UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Abstract Certification of Java Programs in Rewriting Logic

Ph.D. Thesis

Presented by:
Mauricio Alba Castro

Supervisors:
María Alpuente Frasnedo
Santiago Escobar Román

November 2011

DEPARTAMENTO DE SISTEMAS
INFORMÁTICOS Y COMPUTACIÓN

Abstract Certification of Java Programs in Rewriting Logic

Ph.D. Thesis

A dissertation submitted by Mauricio Alba Castro in fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science at the Universitat Politècnica de València.

November 2011

Title:	Abstract Certification of Java Programs in Rewriting Logic
Author:	Mauricio Alba Castro
Address:	Departamento de Ciencias Computacionales Universidad Autónoma de Manizales, Colombia Antigua estación del ferrocarril, Manizales, Colombia
Email:	malba@autonoma.edu.co
Date:	November, 2011

This work has been funded by:

1. Universidad Autónoma de Manizales UAM, Colombia, which granted me a license to do my PhD thesis abroad.

2. LERNet AML/19.0902/97/0666/II-0472-FA. LERNet is an ALFA project approved by the European Community that foresees PhD student mobility from Latin America to the European Union, and vice versa. I am grateful to the Universidad Eafit, Medellín, Colombia, particularly Francisco Correa, for his support in order to obtain this grant.

3. The Latin America Cooperation Program of the Universitat Politècnica de València (internships for PhD thesis, year 2011) with the grant aid from the Generalitat Valenciana International Development Cooperation Program 3015/2010.

This work has also received support from the following research projects:

TIN2004-7943-C04-01, and TIN2007-68093-C02-02, funded by EU (FEDER) and the Spanish MEC/ MICINN.

HA 2006-0007 Integrated Action funded by Deutsch & Spanish Science Research Councils.

GV06/285, and GVPRE/2008/113, funded by Generalitat Valenciana.

Abstract Certification of Java Programs in Rewriting Logic

Ph.D. Thesis in Computer Science

Presented by

Mauricio Alba Castro

Supervisors

María Alpuente Frasnedo and Santiago Escobar Román
Universidad Politécnica de Valencia

External evaluators

Claude Marché	INRIA Saclay & LRI (Équipe ProVal)
Ricardo Peña	Universidad Complutense de Madrid
Francisco Durán	Universidad de Malaga

Jury

Francisco Durán	Universidad de Malaga
Salvador Lucas	Universidad Politécnica de Valencia (President)
Claude Marché	INRIA Saclay & LRI (Équipe ProVal)
Ricardo Peña	Universidad Complutense de Madrid
Alicia Villanueva	Universidad Politécnica de Valencia (Secretary)

To my beloved ones.

Acknowledgments

I am truly grateful to María Alpuente for introducing me to the ELP group and giving me the chance to begin the scientific journey to this dissertation under her experienced advise. I would like to thank Santiago Escobar, for his supervision and unchanged good mood; they, María and Santiago, both make up the perfect advisory team for this sometimes long and hard, and, most times enjoyable but always productive, journey.

I would also like to thank the ELP professors Salva, Javier, Marisa and Alicia; they allowed me to learn from them, not only when I was attending their classes, but anytime. I also thanks the research fellows/Ph.D. students of the ELP group: Dani, Raúl, Bea, Sonia Flores, Toni, Gustavo, Alexei, Diego, Pepe, Ana, Aristides, Rafa, Pedro, Sonia, Marco, Michele, Nando, ... It has been a pleasure to share a cup of coffee, a lunch, a dinner, a "picaeta", a bowling game, a trip or even just few moments with all of them. Also German, Christophe, María José, Cesar, Josep, Carlos and Tama for being so kind.

Special mention goes to those that were not just workmates but also friends: Dani, Sonia Flores, Rosa, Gustavo, Alexei and Pedro.

I would like to thank the anonymous reviewers of all paper submissions derived from this thesis. I would also like to thank my dissertation reviewers. Their comments allowed me to enrich my knowledge and to be one step closer to the goals of this thesis.

Finally, I would like to thank the Colombian people that contributes to make this long trip possible and enjoyable: Pau, Gloria, Eu, Jaime, Martha, Pacho, Gustavo,....

¡Gracias a todos !

Abstract

In this thesis we propose an abstraction based certification technique for Java programs which is based on rewriting logic, a very general logical and semantic framework efficiently implemented in the functional programming language Maude. We focus on safety properties, i. e. properties of a system that are defined in terms of certain events not happening, which we characterize as unreachability problems in rewriting logic. The safety policy is expressed in the style of JML, a standard property specification language for Java modules. In order to provide a decision procedure, we enforce finite-state models of programs by using abstract interpretation. Starting from a specification of the Java semantics written in Maude, we develop an abstraction based, finite-state operational semantics also written in Maude which is appropriate for program verification. As a by-product of the verification based on abstraction, a dependable safety certificate is delivered which consists of a set of rewriting proofs that can be easily checked by the code consumer by using a standard rewriting logic engine. The abstraction based proof-carrying code technique, called JavaPCC, has been implemented and successfully tested on several examples, which demonstrate the feasibility of our approach.

We analyze local properties of Java methods: i. e. properties of methods regarding their parameters and results. We also study global confidentiality properties of complete Java classes, by initially considering non–interference and, then, erasure with and without non–interference. Non–interference is a semantic program property that assigns confidentiality levels to data objects and prevents illicit information flows from occurring from high to low security levels. In this thesis, we present a novel security model for global non–interference which approximates non–interference as a safety property. Erasure is a way of strengthening confidentiality by upgrading data confidentiality levels, up to the extreme of demanding the removal of secret data from the system. In this thesis, we also propose a certification technique for confidentiality of complete Java classes that includes non–interference and erasure policies.

Resumen

En esta tesis se propone una metodología para la certificación de programas Java que está basada en la lógica de reescritura, un marco formal lógico y semántico muy general, implementado eficientemente en el lenguaje de programación funcional Maude. Se consideran propiedades de seguridad (safety), es decir, propiedades de un sistema que son definidas en términos de que no ocurran ciertos eventos. Dichas propiedades se caracterizan como problemas de inalcanzabilidad en la lógica de reescritura. Las propiedades de seguridad (safety) se expresan en el estilo de JML, un lenguaje estándar de especificación de módulos Java. Con el fin de obtener un procedimiento de decisión utilizamos modelos con un número finito de estados que obtenemos mediante el uso de la interpretación abstracta. Partiendo de una especificación de la semántica de Java escrita en Maude, se desarrolla una semántica operacional abstracta con un número finito de estados, también escrita en Maude, que resulta apropiada para la verificación de programas. Como subproducto de la verificación se entrega un certificado de seguridad, que consiste en un conjunto de demostraciones basadas en reescritura, que pueden ser comprobadas fácilmente por el consumidor del código mediante el uso de un motor de reescritura estándar. La técnica de *código portador de demostración* (proof-carrying code) basada en abstracción, denominada JavaPCC, ha sido implementada y probada con éxito en varios ejemplos, lo cual demuestra la viabilidad de nuestro enfoque.

En esta tesis se analizan propiedades locales de métodos Java, es decir, propiedades de sus parámetros y resultados. También se analizan propiedades globales de clases Java completas, primero estudiando la no interferencia, y luego, el borrado, con y sin no interferencia. La no interferencia es una propiedad semántica de los programas, que asigna niveles de confidencialidad a los datos y evita que haya flujos ilícitos de información desde los niveles de seguridad altos hacia los niveles bajos. En esta tesis presentamos un modelo novedoso de seguridad para la no interferencia global que aproxima la no interferencia como una propiedad de seguridad (safety). El borrado (de información confidencial) es una forma de reforzar la confidencialidad mediante el aumento de los niveles de confidencialidad de los datos, que puede llegar incluso a requerir la eliminación de los datos secretos del sistema. En

esta tesis, presentamos también un modelo de seguridad para programas Java completos que incluye políticas de no interferencia y borrado.

