approach can easily be integrated in a dedicated tool for checking non-interference.

Obviously, there is again room for improvement. An extension to full sequential Java seems an obvious next step, as is an actual implementation of this work.

Chapter 7: Interactively proving termination-sensitive non-interference

This chapter discusses another new approach for checking confidentiality. The notion of confidentiality discussed here is termination *sensitive* non-interference. Bisimulations for Java classes are used to formally define non-interference. Confidentiality can then be proved using a relational Hoare logic. This relational Hoare logic has been specified on top of the LOOP semantics. The rules of the logic are realized as (provable) lemmas in LOOP's Java semantics in PVS. Conclusions of this chapter are:

– A Hoare logic on relations, as introduced in this chapter, can be used for (interactively) proving confidentiality properties in the form of termination-sensitive non-interference.

– The Hoare logic on relations can be used for all sequential Java programs (with the exception of programs that contain inner-classes and floating point data types).

– The logic is sound with respect to LOOP's Java semantics.

– If a program is termination-sensitive non-interfering, then –in principle– we can always prove this. Either directly at the level of the relational Hoare logic or by mixing the reasoning on the relational Hoare logic level with reasoning on a semantic level.

A number of challenges remain for future work. A start has been made on automating the interactive proofs by developing powerful proof strategies (tactics) in PVS. This preliminary work can be extended further.

Another question is whether a relational weakest precondition calculus can be constructed (by lifting the 'ordinary' WP-calculus in the LOOP-framework [Jac04b]). It is not clear if this is desirable since mixing semantic and syntactic reasoning is less convenient in WP-calculi. Formulating rules that are applicable in all situations seems to lead to rules that become too big and are thus no longer of any actual use.

In previous work on the LOOP verification framework case studies have been very helpful in 'fine-tuning' the verification framework [BCHJ05]. We expect that this is also the case for the approach discussed in this chapter.

Chapter 8: Enforcing time-sensitive non-interference

This chapter discusses how timing-leaks can be removed for programs written in an object-oriented language. A program transformation technique is introduced that uses a transaction mechanism to prevent timing leaks in sequential object-oriented programs. Under some strong assumptions, the transformation preserves the semantics of programs and yields for every termination-sensitive non-interfering program a time-sensitive termination-sensitive non-interfering program. For this chapter we concluded the following:

- Under certain assumptions, using nested transactions to remove timing leaks from sequential object-oriented programs with exceptions is possible.
- Such timing leaks can be removed automatically by using a semantics preserving program transformation, as introduced in this chapter.

A possible direction of future research is to see if we can use transaction mechanisms to enforce (standard non-time-sensitive) non-interference in a multi-threaded environment.

General conclusions

We have shown that both behavioral specifications and several non-interference properties can be verified for Java-like programs of relatively small size. Recall that, even in its weakest form, non-interference is a very strong property that in practice is only useful in conjunction with a reasonable downgrading policy. The questions of how the proposed methods scale has not been addressed, but dedicated automatic approaches, such as proposed in Chapter 6 will obviously scale better than more general or interactive approaches.

When to use which approach depends on the context. In case one has a project that uses JML as its main specification language and where confidentiality is just one of the many security properties that have to be checked then the approach discussed in Chapter 5 seems to be the most attractive.

However, in cases where confidentiality is the sole security concern an approach like the one introduced in Chapter 6 is clearly more useful. And some (small but) highly critical modules can then be verified using the interactive technique described in Chapter 7.

The program transformation from Chapter 8 is probably only useful in very specific niches –in particular in (parts of) smart cards applets. Time-sensitive termination-sensitive non-interference is a very strong notion indeed and it is not obvious if any meaningful downgrading policies exist in this context.

Regarding the LOOP tool, we suspect that this thesis is the last in a series, started by Marieke Huisman [Hui01] and followed by Joachim van den Berg and Cees-Bart Breunesse [Bre06], that has the LOOP verification framework as its core subject.

As a final afterthought we wish to emphasize that this thesis is in the first place a *theoretical* study of how to enforce specific security properties –in particular confidentiality as non-interference. In real world systems one does not have the luxury of just focusing on one aspect of a system.

Bibliography

[AB96] R. Anderson & S.J. Bezuidenhoudt. *On the reliability of electronic payment systems.* IEEE Transactions on Software Engineering, 22(5):294–301, 1996.

[AB05] T. Amtoft & A. Banerjee. *A Logic for Information Flow Analysis with and Application to Forward Slicing of Simple Imperative Programs.* Science of Computer Programming, 2005. Available at: http://www.cis.ksu.edu/~ab/Publications/lifafs.pdf.

[ABB+04] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager & P.H. Schmitt. *The KeY tool.* Software and System Modeling, 2004. Online First issue, to appear in print.

[ABB06] T. Amtoft, S. Banhakavi & A. Banerjee. *A Logic for Information Flow in Object-Oriented Programs.* In *Proceedings of the Thirty-third Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2006.

[ABDF03] M. Avvenuti, C. Bernardeschi & N. De Francesco. *Java bytecode verification for secure information flow.* ACM SIGPLAN Notices, 38(12):20–27, 2003.

[ABLP93] M. Abadi, M. Burrows, B. Lampson & G. Plotkin. *A Calculus for Access Control in Distributed Systems.* ACM Transactions on Programming Languages and Systems, 15(4):706–734, 1993.

[Abr96] J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AF99] J. Alves-Foss, ed. *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1999.

[Aga00a] J. Agat. *Transforming out Timing Leaks.* In *27th ACM Symposium on Principles of Programming Languages*, pp. 40–53. ACM Press, 2000.

[Aga00b] J. Agat. *Transforming out Timing Leaks in Practice*, chapter II from [Aga00c]. Department of Computing Science Chalmers University of Technology and Göteborg University, 2000.

[Aga00c] J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation.* Ph.D. thesis, Department of Computing Science Chalmers University of Technology and Göteborg University, SE-412 96 Göteborg, Sweden, 2000.

[And01] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley and sons, Inc., 2001.

[AR80] G.R. Andrews & R.P. Reitman. *An Axiomatic Approach to Information Flow in Programs*. ACM Transactions on Programming Languages and Systems, 2(1):56–76, 1980.

[BAN89] M. Burrows, M. Abadi & R. Needham. *A logic of authentication*. Proc. Royal Soc., Series A, Volume 426:233–271, 1989.

[Bar03] J. Barnes. *High Integrity Sofware–The Spark Approach to Safety and Security*. Addison-Wesley, 2003.

[BCC+05] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino & E. Poll. *An overview of JML tools and applications*. International Journal on Software Tools for Technology Transfer (STTT), 7(3):212–232, 2005. Special section on formal methods for industrial critical systems.

[BCHJ05] C.-B. Breunesse, N. Cataño, M. Huisman & B. Jacobs. *Formal methods for smart cards: an experience report*. Science of Computer Programming, 55:53–80, 2005.

[BDR04] G. Barthe, P. D'Argenio & T. Rezk. *Secure Information Flow by Self-Composition*. In R. Foccardi, ed., *Proceedings of CSFW'04*, pp. 100–114. IEEE Press, 2004.

[Bec01] B. Beckert. *A dynamic logic for the formal verification of Java Card programs*. In I. Attali & T. Jensen, eds., *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pp. 6–24. Springer-Verlag, 2001.

[Ben04] A. Benton. *Simple Relational Correctness Proofs for Static Analyses and Program Transformations*. In *Proceedings of the 31th Symposium on Principals of Programming (POPL)*. ACM, 2004.

[Ber04] D. Bernstein. *Cache-timing attacks on AES*, 2004. Available at http://cr.yp.to/papers.html#cachetiming.

[BFMW01] D. Bartetzko, C. Fischer, M. Müller & H. Wehrheim. *Jass - Java with Assertions*. In K. Havelund & G. Rosu, eds., *Proceedings of the First Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier Science, 2001.

[BH02] R. Brinkman & J.H. Hoepman. *Secure method invocation in Jason*. In *Proceedings of Cardis 2002*, pp. 29–40. USENIX Assoc., 2002.

[BHJP00] J. van den Berg, M. Huisman, B. Jacobs & E. Poll. *A Type-Theoretic Memory Model for Verification of Sequential Java Programs*. In D. Bert, C. Choppy & P. Mosses, eds., *Recent Trends in Algebraic Development Techniques*, volume 1827 of *LNCS*, pp. 1–21. Springer-Verlag, 2000.

[Bib77] K.J. Biba. *Integrity considerations for secure computer systems. Technical Report MTR-3153*, MITRE Corp., 1977.

[BJ01] J. van den Berg & B. Jacobs. *The LOOP compiler for Java and JML*. In T. Margaria & W. Yi, eds., *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *LNCS*, pp. 299–312. Springer-Verlag, 2001.

[BJP01] J. van den Berg, B. Jacobs & E. Poll. *Formal specification and verification of Java Card's application identifier class*. In I. Attali & T. Jensen, eds., *Java on Smart Cards: Programming and Security (JavaCard Workshop 2000)*, volume 2041 of *LNCS*, pp. 137–150. Springer-Verlag, 2001.

[BL99] J. Bergstra & M. Loots. *Empirical semantics for object-oriented programs*, 1999. Artificial Intelligence Preprint Series nr. 007, Dep. Philosophy, Utrecht Univ. http://preprints.phil.uu.nl/aips/.

[BLR02] L. Burdy, J.-L. Lanet & A. Requet. *JACK (Java Applet Correctness Kit)*, 2002. http://www.gemplus.com/smart/r_d/trends/jack.html.

[BM03] B. Beckert & W. Mostowski. *A program logic for handling Java Card's transaction mechanism*. In M. Pezzè, ed., *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2003, Warsaw, Poland*, volume 2621 of *LNCS*, pp. 246–260. Springer-Verlag, 2003.

[BN02] A. Banerjee & D. Naumann. *Secure Information Flow and Pointer Confinement in a Java-like Language*. In *Proc. of the Fifteenth IEEE Computer Security Foundations Workshop (CSFW)*, pp. 253–267. IEEE Computer Society Press, 2002.

[BN05] A. Banerjee & D. Naumann. *Stack-Based Access Control for Secure Information Flow*. Journal of Functional Programming, 15(2):131–177, 2005. Special Issue on Language-Based Security.

[BNSS05] M. Barnett, D. Naumann, W. Schulte & Q. Sun. *99.44% Pure: Functional Abstractions in Specifications*, 2005. Submitted to the Journal of Object Technology.

[Boy03] R. Boyer. *Proving Theorems about Java and the JVM with ACL2*. In M. Broy & M. Pizka, eds., *Models, Algebras and Logic of Engineering Software*, volume 191 of *NATO Science Series: Computer & Systems Sciences*, pp. 227–290. IOS Press, Amsterdam, 2003.

[BP03] C.-B. Breunesse & E. Poll. *Verifying JML specifications with model fields*. In *Formal Techniques for Java-like Programs. Proceedings of the ECOOP'2003 Workshop*, 2003.

[BRB05] G. Barthe, T. Rezk & A. Basu. *Security Types Preserving Compilation*. the International Journal of Computer Languages, Systems and Structures, to appear, 2005.

[Bre06] C.-B. Breunesse. *On JML: topics in tool-assisted verification of Java programs*. Ph.D. thesis, Radboud University, Nijmegen, The Netherlands, 2006.

[BRLS05] M. Barnett, K. Rustan, M. Leino & W. Schulte. *The Spec# Programming System: An Overview*. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet & T. Muntean, eds., *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. International Workshop, CASSIS 2004. Revised Selected Papers*, volume 3362 of *LNCS*, pp. 49–70. Springer-Verlag, 2005.

[BRW06] G. Barthe, T. Rezk & M. Warnier. *Preventing Timing Leaks Through Transactional Branching Instructions*. In *Proceedings of the Third Workshop on Quantitive Aspects of Programming Languages (QAPL 2005)*, volume 153 of *ENTCS*, pp. 33–55. Elsevier Science, 2006.

[BS99] E. Börger & W. Schulte. *Initialization problems in Java*. Software—Concepts and Tools, 20(4), 1999.

[CC77] P. Cousot & R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 238–252. ACM Press, New York, NY, Los Angeles, California, 1977.

[CC04] H. Chen & S. Chong. *Owned Policies for Information Security*. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW)*, 2004.

[CDF+04] E. Contejean, J. Duprat, J.-C. Filliâtre, C. Marché, C. Paulin-Mohring & X. Urbain. *The Krakatoa tool for certification of Java/Java Card programs annotated in JML*. Journal of Logic and Algebraic Programming, 58(1-2):89–106, 2004. Available via the Krakatoa home page at http://www.lri.fr/~marche/krakatoa/.

[CDV96] J. Crow & B.L. Di Vito. *Formalizing Space Shuttle software requirements*. In *First Workshop on Formal Methods in Software Practices (FMSP'96)*, pp. 40—48. ACM, 1996.

[Cha02] P. Chalin. *Back to basics: Language support and semantics of basic infinite integer types in JML and Larch. Technical Report 2002.003.1*, Computer Science Department, Concordia University, 2002. http://www.cs.concordia.ca/~faculty/chalin/.

[Cha03] P. Chalin. *Improving JML: For a Safer and More Effective Language*. In *Formal Methods Europe 2003*, 2003.

[Cha04] P. Chalin. *JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics*. Journal of Object Technology, 3(6):57–79, 2004.

[Che00] Z. Chen. *Java Card technology for smart cards: architecture and programmer's guide*. Addison-Wesley, 2000.

[CHH02] D. Clark, C. Hankin & S. Hunt. *Information Flow for ALGOL-like Languages*. Computer Languages, 28(1):3–28, 2002. Special Issue on Computer Languages and Security.

[Chi] *Chipknip, the Dutch electronic wallet smart card*. http://www.chipknip.nl.

[CK04] D.R. Cok & J.R. Kiniry. *ESC/Java2: Uniting ESC/Java and JML*. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet & T. Muntean, eds., *Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pp. 108–128. Springer-Verlag, 2004.

[CKRW99] P. Cenciarelli, A. Knapp, B. Reus & M. Wirsing. *An Event-Based Structural Operational Semantics of Multi-Threaded Java*. In Alves-Foss [AF99], pp. 157–200.

[CL02a] Y. Cheon & G. Leavens. *A runtime assertion checker for the Java Modeling Language (JML)*. In H. Arabnia & Y. Mun, eds., *International Conference on Software Engineering Research and Practice (SERP '02)*, pp. 322–328. CSREA Press, Las Vegas, 2002.

[CL02b] Y. Cheon & G.T. Leavens. *A Simple and Practical Approach to Unit Testing: The JML and JUnit Way*. In B. Magnusson, ed., *ECOOP 2002 – Object-Oriented Programming, 16th European Conference*, volume 2374 of *LNCS*, pp. 231–255. Springer-Verlag, 2002.

[CLSE05] Y. Cheon, G. Leavens, M. Sitaraman & S. Edwards. *Model variables: cleanly supporting abstraction in design by contract*. spe, 35(6):583–599, 2005.

[CM04] S. Chong & A.C. Myers. *Security Policies for Downgrading*. In *11th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, 20004.

[CM04] C.J.F. Cremers & S. Mauw. *Operational semantics of security protocols*. In S. Leue & T. Syst, eds., *Scenarios: models, transformations and tools, international workshop, dagstuhl castle, germany, september 7-12, 2003, revised selected papers*, volume 3466 of *LNCS*. Springer-Verlag, 2004.

[Coq] The Coq proof assistant, http://coq.inria.fr/.

[Cor06] R. Corin. *Analysis Models for Security Protocols*. Ph.D. thesis, University of Twente, Enschede, The Netherlands, 2006.

[Cou96] P. Cousot. *Abstract interpretation*. Symposium on Models of Programming Languages and Computation, ACM Computing Surveys, 28(2):324–328, 1996.

[CR05] P. Chalin & F. Rioux. *Non-null References by Default in the Java Modeling Language*. In *Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS'05)*. Lisbon, Portugal, 2005.

[DAC99] M.B. Dwyer, G.S. Avrunin & J.-C. Corbett. *Patterns in Property Specification for Finite-State Verification*. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, 1999.

[DD77] D.E. Denning & P.J. Denning. *Certification of programs for secure information flow*. Communications of the ACM, 20(7):504–513, 1977.

[Den76] D.E. Denning. *A Lattice Model of Secure Information Flow*. Communications of the ACM, 19(5), 1976.

[Den82] D.E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[DFM05] G. Dufay, A. Felty & S. Matwin. *Privacy-Sensitive Information Flow with JML*. In *Conference on Automated Deduction (CADE) proceeedings*, volume 3632 of *LNCS*. Springer-Verlag, 2005.

[DHS05] Á. Darvas, R. Hähnle & D. Sands. *A Theorem Proving Approach to Analysis of Secure Information Flow*. In *Proc. 2nd International Conference on Security in Pervasive Computing*, LNCS. Springer-Verlag, to appear, 2005.

[Dij76] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DM05] Á. Darvas & P. Müller. *Reasoning About Method Calls in JML Specifications*. In *Workshop on Formal Techniques in Java-like Programs (FTfJP) prooceedings*, 2005.

[DNS05] D. Detlefs, G. Nelson & J.B. Saxe. *Simplify: a theorem prover for program checking*. Journal of the ACM, 52(3):365–473, 2005.

[DPHW05] A. Di Pierro, C. Hankin & H. Wiklicky. *Time-based Interference and Probabilistic Padding*. In *Informal proceedings of the 2nd International Workshop on Programming Language Interference and Dependence (PLID'05)*, 2005.

[Dum05] E. Dumbill. *The Next 50 Years of Computer Security: An Interview with Alan Cox*. O'reilly network, http://www.oreillynet.com/pub/a/network/2005/09/12/alan-cox.html, 2005.

[DY83] D. Dolev & A. Yao. *On the security of public key protocols*. IEEE Transactions on Information Theory, 29(6), 1983.

[EES+03] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson & H. Hansson. *Worst-case execution-time analysis for embedded real-time systems*. International Journal on Software Tools for Technology Transfer (STTT), 4(4):437–455, 2003.

[ESC] ESC/JAVA2 project. Available at: http://secure.ucd.ie/products/opensource/ESCJava2/.

[Ese01] O. Eseling. *iContract: Design by contract in Java*. available at: http://www.javaworld.com/archives/index-jw-02-2001.html, 2001.

[Fil] J.-C. Filliâtre. *Why: A multi-language multi-prover verification condition generator*.

[Fin] *FindBugs - Find Bugs in Java Programs*. Available at: http://findbugs.sourceforge.net/.

[FLL+02] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe & R. Stata. *Extended static checking for Java*. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37(5) of *SIGPLAN Notices*, pp. 234–245. ACM, 2002.

[Flo] Flow Caml website. http://cristal.inria.fr/~simonet/soft/flowcaml/.

[Flo67] R.W. Floyd. *Assigning meaning to programs*. In J.T. Schwartz, ed., *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia*, volume 19, pp. 19–31. American Mathematical Society, Providence RI, 1967.

[For00] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement, FDR2 user manual*, 2000.

[Fow00] M. Fowler. *UML Distilled*. Addison-Wesley, 2000.

[GD00] P. Giambiagi & M. Dam. *Confidentiality for mobile code: The case of a simple payment protocol*. *In* Proc. 13th IEEE Computer Security Foundations Workshop, 2000.

[Geh83] N. Gehani. *Ada: An Advanced Introduction*. Prentice-Hall, 1983.

[GH98] W. Griffoen & M. Huisman. *A Comparison of PVS and Isabelle/HOL*. In *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs '98)*, volume 1479 of *LNCS*, pp. 123–142. Springer-Verlag, 1998.

[GHJ05] A.D. Gordon, C. Haack & A. Jeffrey. *CRYPTYC: Cryptographic Protocol Type Checker*, 2005. http://www.cryptyc.org/.

[GJSB00] J. Gosling, B. Joy, G. Steele & G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000. http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

[Glo01] Global Platform. *Open platform card specification version 2.1*, 2001. Available at http://www.globalplatform.org/.

[GM82] J. Goguen & J. Meseguer. *Security policies and security models*. In *IEEE Symp. on Security and Privacy*, pp. 11–20. IEEE Comp. Soc. Press, 1982.

[GM04] R. Giacobazzi & I. Mastroeni. *Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation*. In *POPL04 proceedings*, 2004.

[GMP93] D. Guaspari, C. Marceau & W. Polak. *Formal verification of ada programs*. In U. Martin & J.M. Wing, eds., *First International Workshop on Larch*, pp. 104–141. Springer, 1993.

[Gra95] P. Graham. *ANSI Common Lisp*. Prentice-Hall, 1995.

[Gri81] D. Gries. *The Science of Programming*. Springer, 1981.