Resum

En esta tesi es proposa una metodologia per a la certificació de programes Java que està basada en la lògica de reescriptura, un marc formal lògic i semàntic molt general, implementat eficientment en el llenguatge de programació funcional Maude. Es consideren propietats de seguretat (safety), ès a dir, propietats d'un sistema que són definides en termes de certs esdeveniments que no han d'ocórrer. Eixes propietats es caracteritzen com a problemes d'inassolibilitat en la lògica de reescriptura. Les propietats de seguretat (safety) s'expressen en l'estil de JML, un llenguatge estàndard d'especificació de mòduls Java. A fi d'obtindre un procediment de decisió utilitzem models amb un número finit d'estats que obtenim per mitjà de l'ús de la interpretació abstracta. Partint d'una especificació de la semàntica de Java escrita en Maude, es desenvolupa una semàntica operacional abstracta amb un número finit d'estats, també escrita en Maude, que resulta apropiada per a la verificació de programes. Com a subproducte de la verificació s'entrega un certificat de seguretat, que consistix d'un conjunt de demostracions basades en reescriptura, que poden ser comprovades fàcilment pel consumidor del codi per mitjà de l'ús d'un motor de reescriptura estàndard. La tècnica de codi portador de demostració (proof-carrying code) basada en abstracció, denominada JavaPCC, ha sigut implementada i provada amb èxit en diversos exemples, la qual cosa demostra la viabilitat del nostre enfocament.

S'analitzen propietats locals de mètodes Java, és a dir, propietats dels seus paràmetres i resultats. També s'analitzen propietats globals de classes Java completes, primer estudiant la no interferència i després, l'esborrament, amb i sense no interferència. La no interferència és una propietat semàntica dels programes, que assigna nivells de confidencialitat a les dades i evita que hi haja fluxos il.lícits d'informació des dels nivells de seguretat alts cap als nivells baixos. En esta tesi presentem un model nou de seguretat per a la no interferència global que aproxima la no interferència com una propietat d'innocuïtat (safety). L'esborrament (d'informació confidencial) ès una forma de reforçar la confidencialitat per mitjà de l'augment en els nivells de confidencialitat de les dades, que pot arribar inclús a requerir l'eliminació de les dades secretes del sistema. En esta tesi, presentem també un model de seguretat per a programes Java complets que inclou polítiques de no interferència i

esborrament.

Contents

List of Figures

Chapter 1

Introduction

The purpose of this thesis is to automatically verify and certify Java programs based on the analysis of the corresponding source code. If the verification analysis succeeds, the corresponding certificate means that the Java code is safe at the source level, regarding some given properties that are detailed below.

Since the 1990's, there has been an increasing interest on formal methods devised to verify the safety of code retrieved from untrusted sources. Proof-Carrying Code (PCC) is a mechanism developed by G. Necula [Necula and Lee, 1996] that aims to guarantee mobile code safety to code consumers. In PCC, the code is distributed together with a safety certificate whose validity entails the compliance with a safety policy predefined and supplied by the code consumer. The safety certificate is automatically generated by the code producer, and then packaged along with the verified code. The certificate encodes an easy-to-check formal proof of the code safety. Then, the code consumer receives and checks the certificate, i.e. the safety proof. If the certificate is valid, the consumer can run the code safely. PCC was developed to guarantee mobile code safety but it is useful for general software development.

The crucial issues for a practical realization of PCC are: (i) the expressiveness of the language used to specify the considered policies, (ii) the size of the transmitted certificate, and (iii) the performance of the validation process at the consumer side. The main technologies most commonly applied in PCC are type analysis [Necula, 1997; Appel and Felty, 2000; Wildmoser et al., 2004; Felty, 2005; Hamid, 2005], theorem proving [Necula and Schneck, 2002; Necula and Schneck, 2003a; Yu and Mok, 2004; Beringer et al., 2003; Hamid, 2005; Chander et al., 2005a; Barthe et al., 2007b; Wildmoser et al., 2004; Besson et al., 2006], and abstract interpretation [Xia and Hook, 2004; Albert et al., 2005b; Besson et al., 2006].

Rewriting logic [Meseguer, 1992] is a flexible and expressive *logical framework* in which a wide range of logics and models of computation can

be faithfully represented. It also provides an easy and inexpensive way to develop formal definitions of programming languages which are *directly executable* [Meseguer and Roşu, 2007] as interpreters in a rewriting logic language such as Maude [Clavel et al., 2007]. The verification of embedded and reactive systems in rewriting logic offers a good number of advantages, an important one being the maturity, generality and sophistication of the formal analysis tools available for it (see, e.g., [Clavel et al., 2007]).

In this work, an abstraction-based PCC technique was developed for the certification of Java source code which exploits the automation, expressiveness and genericity of rewriting logic. Given a safety property (i.e., a system property that is defined in terms of certain events that do not happen [Manna and Pnueli, 1995]), the unreachability of the system states denoting the events that should never occur allow one to infer the desired safety property. Unreachability analysis is performed using the standard Maude (breadth–first) search command, which explores the entire state space of the program from an initial system state. In the case where the unreachability test succeeds, the corresponding rewriting proof that demonstrate that those states cannot be reached are delivered as the expected outcome certificate. Figure 1.1 depicts the proposed PCC framework for Java source code.

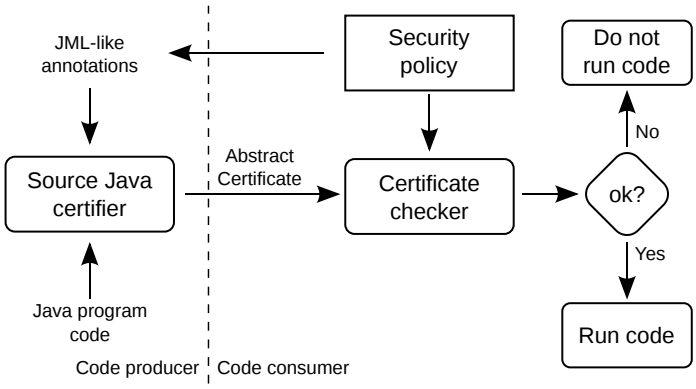

Figure 1.1: Overview of our JavaPCC framework.

Very often, the unreachability test does not succeed because there is an infinite search space; thus, abstraction is used in order to achieve a finite

search space [Cousot and Cousot, 1979]. Abstract interpretation is a program analysis framework where programs describe computations on abstract values instead of concrete ones [Cousot and Cousot, 1977]. Abstract values represent sets of concrete values that satisfy a given condition. In our abstract PCC methodology, certificates are encoded as (abstract) rewriting sequences that (together with an encoding of the abstraction in Maude) can be checked by standard reduction. The safety policy is expressed in JML-style [Leavens et al., 2006], a standard property specification language for Java modules. Program variables in the Java code are annotated following the safety policy supplied by the code consumer. Variable annotations correspond to variable properties, which mean abstract properties on the values of the annotated variables. In other words, annotations associate abstract domains corresponding to sets of concrete values to variables.

The policies considered in this thesis include safety policies based on integer arithmetic properties (e.g., parity, etc.), and two confidentiality policies (security policies) based on information flow analysis: i) Non–interference between high and low confidentiality information, and ii) erasure of confidential information. *Non–interference* is a semantic program property that assigns confidentiality levels to data objects and prevents illicit information flows to occur from high to low security levels [Denning and Denning, 1977; Sabelfeld and Myers, 2003]. *Erasure* is a way of strengthening confidentiality by upgrading data confidentiality levels, up to the extreme of demanding the removal of secret data from the system [Chong and Myers, 2005; Chong and Myers, 2008].

Non–interference and erasure are security properties that are usually defined as hyper-properties [Clarkson and Schneider, 2008]. A hyper-property is defined on a set of sets of traces, and cannot be established by simply checking a (safety) property on a set of runs (essentially, no single run of a system can violate non–interference with and without erasure). However, we are able to analyze non–interference (and erasure) by observing stronger corresponding properties which can be checked as safety[1] properties using an instrumented flow sensitive semantics.

Our methodology is an instance of the Proof-Carrying code (PCC) paradigm based on rewriting logic, and it is the first sound and implemented PCC frame-

[1]There are other approaches for proving non–interference as a safety property, which use self-composition [Darvas et al., 2005; Barthe et al., 2004], or flow sensitive security types [Hunt and Sands, 2006]; see Sections 6.3.1 and 7.3.1.

work for confidentiality certification of Java source programs. However, we have to emphasize that the delivered proof certificates the source code program, nor the corresponding executable code, while most PCC frameworks certificate only executable code (see Section 1.1 below).