[HGSC04] M. Huisman, D. Gurov, C. Sprenger & G. Chugunov. *Checking Absence of Illicit Applet Interactions: A Case Study*. In *Fundamental Approaches to Software Engineering (FASE'04)*, volume 2984 of *LNCS*, pp. 84–98. Springer-Verlag, 2004.

[HJ00a] M. Huisman & B. Jacobs. *Inheritance in higher order logic: Modeling and reasoning*. In M. Aagaard & J. Harrison, eds., *Theorem Proving in Higher Order Logics*, volume 1869 of *LNCS*, pp. 301–319. Springer, Berlin, 2000.

[HJ00b] M. Huisman & B. Jacobs. *Java program verification via a Hoare logic with abrupt termination*. In T. Maibaum, ed., *Fundamental Approaches to Software Engineering (FASE'00)*, volume 1783 of *LNCS*, pp. 284–303. Springer-Verlag, 2000.

[HJKO04] E. Hubbers, B. Jacobs, J. Kiniry & M. Oostdijk. *Counting votes with formal methods*. In C. Rattray, S. Maharaj & C. Shankland, eds., *Algebraic Methodology and Software Technology (AMAST'04)*, volume 3116 of *LNCS*, pp. 241–257. Springer-Verlag, 2004.

[HM01] P.H. Hartel & L. Moreau. *Formalizing the safety of Java the Java virtual machine, and Java Card*. ACM Computing Surveys, 33(4):517–558, 2001.

[HM05] R. Hähnle & W. Mostowski. *Verification of Safety Properties in the Presence of Transactions*. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet & T. Muntean, eds., *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pp. 151–171. Springer-Verlag, 2005.

[HO03] E. Hubbers & M. Oostdijk. *Generating JML specifications from UML state diagrams*. In *Proceedings of the Forum on specification & Design Languages (FDL 2003)*, pp. 263–273. University of Frankfurt, 2003. Proceedings appeared as CD-Rom with ISSN 1636-9874.

[Hoa69] C. Hoare. *An axiomatic basis for computer programming*. Communications of the ACM, 12:576–580, 583, 1969.

[HOP03] E. Hubbers, M. Oostdijk & E. Poll. *From Finite State Machines To Provably Correct Java Card Applets*. In *Proceedings of the 18th IFIP Information Security Conference*. Kluwer Academic Publishers, 2003.

[HOP04] E. Hubbers, M. Oostdijk & E. Poll. *Implementing a formally verifiable security protocol in Java Card*. In D. Hutter, G. Müller, W. Stephan & M. Ullmann, eds., *Proceedings of the 1st International Conference on Security in Pervasive Computing*, volume 2802 of *LNCS*, pp. 213–226. Springer-Verlag, 2004.

[Hor05] C. Horstmann. *Java concepts*. Wiley, 4th edition, 2005.

[HP04] E. Hubbers & E. Poll. *Reasoning about card tears and transactions in Java Card*. In *Fundamental Approaches to Software Engineering (FASE'2004), Barcelona, Spain*, volume 2984 of *LNCS*, pp. 114–128. Springer, 2004.

[HR98] N. Heinze & J. Riecke. *The SLam calculus:programming with secrecy and integrity*. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pp. 365–377, 1998.

[HRS02] D. Haneberg, W. Reif & K. Stenzel. *A method for secure smartcard applications*. In *Proceedings of AMAST 2002*, volume 2422 of *LNCS*, pp. 319–333. Springer-Verlag, 2002.

[HS05] D. Hedin & D. Sands. *Timing Aware Information Flow Security for a JavaCard-like Bytecode*. In F. Spoto, ed., *First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE) proceedings*, ENTCS, pp. 149–166. Elsevier, 2005.

[Hui01] M. Huisman. *Reasoning about Java programs in higher order logic with PVS and Isabelle*. Ph.D. thesis, University of Nijmegen, Nijmegen, The Netherlands, 2001.

[ISO] *International Standard ISO 7816*. Available at: http://www.iso.org/iso/en/ISOOnline.frontpage.

[Jac01] B. Jacobs. *A formalisation of Java's exception mechanism.* In D. Sands, ed., *Programming Languages and Systems (ESOP)*, volume 2028 of *LNCS*, pp. 284–301. Springer, Berlin, 2001.

[Jac02] B. Jacobs. *Exercises in coalgebraic specification.* In R.C. R. Backhouse & J. Gibbons, eds., *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, pp. 237–280. Springer, Berlin, 2002.

[Jac03] B. Jacobs. *Java's integral types in PVS.* In E. Najim, U. Nestmann & P. Stevens, eds., *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *LNCS*, pp. 1–15. Springer, Berlin, 2003.

[Jac04a] B. Jacobs. *Semantics and Logic for Security Protocols*, 2004.
available at:
http://www.cs.ru.nl/B.Jacobs/PAPERS/protsemlog.pdf.

[Jac04b] B. Jacobs. *Weakest precondition reasoning for Java programs with JML annotations.* Journal of Logic and Algebraic Programming, 58(1-2):61–88, 2004.

[Jas] *Jass: Java with assertions.* http://csd.informatik.uni-oldenburg.de/~jass/index.html.

[Jif] *Jif:Java + information flow.* http://www.cs.cornell.edu/jif/.

[JKW03] B. Jacobs, J. Kiniry & M. Warnier. *Java Program Verification Challenges.* In F.S. de Boer, M.M. Bonsangue, S. Graf & W.P. de Roever, eds., *Formal Methods for Components and Objects*, volume 2852 of *LNCS*, pp. 202–219. Springer, Berlin, 2003.

[JL00] R. Joshi & K. Leino. *A semantic approach to secure information flow.* Science of Comput. Progr., 37(1-3):113–138, 2000.

[JML] *JML web site.* http://www.jmlspecs.org.

[JMR04] B. Jacobs, C. Marché & N. Rauch. *Formal verification of a commercial smart card applet with multiple tools.* In C. Rattray, S. Maharaj & C. Shankland, eds., *Algebraic Methodology and Software Technology (AMAST'04)*, volume 3116 of *LNCS*, pp. 21–22. Springer, Berlin, 2004.

[JOW04] B. Jacobs, M. Oostdijk & M. Warnier. *Source Code Verification of a Secure Payment Applet.* Journ. of Logic and Algebraic Programming, 58(1-2):107–120, 2004.

[JP03] B. Jacobs & E. Poll. *Coalgebras and monads in the semantics of Java.* Theoretical Computer Science, 291(3):329–349, 2003.

[JP04] B. Jacobs & E. Poll. *Java Program Verification at Nijmegen: Developments and Perspective.* In *Software Security - Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003*, volume 3233 of *LNCS*, pp. 134 – 153. Springer, 2004.

[JPW05] B. Jacobs, W. Pieters & M. Warnier. *Statically checking confidentiality via dynamic labels.* In *WITS '05: Proceedings of the 2005 workshop on Issues in the theory of security*, pp. 50–56. ACM Press, New York, NY, USA, 2005.

[Kin06] J. Kiniry. *Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application.* In C. Doney et al, ed., *Exception Handling*, volume 4119 of *LNCS*, pp. 288–300. Springer-Verlag, 2006.

[KM04] B. Köpf & H. Mantel. *Eliminating timing leaks by unification (extended abstract)*, 2004.

[Koc96] P. Kocher. *Timing attacks on implementations of diffie-helman, rsa, dss, and other systems.* In N. Koblitz, ed., *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pp. 104–113. Springer-Verlag, 1996.

[Kra98] R. Kramer. *iContract - The Java(tm) Design by Contract(tm) Tool.* In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, p. 295. IEEE Computer Society, 1998.

[KS02] D. Kozen & M. Stillerman. *Eager Class Initialization in Java.* In W. Damm & E. Olderog, eds., *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02)*, volume 2469 of *LNCS*, pp. 71–80. Springer-Verlag, 2002.

[Lam73] B.W. Lampson. *A note on the Confinement Problem.* Communications of the ACM, 16(10):613–615, 1973.

[LB73] L.J. LaPadula & D.E. Bell. *Secure Computer systems: A mathematical model. Technical Report MTR-2547, Vol 2*, MITRE Corp., 1973. Reprinted in *J. of Computer Security*, vol 4, no 2–3, pp. 239–263, 1996.

[LBR99a] G. Leavens, A. Baker & C. Ruby. *Preliminary design of JML: A behavioral interface specification language for Java*, 1999. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. http://www.cs.iastate.edu/~leavens/JML.html.

[LBR99b] G.T. Leavens, A.L. Baker & C. Ruby. *JML: A Notation for Detailed Design.* In H. Kilov, B. Rumpe & W. Harvey, eds., *Behavioral Specification for Businesses and Systems*, chapter 12, pp. 175–188. Kluwer Academic Publishers, 1999.

[Lev01] S. Levy. *Crypto.* Penguin Books, 2001.

[LGD99] S. Lecomte, G. Grimaud & D. Donsez. *Implementation of Transactional Mechanisms for Open SmartCard.* In *GEMPLUS Developer Conference*, 1999.

[LMWF94] N. Lynch, M. Merritt, W. Weihl & A. Fekete. *Atomic Transactions.* Morgan Kaufmann Publishers, 1994.

[Low95] G. Lowe. *An attack on the Needham-Schroeder public-key authentication protocol.* Information Processing Letters, 56:131–133, 1995.

[Low98] G. Lowe. *Casper: A compiler for the analysis of security protocols*. Journal of Computer Security, 6:53–84, 1998.

[LPC+05] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok & J. Kiniry. *JML reference Manual (draft)*, 2005. Available at: http://www.jmlspecs.org/jmlrefman/jmlrefman_toc.html.

[LW94] B. Liskov & J. Wing. *A behavioral notion of subtyping*. ACM Transactions on Programming Languages and Systems, 7(16(6)):1811–1841, 1994.

[LY99] T. Lindholm & F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[LZ05] P. Li & S. Zdancewic. *Downgrading Policies and Relaxed Noninterference*. In *Proceedings of the 32nd Symposium on Principals of Programming (POPL)*, pp. 158–170. ACM, 2005.

[MBRS04] E. Ábrahám Mumm, F.S. de Boer, W.P. de Roever & M. Steffen. *An Assertion-based Proof System for Multithreaded Java*. Special issue of TCS, 331:251–290, 2004.

[Mea96] C. Meadows. *The NRL Protocol Analyzer: An Overview. Technical report*, Center for High Assurance Computer Systems, Navel Research Laboratory, 1996.