Let us summarize the general contributions of this Ph.D. thesis as follows:

- We formalize a correct and automatic reachability verification methodology for Java source code that works for safety properties. This methodology is based on a specification of the semantics of Java source written in Maude and can be extended to handle other programming languages, whose semantics is specified in rewriting logic.
- An abstract version of the Java semantics written in Maude is developed for finite-state verification.
- Abstraction of the Java semantics is produced by increasing the nondeterminism of the semantics using rules, i.e. equations in the semantics representing deterministic computations are transformed into rules representing non–deterministic computations in order to reflect the more abstract computations.
- As a by–product of the verification, a certificate of the safety policy fulfillment is delivered which consists of a set of rewriting proofs that can be easily checked by the code consumer by using a standard rewriting logic engine.
- Starting from the specification of the semantics of Java written in Maude [Farzan et al., 2007], we also have developed an information–flow extension of such an operational Java semantics which allows one to observe non–interference and erasure of Java programs, and is also written in rewriting logic.
- We have provided a novel characterization of non–interference and erasure, which are security properties, as safety properties on extended Java computations.
- A finite-state version of the information-flow Java operational semantics is also provided by using abstract interpretation. Thanks to the different handling of rules and equations in Maude it does not suffer from the state–space explosion of more traditional approaches.

- Finally, we provide an implementation of the abstract certification methodology and the experimental evaluation of the proposed technology in order to demonstrate the feasibility of our approach.

The manuscript is organized as follows. Chapter 2 summarizes some preliminary notions about rewriting logic, Maude, and abstract interpretation. Chapter 3 briefly describes the Java rewriting logic semantics that is the basis for our work. Chapter 4 presents the basic certification methodology, i.e. the verification that a given safety policy (specified in a notation that follows the Java Modeling Language JML style) is satisfied by a Java source program, and the generation of the corresponding certificate. The certification of some integer arithmetic properties by Java function methods that return integer values is introduced in Chapter 5. Chapter 6 presents the case of non–interference policies regarding sequential complete Java classes. In Chapter 7 the methodology of Chapter 6 for the certification of non–interference is extended in order to certify erasure policies as well. Chapter 8 introduces the implementation, i.e. the JavaPCC certification environment. Finally, Chapter 9 concludes.

Let us finish the introduction with some bibliographic remarks about the related work on semantic-based program certification and the publications associated to this thesis.

1.1 Semantic-based program certification

There are different approaches for software verification and certification of security and safety properties where the delivered safety certificates are checkable by code consumers. In this section we recall the most important proposals from the related literature. First, the Proof-Carrying Code (PCC) approach [Necula and Lee, 1996; Necula and Lee, 1997b; Necula and Lee, 1997a; Necula, 1997; Necula and Lee, 1998b] to code safety certification is briefly described, together with some variants that aim to address the main technical drawbacks of PCC. Then, the model-carrying code (MCC) approach [Sekar et al., 2001; Sekar et al., 2003] is introduced, where the code producer delivers a high level model of the code behavior as the safety certificate. Next, we summarize several approaches that introduce certified components within the trusted code base (TCB). Then, we give an overview of an hybrid approach that combines PCC with certifier authorities. This approach is able

to cope with some safety properties outside the PCC scope. Finally, we discuss a novel approach to code synthesis that combines code verification and certification and is known as Code-Carrying Theory (CCT) [Vargun, 2006].

1.1.1 Proof-Carrying Code (PCC)

As already mentioned, the mobile code is distributed in PCC together with a safety certificate whose validity entails the compliance with a safety policy predefined and supplied by the code consumer. This safety certificate is generated by the code producer, and encodes an easy-to-check formal proof of the code safety. The code consumer receives and checks the certificate, i.e. the safety proof, and whenever it is valid, the code consumer can run the code safely. The code consumer uses a code infrastructure that allows him to check the certificate automatically. This certificate validation infrastructure is known as the trusted code base (TCB), because the code consumer assumes that it is trustworthy.

The main technical issues regarding the practical realization of PCC are:

1. The *expressiveness* of the safety policy specification language regarding more general and complex safety and security properties [Appel and Felty, 2000; Appel, 2001; Necula, 2001a; Necula and Schneck, 2002].

2. The *size of the certificates*, that could grow exponentially as a function of the code size [Necula and Lee, 1997a; Necula and Rahul, 2001; Wu et al., 2003]. This also mean exponential generation, transmission and validation times.

3. The *reliability* of the trusted base code (TCB), that is inversely proportional to its size [2] [Appel and Felty, 2000; Appel, 2001; Necula and Schneck, 2002].

4. The *scalability* of the technique regarding program size and the generality and flexibility of the approach to consider different low and high level programming languages [Appel and Felty, 2000; Necula and Schneck, 2002].

[2] A typical Verification Condition Generator (VCGen) implementation has more than 26,000 lines of code. A bug in this code may cause the proving and checking of a wrong property, or the failure to prove a property that is true.

In pioneering work, Necula and Lee [Necula and Lee, 1996; Necula and Lee, 1997b] consider memory safety properties of operating system extensions written in assembly language. The work of Necula [Necula, 1997] deals with memory and type safety of assembly language extensions for an ML compiler. Finally, in [Necula and Lee, 1998b] the authors consider memory safety, authorized access to files, and resource usage constraints (e.g. memory, locks, bandwidth, and CPU cycles) of assembly code. The traditional PCC infrastructure typically consists of three components (see Figure 1.2, borrowed from [Necula and Lee, 1998b]): a verification condition generator (VCGen), the prover which generates the certificate, and the certificate checker.

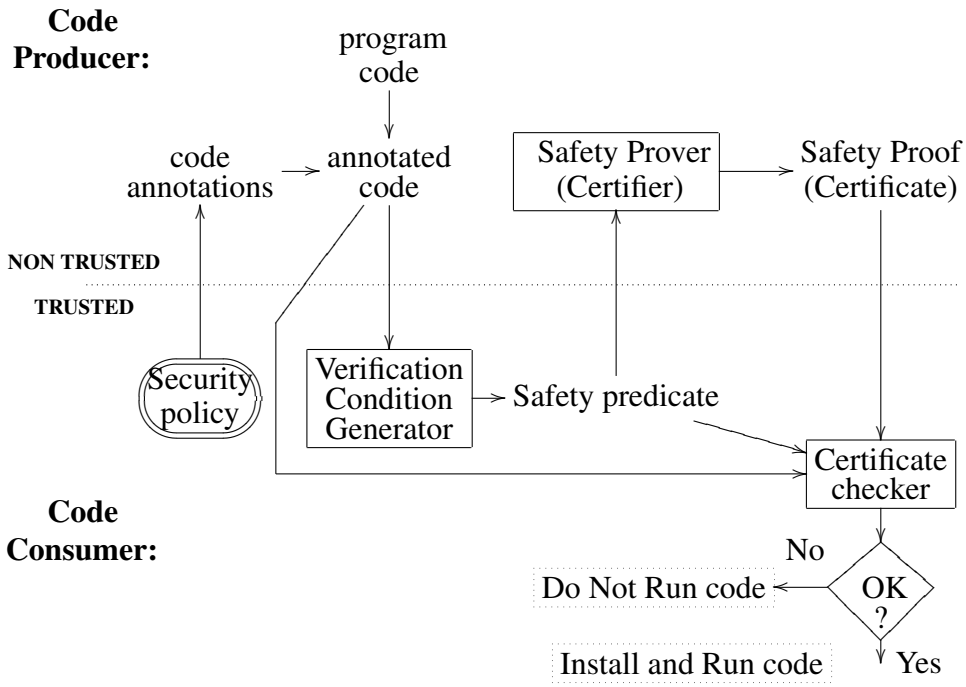

Figure 1.2: Overview of the typical PCC framework.

The VCGen component uses code annotations to generate a security predicate that encodes the supplied policy together with some needed lemmas. The code producer uses the supplied policy, the VCGen, and the prover, whereas

the code consumer uses the VCGen and the proof checker. Another important assumption of the PCC infrastructure is that the code consumer trusts the VCGen and the proof checker, i.e. he assumes that they are both trustworthy. This means that the TCB of traditional PCC includes the verification condition generator and the certificate checker.