[Mea03] C. Meadows. *Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends*. IEEE Journal on Selected Areas in Communication, 21(1):44–54, 2003.

[Mey92] B. Meyer. *Applying "Design by Contract"*. IEEE Computer, 25(10):40–51, 1992.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[ML97] A.C. Myers & B. Liskov. *A Decentralized Model for Information Flow Control*. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 129–142, 1997.

[MM01] R. Marlet & D.L. Métayer. *Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006*, Trusted Logic, 2001.

[MMPH00] J. Meyer, P. Müller & A. Poetzsch-Heffter. *The* JIVE *system—implementation description*, 2000. Available at http://softech.informatik.uni-kl.de/downloads/publications/jive.pdf.

[Moo99] S. Moore. *Proving Theorems About Java-Like Byte Code*. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, volume 1710 of *LNCS*, pp. 139–162. Springer-Verlag, 1999.

[Mos81] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. Ph.D. thesis, MIT, 1981.

[Mos02] W. Mostowski. *Rigorous development of JavaCard applications.* In T. Clarke, A. Evans & K. Lano, eds., *Proc. Fourth Workshop on Rigorous Object-Oriented Methods, London*, 2002.

[Mos05a] W. Mostowski. *Formal Development of Safe and Secure Java Card Applets.* Ph.D. thesis, Chalmers University of Technology, Department of Computer Science and Engineering, Göteborg, Sweden, 2005.

[Mos05b] W. Mostowski. *Formalisation and verification of Java Card security properties in Dynamic Logic.* In M. Cerioli, ed., *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2005, Edinburgh, Scotland*, volume 3442 of *LNCS*, pp. 357–371. Springer, 2005.

[MP05] J. Manson & W. Pugh. *The Java Memory Model.* In *Proceedings of the 32nd Symposium on Principals of Programming (POPL)*, pp. 378–392. ACM, 2005.

[MPH00] J. Meyer & A. Poetzsch-Heffter. *An architecture for interactive program provers.* In S. Graf & M. Schwartzbach, eds., *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *LNCS*, pp. 63–77. Springer, Berlin, 2000.

[MSZar] A.C. Myers, A. Sabelfeld & S. Zdancewic. *Enforcing Robust Declassification.* Journal of Computer Security, 2006, to appear. Available at: http://www.cs.chalmers.se/~andrei/msz-jcs.pdf.

[Mye99] A.C. Myers. *JFlow: Practical Mostly-Static Information Flow Control.* In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, 1999.

[NL97] G. Necula & P. Lee. *Proof-carrying code.* In *Proceedings of the 24th Symposium on Principals of Programming (POPL).* ACM, 1997.

[NN92] H.R. Nielson & F. Nielson. *Semantics with Applications.* Wiley Professional Computing, 1992.

[ORSH95] S. Owre, J. Rushby, N. Shankar & F. von Henke. *Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS.* IEEE Trans. on Softw. Eng., 21(2):107–125, 1995.

[OSRSC99] S. Owre, N. Shankar, J.M. Rushby & D.W.J. Stringer-Calvert. *PVS System Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, USA, 1999. URL http://pvs.csl.sri.com/, available at http://pvs.csl.sri.com/.

[OW03] M. Oostdijk & M. Warnier. *On the combination of Java Card Remote Method Invocation and JML. Technical Report NIII-R0321*, Nijmegen Institute for Computer and Information Sciences, 2003.

[Par81] D. Park. *Concurrency and Automata on Infinite Sequences.* In P. Deussen, ed., *Proceedings 5th GI Conference on Theoretical Computer Science*, volume 104 of *LNCS*, pp. 15–32. Springer, Berlin, 1981.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS. Springer, Berlin, 1994.

[Pau98] L. Paulson. *The inductive approach to verifying cryptographic protocols*. Journ. of Computer Security, 6:85–128, 1998.

[PBB+04] M. Pavlova, G. Barthe, L. Burdy, M. Huisman & J.-L. Lanet. *Enforcing High Level Security Properties For Applets*. In J.J. Quisquater, P. Paradinas, Y. Deswarte & A. El Kalam, eds., *Proceedings of Smart Card Research and Advanced Applications (CARDIS)*. IFIP, Kluwer Academic Publishers, 2004.

[PBJ00] E. Poll, J. van den Berg & B. Jacobs. *Specification of the Java Card API in JML*. In J. Domingo-Ferrer, D. Chan & A. Watson, eds., *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pp. 135–154. Kluwer, 2000.

[PP03] C.P. Pfleeger & S.L. Pfleeger. *Security in Computing*. Prentice Hall, 3th edition, 2003.

[Pro] *Proton, the Belgian electronic wallet smart card*. http://www.proton.be.

[PS03] F. Pottier & V. Simonet. *Information flow inference for ML*. ACM Transactions on Programming Languages and Systems, 25(1):117–158, 2003.

[PVS] The PVS verification system, http://pvs.csl.sri.com/.

[RE00] W. Rankl & W. Effing. *Smart Card Handbook*. John Wiley & Sons, 2nd edition, 2000.

[RG02] M. Richters & M. Gogolla. *OCL: Syntax, Semantics, and Tools*. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pp. 42–68. Springer-Verlag, London, UK, 2002.

[RM02] A. Roychoudhury & T. Mitra. *Specifying Multithreaded Java Semantics for Program Verification*. In *Proceedings of the International Conference on Software Engineering — ICSE*, 2002.

[RTJ01] J. Rothe, H. Tews & B. Jacobs. *The coalgebraic class specification language CCSL*. Journal of Universal Computer Science, 7(2), 2001.

[Rut00] J. Rutten. *Universal coalgebra: a theory of systems*. Theoretical Computer Science, 249:3–80, 2000.

[SCFY96] R. Sandhu, E. Coyne, H. Feinstein & C. Youman. *Role-Based Access Control Models*. IEEE Computer, 29(2nd), 1996.

[Sch86] D. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers Dubuque, IA, USA, 1986.

[Sch00] B. Schneier. *Secrets & Lies*. Wiley, 2000.

[Sec] *European IST-1999-29075 Project SecSafe*. http://www.doc.ic.ac.uk/~siveroni/secsafe/.

[SM03] A. Sabelfeld & A.C. Myers. *Language-Based Information-Flow Security*. IEEE Journal on selected areas in communications, 21(1), 2003.

[SORSC99] N. Shanker, S. Owre, J. Rushby & D. Stringer-Calvert. *PVS prover guide*, 1999. Version 2.3.

[SS01] A. Sabelfeld & D. Sands. *A Per Model of Secure Information Flow in Sequential Programs*. In S. Swierstra, ed., *Programming Languages and Systems: 8th European Symposium on Programming, ESOP'99*, volume 1576 of *LNCS*, pp. 59–91. Springer-Verlag, 2001.

[Str03] M. Strecker. *Formal analysis of an information flow type system for MicroJava (extended version). Technical report*, Technische Universität München, 2003.

[Szy98] C. Szyperski. *Component Software*. Addison-Wesley, 1998.

[The02] The Krakatoa team. *The Krakatoa proof tool*, 2002. Available from http://www.lri.fr/~marche/krakatoa/.

[Tre05] K. Trentelman. *Proving Correctness of JavaCard DL Taclets using Bali*. In B.K. Aichernig & B. Beckert, eds., *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pp. 160–169. IEEE Computer Society, 2005.

[Ver] *European IST-2000-26328 Project VerifiCard*. http://www.verificard.org.

[VS97a] D. Volpano & G. Smith. *A Type-Based Approach to Program Security*. In *Proc. 7th Int'l Joint Conference on the Theory and Practice of Software Development*, volume 1214 of *LNCS*, pp. 607–621. Springer, 1997.

[VS97b] D. Volpano & G. Smith. *Eliminating Covert Flows with Minimum Typings*. In *Proc. 10th IEEE Computer Security Foundations Workshop*, pp. 156–168, 1997.

[VSI96] D. Volpano, G. Smith & C. Irvine. *A sound type system for secure flow analysis*. Journal of computer security, 4(3):167–187, 1996.

[WO05] M. Warnier & M. Oostdijk. *Non-interference in JML. Technical Report ICIS-R05034*, Nijmegen Institute for Computing and Information Sciences, 2005.

[Yan05] H. Yang. *Relational separation logic*, 2005. Available at http://ropas.snu.ac.kr/~hyang/paper/data.ps, submitted to the Journal on Theoretical Computer Science.

[Zan02] M. Zanotti. *Security Typings by Abstract Interpretation*. In *SAS*, volume 2477 of *LNCS*, pp. 360–375. Springer-Verlag, 2002.

[ZM01] S. Zdancewic & A.C. Myers. *Robust Declassification*. In *Proceedings of 14th IEEE Computer Security Foundations Workshop (CSFW)*, pp. 15–23, 2001.