All the above pioneering works are based on type checking and use the logical framework LF[3] in order to encode the proofs (i.e. the type derivations are LF expressions) and type checking to validate the proofs.

Necula and Lee rely on the redundancies of the LF expression that encodes the certificate to reduce both the certificate size (fifteen times) and the validation time (seven times). The new representation was called LFi [Necula and Lee, 1997a].

Certifying Compilers

The first certifying compiler in the literature was proposed in [Tarditi et al., 1996] that, together with the generated code, produces a proof or formal certificate that can be mechanically checked [Tarditi et al., 2004]. The TIL compiler uses a type system to verify and generate the proof that the ML code satisfies, namely a critical invariant that is verified and enforced by a type system. This compiler considers type safety and array safety regarding index bounds. The FLINT/ML certifying compiler [Shao and Appel, 1995; Shao, 1997] considers safety of types and arrays of complete ML programs with modules. Array safety involves execution time analysis regarding array bounds.

Necula and Lee [Necula and Lee, 1998a; Necula and Lee, 2004] introduced the first certifying compiler that automatically generates the full proof (encoded in LF) that a program written in a type-safe subset of C satisfies a given safety policy for memory and types. Necula et al. introduced the certifying compiler for a subset of Java (SpecialJ) [Colby et al., 2000; Necula, 2001b] obeying memory and type safety properties, that also uses LF in order to encode the generated proofs. The VCGen component of the SpecialJ PCC framework has approximately 23K lines of code, the SpecialJ compiler (the Touchstone Java compiler) has 33K lines of code, while the proof checker has about 1.4K lines of code, and the proof rules and axioms amount for 700 lines

[3] It is an extension of lambda calculus with simple and dependent types implemented in ELF.

of code [Colby et al., 2000]. The TCB size of this PCC framework has about 25.1K lines of code. All known certifying compilers use type systems and static analysis in order to verify that source programs fulfill the considered safety policy. Many certifying compilers are based on the typed assembly language TAL [Morrisett et al., 1999b]. These compilers apply to low level languages, such as TAL and TALx86 [Morrisett et al., 1999a; Grossman and Morrisett, 2001], and some high level languages, namely two dialects of C (Popcorn and Cyclone [Jim et al., 2002]), and Scheme. The high level compilers generate TAL code, together with a certificate that consists of suitable type annotations for the code. The certificates are validated by type checking based on the generated annotations. The project TILT was launched in order to develop certifying compilers for the ML family of languages (SML '97, Caml, KML) regarding safe type handling [TILT, 2006].

Yiyun et al. [Yiyun et al., 2007; Chen et al., 2007] proposed a certifying compiler for the PointerC language, a C-like language with restricted pointer and type operations that allow one to check types and pointers statically. There is no type casting operation, and the reference (&) and pointer-arithmetic operations of C are not considered. PointerC has no union types. PointerC has safe explicit memory allocation and deallocation operations. The Touchstone compiler does not have dynamic memory deallocation. The typing rules introduce side effects in order to specify constraints on values. PointerC functions are annotated with pre and post-conditions, and loops with invariants. The compiler front-end includes the verification condition generator and a built-in automatic theorem prover that discharges the proof obligations regarding pointer-related verification conditions. The integer-related verification conditions are proved interactively by using the Coq theorem prover. The compiler back-end generates assembly code and transforms the VCs and the corresponding proofs produced by the front-end into code annotations (assertions), by applying Hoare logic to the operational semantics of the x86 assembly language. PointerC has more types and pointer operations than the Touchstone C certifying compiler.

Li et al. [Li et al., 2010] introduced a certifying compiler for a subset of the ANSI C language, that has a built-in automatic theorem prover. The certifying compiler, named Clike, deals with programs that handle single and double-linked lists and binary trees with explicit memory allocation and deallocation. The safety properties include memory safety, and C type safety, including heap safety. The compiler was implemented in SML/NJ. Their proof

libraries were implemented by using the meta-logic provided by the Coq theorem prover, i.e. the calculus of inductive constructions CiC. At the source level, a constrained first-order logic and a fragment of separation logic are considered. The front-end includes a verification condition generator and the built-in theorem prover. The verification conditions are generated from the input Clike source code that is annotated with pre, post-conditions and invariants. These verification conditions are discharged by the theorem prover. The front-end produces the abstract syntax tree of the program, the verification conditions and their corresponding machine-checkable proof (a Coq term). The back-end generates x86 assembly code, low-level specifications and the corresponding low-level proof. The assembly code, the low-level specifications and the proofs use a variant of the stack-based certifying assembly programming language. The proofs produced by Clike can be checked by Coq. The size of the Clike compiler is 25.500 lines of code. It is smaller than the Touchstone C compiler [Necula and Lee, 2004], and it can certify more complex safety properties.

In the following, we recall some variants of PCC that aim to overcome the main technical issues of PCC.

Foundational Proof-Carrying Code (FPCC)

Appel et al. [Appel and Felty, 2000; Appel, 2001; Appel and Mcallester, 2001; Felty, 2005] proposed a foundational approach to PCC (FPCC), that uses a minimal logic and a small set of axioms. Also FPCC uses a general type system in order to cope with different low level languages and safety properties.

This approach assumes that the *typing rules are not trusted*, and integrates the programming language semantics component with the policy component, in order to reduce the size of the TCB and to increase its reliability. FPCC reduces effectively the size of the TCB, from 26,000 to 2,668 lines of C and LF code, but not the size of the certificates, neither the validation time [Appel, 2001; Hamid et al., 2003; Hamid et al., 2002; Necula and Schneck, 2003b].

Some initial FPCC proposals [Appel and Felty, 2000; Appel, 2001; Appel and Mcallester, 2001] use TWELF (another LF implementation), while more recent FPCC approaches [Felty, 2005] use the theorem prover Coq[4]

[4]Coq is based on the calculus of inductive constructions, an extension of the lambda calculus with types.

in order to generate and check the certificate by using type inference and type checking, respectively. In [Appel, 2001], Appel certified memory safety properties, whereas in [Appel and Felten, 2001] more general properties can be expressed (although not verified neither certified), including fairness regarding process scheduling, memory allocation and deallocation, safe API invocation sequences, lock number bounds, output logging, and package retransmission without modification. Another FPCC proposal that introduced semantic analysis in order to simplify and reduce proof size and to consider recursive types is [Appel and Mcallester, 2001].

Necula and Schneck [Necula and Schneck, 2002; Necula and Schneck, 2003a] considered also *non-trusted type inference rules*, for which a soundness proof is provided. This work uses the theorem prover Coq to demonstrate the soundness of the proof rules for a type safety policy for assembly code that is generated by a Java compiler. The validation of the Coq proof requires only type-checking. In this approach, the TCB simply consists of the type-checker of Coq, a small part of the prover.

FPCC with witnesses Appel et al. [Appel et al., 2003; Wu et al., 2003] used *proof witnesses* instead of full proofs in order to reduce the certificate sizes, and non-trusted inference rules in order to reduce the TCB size. This proposal considers type safety, and represents the typing rules as Prolog-like clauses, which are run without backtracking. The proof witness is a logic program trace that contains the successful goals and sub-goals. The proof witnesses are smaller in size than full proof certificates. But the witnesses size can be one thousand times the program size, which is impractical. The framework uses a deterministic logic interpreter (FLIT) in order to check the proof witnesses. The system FLIT restricts clause and goal syntax in order to achieve a small and efficient proof checker. FLIT was implemented in 282 lines of C code [Wu et al., 2003]. This framework was applied to certify type safety regarding arithmetic integer properties (such as the parity of an integer expression) of machine language code. The machine code is generated by a Core ML compiler (based on SML/PJ) that also generates LTAL (Low-level Typed Assembly Language) expressions that encode the proof witness expressed in LF. In this case, the total TCB size amounts to 3,034 lines of code [Wu et al., 2003].