Appendix A

Java source code listings

A.1 The phone card applet from chapter 4

```
import javacard.framework.*;
import javacard.security.*;

public class PayApplet extends Applet {

    static final byte INS_SETKEY = (byte)0x00;
    static final byte INS_GETVAL = (byte)0x01;
    static final byte INS_DECVAL = (byte)0x02;
    static final byte INS_GETCHAL = (byte)0x03;
    static final byte INS_RESPOND = (byte)0x04;

    static final short DES_KEY_SIZE = 8;
    static final short ID_SIZE = 8;
    static final short CIPHERTEXT_SIZE = 24;
    static final short NONCE_SIZE = 32;
    static final short SHA_SIZE = 20;
    static final short TMP_SIZE = 16;

    static final byte PERS = 0;
    static final byte ISSUED = 1;
    static final byte CHARGING = 2;
    static final byte LOCKED = 3;

    private /*@ spec_public @*/ Cipher cipher;
    private /*@ spec_public @*/ MessageDigest digest;
    private /*@ spec_public @*/ RandomData random;

    private /*@ spec_public @*/ byte state;
    private /*@ spec_public @*/ byte[] key;
    private /*@ spec_public @*/ byte[] id;
    private /*@ spec_public @*/ byte[] tmp;
    private /*@ spec_public @*/ byte[] nonce;
    private /*@ spec_public @*/ byte[] sha_nonce;
    private /*@ spec_public @*/ byte counter;
    private /*@ spec_public @*/ short value;
    private /*@ spec_public @*/ byte[] plaintxt;
    private /*@ spec_public @*/ byte[] ciphertxt;
```

```
    /*@ invariant
      @ (state == PERS || state == ISSUED ||
      @          state == CHARGING || state == LOCKED)
      @   && 0 <= counter && counter <= 5
      @   && (state == LOCKED <==> counter == 5)
      @   && (state == PERS ==> counter == 0)
      @   && 0 <= value && value <= 4096 // = 16 * 256
      @   && key != null && key.length == DES_KEY_SIZE
      @   && id  != null && id.length == ID_SIZE
      @   && tmp != null && tmp.length == TMP_SIZE
      @   && nonce != null && nonce.length == NONCE_SIZE
      @   && sha_nonce != null && sha_nonce.length == SHA_SIZE
      @   && plaintxt != null && plaintxt.length == SHA_SIZE+2
      @   && ciphertxt != null
      @   && ciphertxt.length == CIPHERTEXT_SIZE;
      @*/

    /*@ constraint  // automatically generated
      @    (state==LOCKED ==> \old(state)==ISSUED
      @                        || \old(state)==LOCKED) &&
      @    (state==PERS ==> \old(state)==PERS) &&
      @    (state==ISSUED ==> \old(state)==PERS
      @                        || \old(state)==ISSUED
      @                        || \old(state)==CHARGING) &&
      @    (state==CHARGING ==> \old(state)==ISSUED
      @                        || \old(state)==CHARGING) &&
      @    (\old(state)==LOCKED ==> state==LOCKED) &&
      @    (\old(state)==PERS ==> state==ISSUED
      @                        || state==PERS) &&
      @    (\old(state)==ISSUED ==> state==ISSUED
      @                        || state==CHARGING
      @                        || state==LOCKED) &&
      @    (\old(state)==CHARGING ==> state==ISSUED
      @                        || state==CHARGING);
      @*/

    /*@ constraint  // by hand
      @    (\old(state) == LOCKED ==>
      @       (value == \old(value) &&
      @        counter == \old(counter) &&
      @        state == LOCKED))         &&
      @    (\old(state) == ISSUED ==>
      @       ((state == ISSUED && value <= \old(value) &&
      @        counter == \old(counter))
      @        ||
      @        (state == CHARGING && value == \old(value) &&
      @         counter == \old(counter + 1) &&
      @         \old(counter) < 4)
      @        ||
      @        (state == LOCKED && value == \old(value) &&
      @         counter == \old(counter + 1) &&
      @         \old(counter) == 4))) &&
      @   (\old(state) == CHARGING ==>
      @       ((state == ISSUED && counter == 0)
      @        ||
      @        (state == ISSUED && counter == \old(counter)
      @                        && value <= \old(value))));
      @*/
```

```
    public PayApplet() {
        state = PERS;
        value = 0;
        counter = 0;
        key = new byte[DES_KEY_SIZE];
        id = new byte[ID_SIZE];
        tmp = new byte[TMP_SIZE];
        nonce = new byte[NONCE_SIZE];
        sha_nonce = new byte[SHA_SIZE];
        plaintxt = new byte[SHA_SIZE+2];
        ciphertxt = new byte[CIPHERTEXT_SIZE];
        digest = MessageDigest.getInstance(MessageDigest.ALG_SHA,false);
        random = RandomData.getInstance(RandomData.ALG_PSEUDO_RANDOM);
    }

    public static void install(byte[] buffer, short offset, byte length) {
        PayApplet a = new PayApplet();
        a.register(buffer, (short) (offset + 1), buffer[offset]);
    }

    public boolean select() {
        return true;
    }

    /*@ normal_behavior
      @    requires
      @       apdu != null && dest != null &&
      @       apdu.buffer != null &&
      @       apdu.buffer.length > ISO7816.OFFSET_LC &&
      @       dest.length == apdu.buffer[ISO7816.OFFSET_LC] &&
      @       apdu.buffer.length >= dest.length
      @       + ISO7816.OFFSET_CDATA;
      @    assignable
      @       dest[*];
      @    ensures
      @       (\forall int i; 0 <= i && i < dest.length ==>
      @        apdu.buffer[ISO7816.OFFSET_CDATA + i] == dest[i]);
      @*/
    void readBuffer(APDU apdu, byte[] dest) {
        byte[] buffer = apdu.getBuffer();
        short readCount = apdu.setIncomingAndReceive();
        short i = (short)0;

        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA,dest,i,readCount);

        while ((short)(i + readCount) < dest.length) {
            i += readCount;
            readCount = (short)apdu.receiveBytes(ISO7816.OFFSET_CDATA);
            Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA,dest,i,readCount);
        }
    }
```

```
    /*@ behavior
      @   requires apdu != null
      @              && random != null
      @              && cipher != null
      @              && digest != null;
      @ assignable state, value, counter, apdu.buffer[*],
      @            key[*], cipher, nonce[*], id[*], tmp[*],
      @            ciphertxt[*], plaintxt[*], sha_nonce[*];
      @   ensures true;
      @ signals (ISOException e) true;
      @*/
    public void process(APDU apdu) throws ISOException {
        byte ins = apdu.getBuffer()[ISO7816.OFFSET_INS];
        if(selectingApplet())
            return;
        switch(state) {
        case PERS:
            switch(ins) {
            case INS_SETKEY:
                setKey(apdu);
                break;
            default:
                ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
            }; break;
        case ISSUED:
            switch(ins) {
            case INS_GETVAL:
                getValue(apdu);
                break;
            case INS_DECVAL:
                decValue(apdu);
                break;
            case INS_GETCHAL:
                getChallenge(apdu);
                break;
            default:
                ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
            }; break;
        case CHARGING:
            switch(ins) {
            case INS_GETVAL:
                getValue(apdu);
                break;
            case INS_DECVAL:
                decValue(apdu);
                break;
            case INS_RESPOND:
                respond(apdu);
                break;
            default:
                ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
            }; break;
        case LOCKED:
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        default:
            ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
        }
    }
```

```
  /*@ behavior
    @    requires
    @       state==PERS && apdu != null &&
    @       buffer[OFFSET_INS] == INS_SETKEY &&
    @       counter == 0;
    @    assignable
    @       state, key[*], cipher, tmp[*], id[*];
    @    ensures
    @       state==ISSUED &&
    @       ( \forall int i; 0 <= i && i < DES_KEY_SIZE ==>
    @          apdu.buffer[ISO7816.OFFSET_CDATA + i] == key[i]) &&
    @       ( \forall int j; DES_KEY_SIZE <= j &&
    @          j < DES_KEY_SIZE + ID_SIZE ==>
    @          apdu.buffer[ISO7816.OFFSET_CDATA
    @          + DES_KEY_SIZE + j] == id[j]) &&
    @       DES_KEY_SIZE + ID_SIZE == apdu.buffer[ISO7816.OFFSET_LC];
    @    signals(ISOException e)
    @       state==PERS &&
    @       DES_KEY_SIZE + ID_SIZE != apdu.buffer[ISO7816.OFFSET_LC];
    @*/
  private void setKey(APDU apdu) {
      if((DES_KEY_SIZE + ID_SIZE) == apdu.getBuffer()[ISO7816.OFFSET_LC]){
          readBuffer(apdu,tmp);
          Util.copy(tmp,(short)0,key,(short)0,DES_KEY_SIZE);
          Util.copy(tmp,DES_KEY_SIZE,id,(short)0,ID_SIZE);
          cipher = Cipher.getInstance(Cipher.ALG_DES_CBC_NOPAD, false);
          DESKey des_key = (DESKey)KeyBuilder.buildKey(KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES, true);
          des_key.setKey(key,(short)0);
          cipher.init(des_key,Cipher.MODE_DECRYPT);
          state = ISSUED;
      }
      else{
          ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
      }
  }

  /*@ normal_behavior
    @    requires
    @       (state==ISSUED || state==CHARGING)
    @       && buffer[OFFSET_INS] == INS_GETVAL
    @       && apdu != null;
    @    assignable
    @       state, apdu.buffer[ISO7816.OFFSET_CDATA],
    @       apdu.buffer[ISO7816.OFFSET_CDATA + 1];
    @    ensures
    @       state==ISSUED;
    @*/
  private void getValue(APDU apdu) {
      state = ISSUED;
      apdu.setOutgoing();
      apdu.setOutgoingLength((short)2);
      byte[] buffer = apdu.getBuffer();
      buffer[ISO7816.OFFSET_CDATA] = (byte)(value >> 8);
      buffer[ISO7816.OFFSET_CDATA + 1] = (byte)value;
  }
```

```
    /*@ behavior
      @    requires
      @        (state==ISSUED || state==CHARGING) &&
      @        buffer[OFFSET_INS] == INS_DECVAL;
      @    assignable
      @       state, value;
      @    ensures
      @       state==ISSUED
      @         && \old(value) > 0
      @         && value==\old(value)-1;
      @    signals (ISOException e)
      @       state==ISSUED
      @         && \old(value) == 0
      @         && value==\old(value);
      @*/
    private void decValue(APDU apdu) throws ISOException {
        state = ISSUED;
        if (value > 0) {
            value--;
        }
        else {
            ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
        }
    }

    /*@ behavior
      @    requires
      @       state == ISSUED
      @         && buffer[OFFSET_INS] == INS_GETCHAL
      @         && counter < 5
      @         && apdu != null
      @         && apdu.buffer != null
      @         && apdu.buffer.length >= ISO7816.OFFSET_CDATA
      @            + NONCE_SIZE + ID_SIZE
      @         && random != null;
      @    assignable
      @       state, counter, nonce[*], apdu.buffer[*];
      @    ensures
      @       state == CHARGING
      @         && counter == \old(counter) + 1
      @         && counter < 5;
      @    signals (ISOException e)
      @       state == LOCKED && counter == 5;
      @*/
    private void getChallenge(APDU apdu) throws ISOException {
        counter++;
        if (counter<5) {
            state = CHARGING;
            random.generateData(nonce,(short)0,NONCE_SIZE);
            apdu.setOutgoing();
            apdu.setOutgoingLength((short)(NONCE_SIZE+ID_SIZE));
            byte[] buffer = apdu.getBuffer();
            Util.arrayCopy(nonce,(short)0, buffer, ISO7816.OFFSET_CDATA,NONCE_SIZE);
            Util.arrayCopy(id, (short)0, buffer, (short)(ISO7816.OFFSET_CDATA+NONCE_SIZE),ID_SIZE);
        }
        else {
            state = LOCKED;
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        }
    }
```

```
    /*@ behavior
      @    requires
      @       state == CHARGING
      @         && buffer[OFFSET_INS] == INS_RESPOND
      @         && counter < 5
      @         && apdu != null
      @         && apdu.buffer != null
      @         && apdu.buffer.length >= CIPHERTEXT_SIZE
      @         && cipher != null
      @         && ciphertxt != null
      @         && digest != null;
      @    assignable
      @       state, counter, value, ciphertxt[*],
      @       plaintxt[*], sha_nonce[*];
      @    ensures
      @       state == ISSUED
      @         && value == (short)(256 *
      @                     (plaintxt[SHA_SIZE] & 0x0F) +
      @                     (plaintxt[SHA_SIZE+1] & 0xFF))
      @         && counter == 0
      @         && (\forall int i; 0 <= i && i < SHA_SIZE ==> sha_nonce[i] == plaintxt[i]);
      @    signals (ISOException e)
      @       state == ISSUED
      @         && value == \old(value)
      @         && counter == \old(counter)
      @         && (apdu.buffer[ISO7816.OFFSET_LC] != ciphertxt.length
      @            ||
      @            !(\forall int i; 0 <= i && i < SHA_SIZE ==> sha_nonce[i] == plaintxt[i]));
      @*/
    private void respond(APDU apdu) throws ISOException {
        state = ISSUED;
        byte[] buffer = apdu.getBuffer();
        if (buffer[ISO7816.OFFSET_LC] != ciphertxt.length) {
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        };
        readBuffer(apdu,ciphertxt);
        cipher.doFinal(ciphertxt,(short)0, CIPHERTEXT_SIZE,plaintxt,(short)0);
        digest.doFinal(nonce,(short)0, NONCE_SIZE,sha_nonce,(short)0);
        if (Util.arrayCompare(sha_nonce,(short)0, plaintxt,(short)0,SHA_SIZE)==0) {
            value = (short)(((plaintxt[SHA_SIZE] & 0x0F) << 8) | (plaintxt[SHA_SIZE+1] & 0xFF));
            counter = 0;
        }
        else {
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        }
    }
}
```

A.2 The cash-logger from chapter 5

```
class PaymentsLog {

    private /*@ spec_public @*/ int size, counter;
    private /*@ spec_public @*/ int[] payments;

    /* @
      @ invariant
      @   size > 0 && counter > 0 && counter <= size &&
      @   payments != null && payments.length == size &&
      @   (\forall int i; i > counter && i < size ==> payments[i] == 0);
      @*/


    /*@
      @ constraint
      @   size == \old(size) && payments == \old(payments);
      @*/

    /*@ requires s > 0; */
    public PaymentsLog(int s) {
        size = s;
        counter = 1;
        payments = new int[s];
        payments[0] = (int) (Integer.MAX_VALUE * Math.random());
    }

    /*@
      @ normal_behavior
      @   requires true;
      @ assignable \nothing;
      @    ensures \result == (counter == size);
      @*/
    public boolean isFull() {
        return counter == size;
    }


    /*@
      @ behavior
      @   requires counter > 0;
      @ assignable counter, payments[counter];
      @    ensures amount > 0 && \old(counter) < size &&
      @              counter == \old(counter) + 1 &&
      @              payments[counter -1] ==
      &                       \old(payments[counter - 1] + amount)  &&
      @              payments[counter] ==  payments[\old(counter)]  &&
      @              \result == \old(money) - \old(amount) &&
      @              \result >= 0;
      @ signals(NotEnoughMoneyException) \old(amount) > \old(money) &&
      @                                  payments[counter - 1] ==
      @                                  \old(payments[counter -1]) &&
      @                                  counter == \old(counter);
      @ signals(LogFullException) (\old(counter) >= \old(size)  ||
      @                           amount <= 0) &&
      @                           counter == \old(counter) &&
      @                           payments[counter -1] ==
      @                               \old(payments[counter -1]);
      @ signals(OverflowException)
      @   amount > \old(Integer.MAX_VALUE - payments[counter - 1] &&
      @   counter == \old(counter) &&
      @   payments[counter] == \old(payments[counter]);
      @*/
```

```
    public int add(int amount, int money) throws NotEnoughMoneyException,
                                                 LogFullException, OverflowException{
65          if (amount > money)
                NotEnoughMoneyException.throwIt();
            if (amount > 0 && counter < size ) {
                if(payments[counter-1] > Integer.MAX_VALUE - amount) //@ nowarn Null; nowarn IndexTooBig;
                    OverflowException.throwIt();
70              payments[counter] = //@ nowarn IndexTooBig;
                    // overflow possible:
                    payments[counter-1] + amount;
                counter++;
            }
75          else LogFullException.throwIt();
            return money - amount;
        }


80      /*@
          @ normal_behavior
          @   requires true;
          @ assignable counter, payments[*];
          @    ensures \result != null && \result != payments &&
85        @            \result.length == size &&
          @            counter == 0 &&
          @            (\forall int i; i >= 0 && i < size ==>
          @                \result[i] == \old(payments[i]) &&
          @                payments[i] == 0);
90        @*/
        public int[] dump() {
            int[] dmp = new int[size];
            for(int i = 0; i < size; i++) {
                dmp[i] = payments[i]; //@ nowarn Null; nowarn IndexTooBig; nowarn IndexNegative ;
95              payments[i] = 0;
            };
            counter = 0;
            return dmp;
        }
100


    }

105
    class Casher {

        public boolean locked;
        private /*@ spec_public */ PaymentsLog log;
110
        /*@
          @ constraint
          @   log == \old(log);
          @*/
115
        /*@
          @ invariant
          @   log != null;
          @*/
```

```
120     /*@
          @ behavior
          @   requires log.counter > 0;
          @ assignable locked, log.counter, log.payments[*];
          @ ensures (\old(!locked) && (log.size != \old(log.counter))) ==>
125       @              (amount > 0 && \old(log.counter) < log.size  &&
          @              log.counter == \old(log.counter) + 1 &&
          @              (\forall int i; i > 0 && i < log.size ==>
          @                   (i == \old(log.counter))
          @                     ? (log.payments[i] == \old(log.payments[i-1] + amount))
130       @                     : (log.payments[i] == \old(log.payments[i]))) &&
          @              \result == \old(money) - \old(amount) &&
          @              (\old(log.counter) == log.size -1 ==> locked));
          @ signals(NotEnoughMoneyException) \old(amount) > \old(money) &&
          @                                  (\forall int i;i >=0 && i < log.size ==>
135       @                                  log.payments[i] == \old(log.payments[i])) &&
          @                                  log.counter == \old(log.counter);
          @ signals(LogFullException) (\old(log.counter) >= \old(log.size)  || amount <= 0) &&
          @                           log.counter == \old(log.counter) &&
          @                           (\forall int i;i >=0 && i < log.size ==>
140       @                            log.payments[i] == \old(log.payments[i]));
          @ signals(OverflowException)
          @   \old(log.payments[log.counter - 1]) > (Integer.MAX_VALUE - \old(amount)) && // Conf. fails
          @   log.counter == \old(log.counter) &&
          @   (\forall int i;i >=0 && i < log.size ==>
145       @   log.payments[i] == \old(log.payments[i]));
          @
          @*/
        public int deposit (int amount, int money) throws NotEnoughMoneyException,
                                                           LogFullException, OverflowException{
150         int returnvalue = 0;
            if ( !locked && !log.isFull()) { //@ nowarn Null;
                returnvalue = log.add(amount,money);
                if ( log.isFull() ) locked = true;
            }
155         return returnvalue;
        }

        /*@
          @ behavior
          @   requires true;
160       @ assignable locked, log.counter, log.payments[*];
          @   ensures \old(locked) ==> (!locked &&
          @           \result != null && \result != log.payments &&
          @            \result.length == log.size &&
165       @            log.counter == 0 &&
          @            (\forall int i; i >= 0 && i < log.size ==>
          @                \result[i] == \old(log.payments[i]) &&
          @                log.payments[i] == 0));
          @ signals(LogNotLockedException) \old(locked) == locked && !locked;
170       @*/
        public int[] unlock() throws LogNotLockedException{
            if (locked)
                locked = false;
            else LogNotLockedException.throwIt();
175         /*@ assert
              @  (!locked) && \old(locked) &&
              @  log.counter == \old(log.counter) &&
              @  (\forall int i; i>=0 && i < log.size ==>
              @   log.payments[i] == \old(log.payments[i]));
180           @*/
            return log.dump();
        }
    }
```

```
class LogNotLockedException extends Exception{

    public static  final LogNotLockedException lnle = new LogNotLockedException();

    private LogNotLockedException(){
        super();
    }

    /*@ exceptional_behavior
      @  assignable \nothing;
      @ signals(LogNotLockedException) true;
      @*/
    public static void throwIt() throws LogNotLockedException{
        throw lnle;
    }

}

class NotEnoughMoneyException extends Exception{

    public static  final NotEnoughMoneyException neme = new NotEnoughMoneyException();

    private NotEnoughMoneyException(){
        super();
    }


    /*@ exceptional_behavior
      @  assignable \nothing;
      @ signals(NotEnoughMoneyException) true;
      @*/
    public static void throwIt() throws NotEnoughMoneyException{
        throw neme;
    }
}

class LogFullException extends Exception{

    public static  final LogFullException lfe = new LogFullException();

    private LogFullException(){
        super();
    }

    /*@ exceptional_behavior
      @  assignable \nothing;
      @ signals(LogFullException) true;
      @*/
    public static void throwIt() throws LogFullException{
        throw lfe;
    }
}
```

```
class OverflowException extends Exception{

    public static final OverflowException oe = new OverflowException();

    private OverflowException(){
        super();
    }

    /*@ exceptional_behavior
      @  assignable \nothing;
      @ signals (OverflowException) true;
      @*/
    public static void throwIt() throws OverflowException{
        throw oe;
    }
}
```

Appendix B

Formal properties of OO

B.1 Operational semantics of OO and OO with exceptions

In this section we give the timed operational semantics of OO. The set of values of OO is defined as $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{null\}$, where $\mathcal{L}$ is an (infinite) set of locations, $x \in \mathcal{X}$, where $\mathcal{X}$ is a set of local variables, $n \in \mathbb{Z}$ and o is an object from a set, namely $\mathcal{O}$.

The set State of OO states is defined as the set of pairs $\langle sf, h \rangle$ where sf is a stack of frames, and h is a heap.

A frame is of the form $\langle s, \rho \rangle$ where s is in Stat, ρ is a mapping from local variables from a set of variables $\mathcal{X}$ to values. We distinguish a special variable res to store results of execution of programs.