In [Wu, 2005], Wu developed the *first end-to-end* FPCC system, that integrates the Core ML type-preserving compiler, the LTAL language, and the

FLIT proof checker altogether. The compiler generates SPARC machine code together with the LTAL code. This proposal includes a machine checkable soundness proof that was checked using TWELF and FLIT. Wu's experiments include some small "but not trivial" LTAL benchmarks. In this case, the full proof size is 143,4 Kb, including the safety specification, logic, arithmetic, algebra, sets, relations, functions, lists, vector, trees, machine specifications, and LTAL specifications. Regarding the experiments with the even/odd property, the validation time grows slower than the code size by a factor of 1/2; i. e., if the number of expressions of the code grow by a factor of 10, then the corresponding validation time grows by a factor of 5. In the case of the LTAL benchmarks, the validation time also grows slower than the code size. However, the validation time growing rates tend to be similar as the code size increases: if the code size changes from 32 to 870 lines of machine code (i. e., it grows by a factor of 27), the validation time grows by a factor of 3; if the code size changes from 870 to 1816 (a factor of 2) lines of code, the validation time grows less than 2 times [Wu, 2005].

Syntactical FPCC: Hamid et al. [Hamid et al., 2003; Hamid et al., 2002; Hamid and Shao, 2004; Hamid, 2005] introduced a *syntactical* approach to foundational proof-carrying code (SFPCC) in order to reduce the complexity and size of the proofs generated by FPCC. The safety proof is based on syntactical type inference. The certificate includes the safety proof of the used type system, that is not assumed to be trusted. This framework was used to certify code written in the typed assembly language FTAL (Featherweight Typed Assembly Language), a version of TAL. FTAL supports integer arithmetic, records, recursive types, and pointers as first-class values. FTAL can be used as the object language of Java, ML, and a typed restricted of C. The safety policies include memory allocation safety and safe execution with no illegal instructions. The authors use the Coq prover in order to generate the proof, whereas the Coq type-checker is used to validate the proof. SFPCC can also certify code written in XTAL, a language that can be used in combination with other low and high level languages that could use different type systems.

Open Verifier, a non-standard FPCC approach: Chang et al. [Chang et al., 2005; Schneck, 2004] introduced a framework called the Open Verifier for developing foundational verifiers for FPCC. Non-trusted code is verified by using customized verifiers. The code producer does not supply a safety

proof. Instead, he delivers an executable extension of the verifier, which is specific for a source language, together with some program meta-data. In this way, the Open Verifier framework avoids common FPCC drawbacks regarding proof encoding and proof size [Necula and Rahul, 2001]. The customized verifier generates weak program post-conditions, while the Open Verifier generates stronger post-conditions by using executable code instead of logic axioms in order to describe machine transitions. The performed verifications are not foundational in the sense that they are not fully based on a given compact logic.

The code consumer generates the safety proof by using the Open Verifier and the extension of the verifier that corresponds to the selected programming language. The Open Verifier is considered trusted while the customized extension is not. The framework includes a proof checker and a fix point approximation interpreter. The Open Verifier checker and the interpreter have an implementation written in Ocaml with some extensions that support different verification strategies. There is an extension of traditional PCC (PCCExt), another for the TALx86 language (TALExt) [Morrisett et al., 1999a] and other for Cool, a type-safe object oriented language similar to a subset of Java, called CoolAid. The safety policies considered in [Chang et al., 2005; Schneck, 2004] include memory and type safety. The extensions are the most complex components of the Open Verifier: the PCCExt extension has 2400 lines of code, the TALExt extension has 9300 lines of code and the CoolAid has 6900 lines of code. The TCB size of the Open Verifier has 4.250 lines of code [Schneck, 2004].

Other remarkable PCC variants

Configurable PCC: Necula and Schneck [Necula and Schneck, 2003b] introduced a configurable proof-carrying code approach (CPCC) that is more trustworthy than PCC because the trusted code is only a small part of the verifier (core VCGen). The code producer supplies not only the code but also most part of the code for the verifier (custom VCGen). The trusted verifier (core VCGen) checks some properties of the supplied untrusted custom verifier. The custom untrusted verifier is executable code. Instead of the language semantics, the trusted code includes a decoder, which returns a symbolic predicate that must be satisfied in the current state in order to ensure a safe execution. The core VCGen handles general safety policies (memory safety, safe

arithmetic, and no side effects), while the custom VCGen may handle special safety policies. In this way, CPCC is also more flexible that standard FPCC approaches. CPCC can be applied to low level languages (assembly and virtual machine languages). To the best of our knowledge, there is no known implementation yet.

Prototype PCC: Wildmoser et al. [Wildmoser et al., 2004] introduced Prototype Proof Carrying Code (ProtoPCC) as a complement of the mentioned CPCC proposal of Necula and Schneck [Necula and Schneck, 2002]. The framework is instantiated for a simple assembly language (SAL) and a safety policy that includes safe types and lack of overflows in arithmetic operations. The safety policy is expressed as program annotations. The code producer can enter his SAL annotated programs into the inductive theorem prover Isabelle. The verification condition can be generated by a supplied simplifier. The code producer can demonstrate the verification condition by using Isabelle, the simplifier, and a Presburger arithmetic procedure. The proof is a lambda term whose type corresponds to the proved theorem. Proof validation reduces to type-checking the term that encodes the proof. The code consumer uses the Isabelle type-checker in order to validate the proofs. Wildmoser et al. [Wildmoser et al., 2005] reported an implementation of this framework to certify that bytecode programs have no overflows because of arithmetic operations, and that vector indexes are not out of bounds. The bytecode is generated by the Jinja compiler, that considers only a subset of the bytecode language (JVM language) with no objects and no method invocations. In previous work the program annotations were written by hand, while they are automatically generated in [Wildmoser et al., 2005].

Oracle PCC: Necula and Rahul [Necula, 2001a; Necula and Rahul, 2001] proposed the use of a higher-order logic program for representing the trusted type inference rules, and the use of a non-deterministic higher-order logic interpreter as a proof checker, and they introduce *oracles* instead of proofs as certificates. The oracles are bit streams that resolve the non-deterministic interpretation choices. The logic interpreter is similar to λ-Prolog. This work considers type safety of assembly code generated by a Java compiler. The oracles, i.e. the certificates, are 12% the size of the code, and 26-35% the size of the corresponding LFi proofs (c.f. Section 1.1.1 of [Necula and Lee, 1997a]). However, the validation times are three times higher than type

checking LFi expressions [Necula and Rahul, 2001], and the TCB size is about 26.000 lines of code.

Interactive Proof Carrying Code IPCC: This is a proposal that aims to address the certificate size problem. It is known that the size of the proof that a given code satisfies a safety property, increases as size and code complexity increase such that it reduces PCC application to cases where the proof checking time would be reasonable from the consumer point of view. Tsukada et al. [Tsukada, 2000; Tsukada, 2005] proposed an interactive and probabilistic extension to PCC, called Interactive PCC, that reduces the checking time for a property of a given code with m lines of code and n conditionals, from $2^k * poly(m)$ with PCC to $poly(m)$ with iPCC. The probability that a iPCC proof is correct is $\leq 1/2^m$.

The safety property is specified in negative form with boolean formulae that are existentially quantified. The program is safe if and only if the formula is false. The boolean formulae, disjunctions of conjunctions, are expressed in arithmetic form as summation of multiplications (variables have 0 or 1 values, disjunction is transformed into addition, conjunction into multiplication, and negation into complement). This way, the boolean formula is false iff the corresponding arithmetic formula is evaluated to the 0 value. The arithmetic formulae are grade one polynomials whose coefficients are interactively delivered by the code producer to the code consumer, one by one. These coefficients are the proof that the code satisfies the given property. To the best of our knowledge, this proposal has not been implemented yet.

Extended Proof Carrying Code EPCC: Pirzadeh and Dubé [Pirzadeh and Dubé, 2008b; Pirzadeh and Dubé, 2008a] introduced the Extended Proof Carrying Code EPCC approach in order to tackle the proof size issue of PCC for assembly-like languages. Similarly to the non-standard PCC approach of the Open Verifier, the code producer does not supply a safety proof by using EPCC, but a program. The program delivered by using EPCC is a proof generator. In order to get the proof in a secure way, the code consumer runs the proof generator by using a specific stack-based virtual machine that is small and safe [Pirzadeh and Dubé, 2008b]. The EPCC virtual machine VEP is part of the TCB and has less than 300 lines of code. VEP enforces memory safety (i.e. valid memory addresses), control-flow safety (no jumps outside the address space), type safety (including numeric ranges) and resource

bounds safety (code size, stack size, heap size and timeout).

The authors implemented an EPCC prototype that includes an assembler for the VEP machine, a C compiler that produces the assembly code for the VEP assembler and a proof generator written in C [Pirzadeh and Dubé, 2008b]. The input of the proof generator is the proof produced by a theorem prover, given the verification conditions generated by a suitable VCGen. The transmitted proof generator is compressed by using an off-the-shelf compressor. The code consumer uses GUNzip, a decompression-only program for the VEP machine that is transmitted together with the proof generator. In the EPCC approach the TCB is composed by the VEP machine, the proof checker and the VCGen [Pirzadeh and Dubé, 2008a].

Resource consumption in high level programming languages

Resource consumption by programs has been recently considered a safety and security issue because excessive resource consumption may compromise the availability of the services offered by the hosting computer system. Most denial of service (DoS) attacks to Internet servers are based on the exhaustion of the limited resources of the attacked systems (processing time, bandwidth, main memory, etc.). In the following, we summarize the approaches to resource consumption certification in three high level programming languages C, Camelot and Java.

The Tinman framework for C: Mok and Yu [Mok and Yu, 2002a; Mok and Yu, 2002b] proposed a framework to guarantee *safe resources handling* by C programs. Tinman framework combines static analysis with run-time monitoring. The certificate is a forecast of run-time resource consumption that includes memory allocation. The security policy has three parts: i) the resource consumption of each invoked service of the host system (pre and post conditions), ii) the resource consumption of the code, and iii) the proof system that interprets the policy specification. Time consumption analysis considers the worst case, but there are programs that require programmers to annotate the maximum execution time. Tinman uses the PVS (Prototype Verification System) prover system in order to generate the proof. PVS is a top-down, goal oriented prover, based on a higher-order logic with types. In order to reduce the size of the PVS proofs, the certificate only includes the tactics used by PVS with their parameters, i.e. a proof skeleton. Tinman framework

formalization and soundness are considered in [Yu and Mok, 2004].

The functional programming language Camelot: The MRG (Mobile Resources Guarantees) european project [Sannella et al., 2005; Gilmore and Prowse, 2005] developed two programming languages devised to certify the resource consumption of mobile code written by using those languages. The first-order functional programming language Camelot was designed in order to certify linear heap memory consumption of Camelot functions. Camelot is similar to ML but has classes and objects. The certificate proves that the memory consumption is linear regarding the size of the input.

The Camelot compiler generates Grail code (Guaranteed Resources Allocation Intermediate Language), a functional virtual machine bytecode, together with the corresponding resource consumption proof, i.e. the certificate [Beringer et al., 2003]. The resource consumption certificate is a term of an LDF type [Hofmann and Jost, 2003] encoded as an Isabelle term. The certificate includes an encoding of the abstract syntax of Grail. The Grail compiler can generate bytecode for the JVM Java Virtual Machine, and also CIL code for the .Net framework [Beringer et al., 2003; MacKenzie and Wolverson, 2004].

The code producer uses the Camelot and Grail compilers to verify, compile and certify a Camelot function. The code producer transmits the JVM code, together with the certificate (the LDF term and the Grail syntax). The code consumer uses a Grail decompiler in order to generate the Grail code from the JVM bytecode, and the Isabelle prover to type-check the certificate and the Grail code. The theorem prover Isabelle big size in kbytes implies that the code consumer cannot use it in mobile devices, nor desktops computers [Aspinall et al., 2004].

JVer, a Java verifier: The work of Chander et al. [Chander et al., 2005a; Chander et al., 2007] considered physical resource consumption bounds (CPU time, memory and disk space, and bandwidth), and virtual resources (files, database connections, and program threads). The resource consumption policy is expressed as code annotations that mean run-time code behavior regarding resource consumption. The annotations determine the allocation of resources to the program, before its actual execution and resource consumption. These “dynamic” annotations are verified statically.

The framework has general application. It is composed of a VCGen, and a prover based on logical satisfiability. The VCGen generates predicates that are conjunctions of relations between integer variables and constants, i.e. linear inequalities. These simple predicates can be more efficiently verified than first-order formulae.

There is one implementation of the framework that applies to the certification of source Java code with non-standard JML annotations [Chander et al., 2005a]. This implementation uses the ESC/Java tool as VCGen and the theorem prover Simplify [Detlefs et al., 2003] as prover. The Symplify prover implements a satisfiability procedure to solve linear inequalities. However, the prover Simplify cannot generate independently-checkable proofs. The ESC/Java based implementation was used to verify a Java version of the `tar` program, with 1700 lines of code that includes 577 relevant I/O lines of code. There is another implementation that applies to annotated Java bytecode [Chander et al., 2005b]. This implementation uses the JVer framework [Chander et al., 2005a] as VCGen and the theorem prover Kettle as an independent-checkable proof generator. JVer was used in experiments with game code for mobile phones.

1.1.2 Model Carrying Code (MCC)

Sekar et al. [Sekar et al., 2001] introduced "Model-carrying code" MCC as a new approach to software security certification, where the certificate is not a proof in any computational logic. Instead, it is a high level model of the program behavior. In this proposal, the model is the finite state automata whose transitions correspond to system events that occur during program execution, like file writing and reading and network communication.

As usual, the MCC code consumer knows the code and its model, and can check if the code has conformance with the model, prior to program execution. Moreover, the MCC code consumer can monitor and enforce the observed policy during code execution, and also she can refine the policy, i.e. the model. In [Sekar et al., 2003] the general policy corresponds to an application classification that distinguishes between *file only* and *communications only* applications. *File only* applications cannot communicate through the network, but can read and write files. *Communications only* applications cannot access any file but can communicate through the network. The general policy *file only* can be refined in order to allow network accesses before any sensible

file reading. Policies can be also refined in order to consider the sites where files are located.

MCC cannot be applied to dynamically loaded code, or code that does not use system calls for file handling and network communication.

Abstraction-Carrying Code (ACC)

In order to verify and certify liveness temporal properties of C code, Xia et al. [Xia and Hook, 2003a; Xia and Hook, 2003c; Xia and Hook, 2003b; Xia and Hook, 2004] introduced an MCC variant based on abstract interpretation called "Abstraction-carrying code" ACC. In ACC, the certificate is an abstract interpretation of the program. More formally, it is a boolean program abstracted from the LTL predicate (without "next" operators) that expresses the liveness condition. The C compiler generates the abstraction of the program together with the corresponding assembly code. The boolean program is encoded as a set of type annotations that are written in an assembly language with dependent types called SDTAL ("Simplified Dependent Typed Intermediate Language"). The code consumer checks the type of the certificate, and then executes an abstract model checker in order to validate the temporal property. Typical ACC certificate sizes are smaller than the actual code size, but ACC TCB size includes the model checker and the SDTAL typechecker. The implementation of ACC, called ACCEPT/C, uses the BLAST tool [Henzinger et al., 2002] in order to generate the control-flow graph of the C program.

Abstract Carrying Code (ACC)

Albert et al. [Albert et al., 2005a] proposed another abstraction-based approach to MCC, which is based on Constraint Logic Programming CLP. The framework is applied to analyze and certify CLP Ciao programs. It is also implemented in Ciao, and uses the assertion language of the Ciao pre-processor with pre- and post-conditions. In this case, the certificate is the result of a fix point abstract analysis of the Ciao program.

The security policies include file access control based on regular file naming, i.e. file names are regular expressions [Albert et al., 2005a], determinacy, termination, and non-failure, resource consumption bounds, time and space cost, and absence of side-effects [Albert et al., 2004; Albert et al., 2005c].

In [Albert et al., 2011], Albert et al. also introduced a reduced size certificate with a 60% size reduction. Validation time increases, but only a 6%.

This approach has some limitations reported in [Albert et al., 2004] which are due to the approximation by abstract interpretation, and also because of the undecidability of some properties.