Heaps are modeled as a partial function $h : \mathcal{L} \rightharpoonup \mathcal{O}$, where the set $\mathcal{O}$ of objects is modeled as $\mathcal{F} \rightharpoonup \mathcal{V}$, i.e. as the set of finite functions from $\mathcal{F}$ to $\mathcal{V}$. We let Heap be the set of heaps and R be the set of all variable mappings. We further use $\sqsubseteq: \mathcal{C} \to \mathcal{C} \to \mathbb{B}$ to denote subclass relation and the functions static : $\mathcal{O} \to \mathcal{C}$ and dynamic : $\mathcal{O} \to \mathcal{C}$ give the static and dynamic type of an object respectively, the function cdynamic : $\mathcal{C} \times \mathcal{O} \to \mathcal{O}$ assigns a (new) dynamic type to an object. Furthermore we have an allocator function fresh : Heap $\times \mathcal{C} \to \mathcal{L}$ and a function default : $\mathcal{C} \to \mathcal{O}$ which puts a new object on the heap. The operational semantics of OO can be found in Figure B.1. Note that we assume that the object `this` remains unmodified during the execution of a program.

The relation $\rightsquigarrow$ such that $\rightsquigarrow \subseteq T \times$ State $\times$ ((`skip` $\cup \mathcal{V}$) $\times$ R $\times$ Heap) formalizes –besides the operational semantics–the *execution time* of OO. Figure B.1 shows both *quantified* variables, such as h, or ρ, and *constants*, such as t. The set of constants includes constant executions times, namely t_{R_1}, t_{R_2}, t_{OP}, $t_{:=}$, t_W, t_N, t_C , for different statements and operations.

We extend the semantics of OO with exceptions in Figure B.2. The exceptional states of the form $< exc > s$ where $s \in$ State are used for propagation of exceptions, that is every time that a subexpression or substatement is evaluated to a state of the form $< exc > s$, this state is propagated to the containing expression or statement, or in the case of method calls, if a method evaluates to an exceptional states, the exception is propagated to the caller method.

B.2 Non-interference for OO

Defining non-interference for a language with dynamic object creation is somewhat more involved then the relatively straightforward definition of non-interference for a simple stack based language (such as defined in Definition 20).

We assume $\mathsf{sl} : \mathsf{Loc} \to \Sigma$ with $\mathsf{Loc} = \mathcal{V} \cup \mathcal{L}$, $\Sigma = \{\mathsf{High}, \mathsf{Low}\}$ and $\mathsf{Low} \sqsubseteq \mathsf{High}$ as usual. Furthermore, it is assumed that objects that are created 'under a high context'[76] are considered to be indistinguishable for an attacker[77]. We follow Banerjee and Naumann [BN05] and define non-interference relative to a partial bijection β on locations:

Definition 24 (non-interference)**.**

i. Value indistinguishability $v \sim_{\beta,\sigma} v'$ where $v, v' \in \mathcal{V}$ and $\sigma \in \Sigma$, is defined by the clauses:

$$\texttt{null} \sim_{\beta,\mathsf{Low}} \texttt{null} \qquad v \sim_{\beta,\mathsf{High}} v' \qquad \frac{v \in \mathbb{Z}}{v \sim_{\beta,\mathsf{Low}}} \qquad \frac{v, v' \in \mathcal{L} \quad \beta(v) = v'}{v \sim_{\beta,\mathsf{Low}} v'}$$

ii. Two heaps h_1 and h_2 are indistinguishable (relative to β), written $h_1 \sim_\beta h_2$, if:

   - $dom(\beta) \subseteq dom(h_1)$ and $rng(\beta) \subseteq (h_2)$
   - for every $o \in dom(\beta)$, we have $dom(h_1)(o) = dom(h_2)(\beta(o))$
   - for every $f \in dom(h_1)(o)$, we have $h_1(o)(f) \sim_{\mathsf{sl}(f)} h_2(\beta(o))(f)$

iii. A program P is termination-insensitive non-interfering, if for every $\rho, \rho' \in \mathcal{X} \to \mathcal{V}$, and $h, h', h_f, h'_f \in \mathsf{Heap}$ and $v, v' \in \mathcal{V}$, and partial bijection on locations β, we have $P, \rho, h \Downarrow_t v, h_f$ and $P, \rho', h' \Downarrow_{t'} v', h'_f$ and $\rho \sim_\beta \rho'$ and $h \sim_\beta h'$ imply $h_f \sim_{\beta'} h'_f$ and $v \sim_{\beta'} v'$ for some partial bijection $\beta' \subseteq \beta$

iv. A program P is termination-sensitive non-interfering, if for every $\rho, \rho' \in \mathcal{X} \to \mathcal{V}$, and $h, h', h_f, h'_f \in \mathsf{Heap}$ and $v, v' \in \mathcal{V}$, and partial bijection on locations β, we have:

   - $P, \rho, h \Downarrow_t v, h_f$ and $\rho \sim_\beta \rho'$ and $h \sim_\beta h'$ imply $P, \rho', h' \Downarrow_{t'} v', h'_f$ and $h_f \sim_{\beta'} h'_f$ and $v \sim_{\beta'} v'$ for some partial bijection $\beta' \subseteq \beta$, *or*
   - $P, \rho, h \Downarrow_t \Uparrow$ and $\rho \sim_\beta \rho'$ and $h \sim_\beta h'$ imply $P, \rho', h' \Downarrow_{t'} \Uparrow$.

v. A program P is time-sensitive termination-sensitive non-interfering, if item iv above is valid and $t = t'$.

Where $\Uparrow$ denotes non-termination

[76] I.e., an object is 'created under a high context' if the expression that creates the object has security level High or if this expression is under the influence of a conditional with security level High. In the context of Chapter 6 this means that 'an object is created under a high context' if the environment level lenv has a high security level (is not $\bot$).

[77] This assumption also implies that we do not consider memory consumption, which forms another resource that can be used as covert channel that leaks sensitive information.

$$\frac{}{\langle x, \rho\rangle :: sf, h \rightsquigarrow_{t_{R_1}} \langle \rho(x), \rho\rangle :: sf, h} \qquad \frac{h(o).f = v}{\langle o.f, \rho\rangle :: sf, h \rightsquigarrow_{t_{R_2}} \langle v, \rho\rangle :: sf, h}$$

$$\frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t \langle e', \rho\rangle :: sf, h'}{\langle e.f, \rho\rangle :: sf, h \rightsquigarrow_t \langle e'.f, \rho\rangle :: sf, h'} \qquad \frac{v \ op \ v' = v''}{\langle v \ op \ v', \rho\rangle :: sf, h \rightsquigarrow_{t_{OP}} \langle v'', \rho\rangle :: sf, h}$$

$$\frac{\langle e_1, \rho\rangle :: sf, h \rightsquigarrow_t \langle e_1', \rho\rangle :: sf, h'}{\langle e_1 \ op \ e_2, \rho\rangle :: sf, h \rightsquigarrow_t \langle e_1' \ op \ e_2, \rho\rangle :: sf, h'} \qquad \frac{\langle e_2, \rho\rangle :: sf, h \rightsquigarrow_t \langle e_2', \rho\rangle :: sf, h'}{\langle v \ op \ e_2, \rho\rangle :: sf, h \rightsquigarrow_t \langle v \ op \ e_2', \rho\rangle :: sf, h'}$$

$$\frac{}{\langle x := v, \rho\rangle :: sf, h \rightsquigarrow_{t_:=} \langle \texttt{skip}, \rho \oplus \{x \mapsto v\}\rangle :: sf, h} \qquad \frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t \langle e', \rho\rangle :: sf, h'}{\langle x := e, \rho\rangle :: sf, h \rightsquigarrow_t \langle x := e', \rho, h'\rangle}$$

$$\frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t \langle e', \rho\rangle :: sf, h'}{\langle x := e.m(\vec{e}), \rho\rangle :: sf, h \rightsquigarrow_t \langle x := e'.m(\vec{e}), \rho\rangle :: sf, h'}$$

$$\frac{\langle \vec{e}, \rho\rangle :: sf, h \rightsquigarrow_t \langle \vec{e'}, \rho\rangle :: sf, h'}{\langle x := o.m(\vec{e}), \rho\rangle :: sf, h \rightsquigarrow_t \langle x := o.m(\vec{e'}), \rho\rangle :: sf, h'}$$

$$\frac{m(\vec{x}) := s}{\langle x := o.m(\vec{v}), \rho\rangle :: sf, h \rightsquigarrow_t \langle s, \vec{x} \mapsto \vec{v}\rangle :: \langle x := res, \rho\rangle :: sf, h}$$

$$\frac{\langle s_1, \rho\rangle :: sf, h \rightsquigarrow_t \langle s_1', \rho'\rangle :: sf, h'}{\langle s_1; s_2, \rho\rangle :: sf, h \rightsquigarrow_t \langle s_1'; s_2, \rho'\rangle :: sf', h'}$$

$$\frac{\langle x := o.m(\vec{v}), \rho\rangle :: sf, h \rightsquigarrow_t \langle s, \rho'\rangle :: \langle x := res, \rho\rangle :: sf, h'}{\langle x := o.m(\vec{v}); s_2, \rho\rangle :: sf, h \rightsquigarrow_t \langle s, \rho'\rangle :: \langle x := res; s_2, \rho'\rangle :: sf', h'}$$

$$\frac{}{\langle \texttt{skip}; s_2, \rho\rangle :: sf, h \rightsquigarrow_0 \langle s_2, \rho'\rangle :: sf, h'}$$

$$\frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t \langle true, \rho\rangle :: sf, h'}{\langle \textsf{while } e \textsf{ do } s, \rho\rangle :: sf, h \rightsquigarrow_t \langle s; \textsf{while } e \textsf{ do } s, \rho\rangle :: sf, h'} \qquad \frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t \langle false, \rho\rangle :: sf, h'}{\langle \textsf{while } e \textsf{ do } s, \rho\rangle :: sf, h \rightsquigarrow_t \langle \texttt{skip}, \rho\rangle :: sf, h'}$$

$$\frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t \langle true, \rho\rangle :: sf, h'}{\langle \textsf{if } e \textsf{ then } s_1 \textsf{ else } s_2, \rho\rangle :: sf, h \rightsquigarrow_t \langle s_1, \rho\rangle :: sf, h'}$$