Relational abstraction

As part of the Lande project, Besson et al. [Besson et al., 2005; Besson et al., 2006; Besson et al., 2007] introduced a further FPCC approach: the certified abstract interpretation approach to the verification of relational linear properties among integer program variables. The proposal considers a subset of a stack-based language like JVM, i.e. imperative bytecode, with procedures, arrays, and global variables. This framework can be applied to certify properties based on value ranges of integer variables. This way, it can be used to verify resource consumption bounds and array bounds, another kind of memory safety property. Safe array index handling means that index variables are within the corresponding array index bounds, as in the ProtoPCC framework. The automatic abstract analysis of programs can infer *relational* invariants and procedure pre- and post-conditions by an abstract interpretation of linear relations between program variables and symbolic execution.

The result of the abstract certification is the certificate, which contains the program annotations and the fix point abstract interpretation of the program, i.e. a system of linear inequalities over program variables. This system of linear constraints is a polyhedra inclusion. The approach instantiates the abstract domain by using polyhedra [Cousot and Halbwachs, 1978]. The system of linear constraints can be validated in quadratic time. The certificate sizes are one order of magnitude smaller than the program sizes.

The abstract interpretation analysis is specified in Coq. The code producer uses this VCGen and Coq to generate the certificate, that is a lambda term. The code consumer uses Coq and a simple type-checker to validate the lambda term. The proof checker is also verified and certified in Coq. In order to simplify and improve the efficiency of the certificate generation and validation, the program analyzer and the certificate checker were compiled to Caml.

1.1.3 Certified components for FPCC

In order to reduce the risk associated to the use of the TCB, an alternative is to introduce non-trusted components that replace part of the TCB, i.e. components that are not assumed to be trustworthy, as in CPCC. Below we summarize several approaches that rely on non-trusted components for FPCC which are also verified and certified.

A certified proof checker

The Mobius project work of Barthe et al. [Barthe et al., 2007b] introduced a verified proof checker for a bytecode-like language that could be used as a FPCC proof checker by code consumers. In other words, the proof checker is not a part of the TCB. This work considers non–interference policies as discussed in Chapter 6. The verified checker was extracted from the Coq soundness proof.

A certified program analyzer

Chang et al. [Chang et al., 2006; Chlipala, 2007] introduced an abstract interpretation based program analysis framework that can be used by a FPCC infrastructure regarding type and memory safety policies. The code producer does not supply a proof of the fulfillment of a given code consumer safety policy. Instead, he delivers the source code of the program analyzer that verifies the fulfillment of the given safety policy, together with the corresponding soundness proof, i.e. a verified and certified program analyzer developed by using the proposed framework. The soundness proof are produced by using abstract interpretation and the Coq theorem prover. The Coq specifications are extracted from the source code implementations of the program analyses written in ML. The code consumer extracts the specifications from the delivered source code, then she uses the specifications to validate the soundness proof using a proof-checker. This validation uses three trusted components: i) a compiler, ii) a specification extractor, and iii) a proof checker. If the proof validation succeeds, then the code consumer installs the program analyzer. The code consumer verifies non-trusted code using the compiled analyzer and the proof checker.

There is a prototypical implementation that allows one to analyze assembler code produced by the TALx86 compiler [Morrisett et al., 1999a]. The

prototype disregards only the modular features of the language. There is a partial implementation of a bytecode verifier, with no exceptions, object initialization, nor methods. This implementation allows one to certify ML code. The specification extractor supports a significant subset of the ML language. The TCB includes the Ocaml compiler and the Coq proof checker.

1.1.4 PCC with certifier authorities

Whitehead et al. [Whitehead et al., 2004] claim that there are some safety policies that cannot be verified and certified with a PCC framework without a certifier authority. They proposed a novel approach that complements a PCC system with an authorization system based on the Binder language that allows authorities to certify code by using digital signatures, when the safety policies are outside the scope of the PCC subsystem. The Binder language can express trust relationships between different agents.

1.1.5 Code synthesis and certification

Automated code generation has been considered an enabling technology by model-driven software development (MDD) for specific domains, even for safety-critical domains, where testing it is not enough and code generators could be flawed. We summarize the main achievements in this research trend, as follows.

Domain specific model driven code synthesis

Schuman et al. [Whalen et al., 2002; Schumann, 2003; Whalen et al., 2003] simultaneously generated code and all annotations required to verify and certify code, by using Hoare logic and theorem proving. Their approach is implemented as an extension of AutoBayes, a synthesis tool for automatically generating C and C++ programs for data analysis from equations on a specific domain. The extension generates Modula-2 code that includes formal proof annotations. The generated and annotated code is processed by the verification condition generator that is implemented with MOPS (Modula Proving System) in order to generate proof obligations that are then discharged by the first-order strategy-based theorem prover E-SETHEO. The framework was used to certify operator safety regarding partial functions, and memory safety

on a data-classification program. It was also used to certify variable initialization safety [Whalen et al., 2003]. There are cases with proofs that have to be obtained interactively with user input (4 of 69).

Denney and Fischer introduced a different approach to certifiable program generation [Denney and Fischer, 2006a] for safety-critical applications that uses Hoare logic, annotation inference [Denney and Fischer, 2005; Denney and Fischer, 2006b] and a first-order theorem prover (E-SETHEO, Vampire, and Spass were tried). The generated code is used for guidance, navigation, and control of spaceships. A range of safety properties are considered, including initialization safety, and no out-of-bounds array accesses. Initialization safety ensures that each variable or array element has been explicitly assigned a value before it is used. The approach has been implemented in the tool Autocert [Denney and Trac, 2008], a software safety certification tool for automatically generated C code from state flow Simulink models. The AutoCert tool formally verifies that the generated code is free of different safety violations, by constructing an independently checkable certificate. Code generation takes into account not only the code itself but also the annotations required for verification. The tool generates verification conditions that are discharged by the first-order theorem prover. The proof obligations generated (25000 in some experiments) required a lot of preprocessing before they can be discharged by the theorem prover. Some proofs have more than 8000 inference steps. The proofs are encoded as acyclic directed graphs that contain the formulae derivations. Proofs can be checked by semantic derivation verification. The certificate includes the proof, the proof explanations, and some links that allow code consumer not only to check the proof but also to understand it, by using a certification assistant.

Code-Carrying Theory (CCT)

Program synthesis follows the proof-as-program paradigm. That is, the programming problem is stated, i. e., it is specified in some logic, and then, an interactive theorem prover, for instance Coq or Isabelle, is used to build a proof. Finally, the source code can be extracted from the proof. For instance, CaML code can be extracted from Coq proofs. This is an alternative approach to program verification, where programs are correct or safe by construction and do not need to be verified a posteriori.

Musser and Vargun [Vargun, 2006] introduced Code-Carrying Theory

CCT, a certification system where, given some code consumer functional requirements, the code producer delivers code assertions and a functional correctness proof, but not the code itself. If the code consumer succeeds checking the proof, she can automatically generate correct code by using the CodeGen tool.

The code consumer supplies axioms, that are defined by means of functions. Then, these functions are proved to have some required properties, using conditional equations and induction. CCT transmits a set of axioms and theorems, i.e. a theory, and the proofs of the theorems. There is no need for code transmission neither verification condition generators.

CCT can be applied to general safety properties including security properties. However, in [Vargun, 2006], CCT is applied to functional correctness of generic types code, and type safety and termination. CCT is applied to generic types as numbers (sum-list) that can be instantiated with specific numbers (natural, integer, etc.). It also includes characteristics of generic data structures like the iterators of C++ Standard Template Library, but extended with range properties that provide generic types safety.

The CCT approach is composable. It has been applied to problems composed of different functions whose correctness were proved independently. In this case, Oz code was generated. Oz is a multi-paradigm functional, imperative, and object-oriented programming language. CCT is implemented in Athena, a higher-order theorem prover and functional programming language, that allows one to express proofs as functions. Athena computation ends with a theorem as an outcome, or generates an error condition. CCT produces generic proofs, i.e. proofs that are sound for generic types, and can be used to proof instances of the proved generic types.

Alpuente et al. [Alpuente et al., 2010a; Alpuente et al., 2010b] introduced a rewriting logic framework for CCT based on automatic program transformation, regarding sorts, rules, equational theories and algebraic laws. In this case, the program functional requirements are specified as a rewrite theory, from which a corresponding correct optimized program code can be obtained via fold/unfold transformations. The transmitted certificate includes the program requirements and the applied transformations. The proposed CCT program transformation framework is implemented in Maude and applies to Maude code synthesis.