$$\frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t \langle false, \rho\rangle :: sf, h'}{\langle \textsf{if } e \textsf{ then } s_1 \textsf{ else } s_2, \rho\rangle :: sf, h \rightsquigarrow_t \langle s_2, \rho\rangle :: sf, h'}$$

$$\frac{o \in dom(h) \qquad f \in dom(h(o))}{\langle o.f := v, \rho\rangle :: sf, h \rightsquigarrow_{t_W} \langle \texttt{skip}, \rho, h \oplus \{o \mapsto h(o) \oplus \{f \mapsto v\}\}\rangle} \qquad \frac{\langle e_2, \rho\rangle :: sf, h \rightsquigarrow_t \langle e_2', \rho, h'\rangle}{\langle o.f := e_2, \rho\rangle :: sf, h \rightsquigarrow_t \langle o.f := e_2', \rho, h'\rangle}$$

$$\frac{\langle e_1, \rho\rangle :: sf, h \rightsquigarrow_t \langle e_1', \rho, h'\rangle}{\langle e_1.f := e_2, \rho\rangle :: sf, h \rightsquigarrow_t \langle e_1'.f := e_2, \rho\rangle :: sf, h'} \qquad \frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t \langle e', \rho\rangle :: sf, h'}{\langle \textsf{return } e, \rho\rangle :: sf, h \rightsquigarrow_t \langle \textsf{return } e', \rho, h'\rangle}$$

$$\frac{}{\langle \textsf{return } v, \rho\rangle :: \langle c, \rho\rangle :: sf, h \rightsquigarrow_{t_R + t_:=} \langle c, \rho \oplus \{res \mapsto v\}\rangle :: sf, h} \qquad \frac{}{\langle \textsf{return } v, \rho\rangle :: \texttt{skip}, h \rightsquigarrow_{t_R} \langle v, \rho\rangle :: sf, h}$$

$$\frac{o = \textsf{fresh}(h, C)}{\langle \textsf{new } C, \rho\rangle :: sf, h \rightsquigarrow_{t_N} \langle o, \rho\rangle :: sf, h \oplus \{o \mapsto \textsf{default}_C\}}$$

$$\frac{\textsf{dynamic}(o) \sqsubseteq C}{\langle (C)o, \rho\rangle :: sf, h \rightsquigarrow_{t_C + t_W} \langle o, \rho\rangle :: sf, h' \oplus \{o \mapsto \textsf{cdynamic}(C)\}}$$

$$\frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t \langle e', \rho\rangle :: sf, h'}{\langle (C)e, \rho\rangle :: sf, h \rightsquigarrow_t \langle (C)e', \rho\rangle, h'}$$

Figure B.1: Small step operational semantics for OO.

$$\frac{\langle e_1, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}{\langle e_1\ op\ e_2, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s, h'}$$

$$\frac{\langle e_2, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}{\langle v\ op\ e_2, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}$$

$$\frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}{\langle (C)e, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}$$

$$\frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}{\langle e.f, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}$$

$$\frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf', h'}{\langle x := e, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}$$

$$\frac{\langle e, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}{\langle \mathsf{return}\ e, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}$$

$$\frac{o = \mathsf{fresh}(h', \mathsf{Throwable})}{\langle \mathsf{null}.f := e_2, \rho\rangle :: sf, h \rightsquigarrow_{t_N} < exc > \langle o, \rho\rangle, h \oplus \{o \mapsto \mathsf{default}_{\mathsf{Throwable}}\}}$$

$$\frac{o = \mathsf{fresh}(h', \mathsf{Throwable})}{\langle \mathsf{null}.m(\vec{e}), \rho\rangle :: sf, h \rightsquigarrow_{t_N} < exc > \langle o, \rho\rangle, h' \oplus \{o \mapsto \mathsf{default}_{\mathsf{Throwable}}\}}$$

$$\frac{o = \mathsf{fresh}(h, \mathsf{Throwable})}{\langle \mathsf{Throw}, \rho\rangle :: sf, h \rightsquigarrow_{t_N} < exc > \langle o, \rho\rangle, h \oplus \{o \mapsto \mathsf{default}_{\mathsf{Throwable}}\}}$$

$$\frac{\langle s_1, \rho\rangle :: sf, h \rightsquigarrow_t < exc > \langle o, \rho'\rangle :: sf, h'}{\langle \mathsf{try}\{s_1\}\ \mathsf{catch}(\mathsf{Exception\ x})\{s_2\}, \rho\rangle :: sf, h \rightsquigarrow_{t+t_{:=}} \langle s_2, \rho' \oplus \{x \mapsto o\}\rangle :: sf, h'}$$

$$\frac{\langle s_1, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}{\langle s_1; s_2, \rho\rangle :: sf, h \rightsquigarrow_t < exc > s :: sf, h'}$$

Figure B.2: Operational semantics for OO with exceptions.

List of Figures

Index

Samenvatting (Dutch summary)

Computer programma's bevatten fouten. Het is belangrijk om deze fouten te vinden. Vooral als programma's gebruikt worden in situaties waar de consequenties van zulke fouten verstrekkend kunnen zijn, zoals programma's die voor financiële transacties of in de boordcomputer van een vliegtuig worden gebruikt. Dit proefschrift behandelt een aantal analyse methodes die garanderen dat wiskundig kan worden vastgesteld dat bepaalde fouten niet voorkomen in een programma.

De programmeertaal Java staat centraal in dit proefschrift. De voorgestelde analyse-methodes werken op de broncode van programma's die in deze taal geschreven zijn. Het eerste gedeelte van het proefschrift behandelt het gebruik van JML, een specificatietaal voor Java. In JML is het mogelijk om –op een formele manier– op te schrijven wat de beoogde functionaliteit van een programma is. Speciale programma's, zoals de LOOP tool en ESC/Java2, kunnen worden gebruikt om te bewijzen dat JML specificaties *correct* zijn voor een specifiek programma.

Het eerste deel van het proefschrift laat zien hoe JML-specificaties formeel correct kunnen worden bewezen voor een aantal kleinere, maar semantische complexe, Java programma's. Vervolgens wordt er getoond hoe een Java Card programma –een Java programma dat draait op een smart card– dat een elektronische portemonnee implementeert op een structurele manier kan worden ontwikkeld. Bepaalde gewenste eigenschappen, zoals dat de portemonnee geen negatief saldo kan bevatten, kunnen dan worden gegarandeerd. Met behulp van JML specificaties worden deze en andere eigenschappen van de elektronische portemonnee formeel bewezen

Het tweede deel van het boek behandelt het probleem van veilige informatie stromen in programma's. We willen bewijzen dat geheime waarden in een applicatie, bijvoorbeeld een PIN-code, volledig onafhankelijk zijn van niet geheime (publieke) waarden, zoals bijvoorbeeld het saldo van een elektronische beurs. Dit probleem mag in eerste instantie makkelijk lijken, maar als we de onafhankelijkheid van *alle* geheime waardes ten opzichte van publieke waardes in een programma op een mathematische manier willen vaststellen blijken er vele moeilijkheden op te treden.

Een eerste aanpak van dit probleem bestaat eruit dat we JML gebruiken om de afhankelijkheden tussen waardes (variabelen) in Java programma's aan te geven. Dit is in principe mogelijk omdat we in JML voor elke aparte methode (subroutine) kunnen aangeven hoe variabelen van elkaar afhangen. Met behulp van een specificatie-patroon kunnen zulke afhankelijkheden op een gestructureerde manier worden weergegeven. Vervolgens kan de specificatie met een aparte tool worden gecheckt en kan worden vastgesteld of er ongewenste afhankelijkheden tussen geheime en publieke waardes bestaan. Een nadeel van deze methode is dat de specificaties erg groot en complex kunnen worden, zelfs voor kleinere programma's.

Een alternatieve aanpak wordt in het volgende hoofstuk uitgewerkt: Het programma wordt op een abstract niveau geëvalueerd. Dit wil zeggen dat we niet kijken hoe het (computer) geheugen

verandert door de executie van een programma, maar hoe het veiligheidsniveau –bevat een variabele een geheime of een publieke waarde?– van een programma verandert tijdens de executie. Dit resulteert in een methode, gebaseerd op de wiskundige techniek van abstracte interpretatie, die automatisch kan bepalen of een programma geschreven in een subset van Java informatie lekt (door een afhankelijkheid tussen geheime en publiek variabelen). De beschreven methode is correct bewezen met behulp van de stellingbewijzer PVS.

De voorgaande methodes kunnen een onafhankelijkheid tussen geheime en publieke variabelen aantonen in het geval dat een programma normaal tot zijn einde komt (termineert). Maar programma's kunnen ook weigeren om te termineren (hangen) of kunnen abrupt stoppen, bijvoorbeeld als er een fout optreed omdat een programma door nul probeert te delen. Als we zulke alternatieve terminatiemogelijkheden van een programma ook willen meenemen in onze analyse dan wordt het onderliggende semantisch model (wiskundige beschrijving) van de programmeertaal ingewikkelder. Om deze reden introduceren we een methode gebaseerd op een nieuwe Hoare logica. Met deze methode is het mogelijk om afhankelijkheden tussen variabelen aan te tonen die niet op predikaten, maar op relaties werken. Deze relationele Hoare logica is geïmplementeerd in de stellingbewijzer PVS en correct bewezen ten opzichte van de formele semantiek van Java zoals die ontwikkeld is binnen het LOOP project. De logica kan gebruikt worden om onafhankelijkheden tussen variabelen aan te tonen voor programma's die geschreven zijn in (single threaded) Java. Omdat het hier om een interactieve methode gaat –een mens moet de bewijzen met behulp van de relationele Hoare logica *zelf* construeren– is de methode feitelijk vooral geschikt om kleine kritieke componenten van een programma te analyseren.

Programma's kunnen ook op andere manieren geheime informatie lekken. Eén zo'n manier is via het tijdsgedrag van een programma. Zo is het bijvoorbeeld soms mogelijk om met tijdswaarneming een pincode te achterhalen. Het proefschrift eindigt met een voorstel dat dit probleem behandelt. Een programmatransformatie wordt geïntroduceerd die afdwingt dat het getransformeerde programma geen informatie via verschillen in executietijd lekt. Onder bepaalde aannames bewijzen we dat deze transformatie correct is (het getransformeerde programma behoudt dezelfde functionaliteit) en geen informatie lekt via verschillen in executietijd.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty

of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

9 789090 209227

ISBN-10: 90-9020922-0