The main high-level properties considered for high-level languages are summarized in Table 1.1.

Table 1.1: High-level languages and certified properties.

Security Policies	**Language**
The even result of an integer expression [Wu et al., 2003; Wu, 2005]	Functional ML
Bounded resource consumption (the size of heap memory used by a function linearly depends on input size) [Hofmann and Jost, 2003; Beringer et al., 2003; MacKenzie and Wolverson, 2004; Sannella et al., 2005; Gilmore and Prowse, 2005; Aspinall et al., 2004]	Functional Camelot
Secure file opening on temporal site [Albert et al., 2005a] Termination, determinacy, and non-failure [Albert et al., 2004] Bounded execution time, bounded data structures memory consumption, and no side-effects [Albert et al., 2004; Albert et al., 2005c]	Constraint logic Ciao
Bounded execution time, bounded data structures memory consumption explicitly demanded, heap memory, bandwidth [Mok and Yu, 2002a; Mok and Yu, 2002b; Yu and Mok, 2004]	Imperative C
Temporal properties specified as LTL predicates without next operators, like liveness [Xia and Hook, 2003a; Xia and Hook, 2003c; Xia and Hook, 2003b; Xia and Hook, 2004]	Imperative C
Operator safety regarding partial functions, memory safety and variable initialization safety [Whalen et al., 2003; Whalen et al., 2002; Schumann, 2003; Whalen et al., 2003]	Generated imperative C/C++ and Modula 2
Bounded resource consumption, including execution time, memory, bandwidth, files, threads and database connections [Chander et al., 2005a; Chander et al., 2007]	Imperative object-oriented Java
Initialization safety, and no out-of-bound arrays [Denney and Fischer, 2006a; Denney and Fischer, 2005; Denney and Fischer, 2006b]	C

High-level languages and certified properties

Continued from previous page	
Security Policies	**Language**
Functional correctness, termination and type safety [Vargun, 2006]	Multi-paradigm imperative, functional and object-oriented Oz, and imperative object-oriented C++
Correctness [Alpuente et al., 2010a; Alpuente et al., 2010b]	Rewriting Logic Maude

At the end of this thesis we provide a table with the comparison of the main approaches to software *certification* of security and safety properties for: (i) low-level languages, (ii) C language, (iii) Java, (iv) Camelot and (v) logic languages (Appendix A.1, A.2, A.3, A.4, and A.5, respectively). In general, we do not consider other approaches to software verification that do not explicitly deal with software certification, because there are too many of these approaches, and most of them do not deliver a formal proof that could be used as a certificate that would be independently validated by code consumers.

However, Chapters 5, 6 and 7 include specific proposals that deal with the considered properties, particularly type systems and verification approaches that are based on proof tools, i.e. theorem provers.

1.2 Thesis publications

In [Alba-Castro et al., 2008], we consider some integer arithmetic properties of Java source code of terminating and deterministic function methods with integer parameters and integer results. The integer properties (e.g. "modulo 2", i.e. parity, and "modulo 4") are specified as safety policies in the style of the specification language JML by using the `requires` and `ensures` clauses and the operator `\result`. The specified integer properties include simple relational properties of program variables (i.e. `i <= n`).

In [Alba-Castro et al., 2009a], we introduce our rewriting logic approach to non–interference analysis and certification of Java source code. This paper considers local non–interference of deterministic and terminating Java function methods. We focus on the methodology as well as the PCC and rewriting-based particulars of our approach, with a specific emphasis on practicality and good performance. The proof of concept prototypes developed in previous work [Alba-Castro et al., 2008; Alba-Castro et al., 2009a] were integrated into a web based tool (a running prototype) [Alba-Castro et al., 2009b], that is publicly available at `http://zenon.dsic.upv.es:8080/rewritingLogic/control`.

In [Alba-Castro et al., 2010a], we present a novel and sound security model for global non–interference of Java source code which approximates non–interference of complete Java classes as a safety property. This work formalizes foundational semantic security aspects regarding complete Java classes, but it also provides a comprehensive and full-fledged formulation of our abstract original non–interference certification methodology [Alba-Castro et al., 2009a], that considers Java methods. We also propose in [Alba-Castro et al., 2010a] a certification technique for global non–interference of complete Java classes based on rewriting logic. Starting from an existing Java semantics specification written in Maude, we developed an extended, information–flow Java semantics that allows one to correctly observe global non–interference policies. In order to achieve a finite state transition system, we also developed an abstract Java semantics that we use for secure and effective non–interference Java analysis. The analysis produces certificates that are independently checkable and are small enough to be used in practice.

In [Alba-Castro et al., 2010b], we extended the certification technique for non–interference of complete Java classes introduced in [Alba-Castro et al., 2010a], in order to consider non–interference policies with and without era-

sure policies. We implemented the global confidentiality certification methodology for non–interference and erasure, and developed some experiments that demonstrate the feasibility of our approach. A completely redesigned, easy to use, Web tool was developed [Alba-Castro et al., 2010c] and it is available at http://zenon.dsic.upv.es:8080/certificateX/.

The publications derived from this thesis are:

* Alba-Castro, M., Alpuente, M., and Escobar, S.: Automatic certification of Java source code in rewriting logic, in *12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007)*, Vol. 4916 of *Lecture Notes in Computer Science*, pp 200–217, Springer-Verlag, 2008.

* Alba-Castro, M., Alpuente, M., and Escobar, S.: Automated certification of non–interference in rewriting logic, in *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, Vol. 5596 of *Lecture Notes in Computer Science*, pp 182–198, Springer-Verlag, 2009.

* Alba-Castro, M., Alpuente, M., and Escobar, S.: Abstract certification of global non–interference in rewriting logic, in *Proc. 8th Int. Symp. Formal Methods for Components and Objects (FMCO 2009), Revised Lectures.*, Vol. 6286 of *Lecture Notes in Computer Science*, pp 105–124, Springer-Verlag, 2010.

* Alba-Castro, M., Alpuente, M., and Escobar, S.: Approximating non–interference and erasure in rewriting logic, in *12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2010) Sept. 23-26, Timisoara, Romania*, pp 124–132, IEEE Computer Society, 2010.

* Alba-Castro, M., Alpuente, M., and Escobar, S.: Confidentiality certification of source Java code in JavaPCC, in Guest Editors: Jens Bendisposto, Michael Leuschel (ed.), *Proceedings of the 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010)*, Vol. 35 of *Electronic Communications of the EASST*, To appear.

* Alba-Castro, M., Alpuente, M., Escobar, S., Ojeda, P., and Romero, D.: A tool for automated certification of Java source code in Maude,

in *Revised selected papers of Spanish Conference on Programming and Computer Languages, VIII Jornadas sobre Programación y Lenguajes (PROLE 2008)*, Vol. 248 of *Electronic Notes on Theoretical Computer Science*, pp 19–29, Elsevier, 2009.

Chapter 2

Preliminaries

2.1 Rewriting Logic and Maude

We assume some basic knowledge of term rewriting [TeReSe, 2003] and rewriting logic [Meseguer, 1992].

Maude is a high-level, declarative programming language supporting executable specification written in rewriting logic. Maude supports both equational and rewriting logic computation because rewriting logic includes equational logic as a sublogic [Clavel et al., 2002; Clavel et al., 2005; Clavel et al., 2007]. A Maude program is a rewriting logic theory, that includes an equational subtheory. A Maude computation is a logical deduction w.r.t. the rewriting rules of the rewriting theory and the equations of the equational theory.

The triple $\langle K, \Sigma, S\rangle$ is a membership equational logic *signature*, with K a set of kinds, $\Sigma = \{\Sigma_{(w,k)}\}_{(w,k)\in K^*\times K}$ a many-kinded signature, and $S = \{S_k\}_{k\in K}$ a K-kinded disjoint sets of sorts [Meseguer and Roşu, 2007]. $[s]$ denotes the kind of sort s. A membership equational logic K-algebra A contains a set A_K for each kind $k \in K$, a function $A_f : A_{k_1} \times \ldots A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1\ldots k_n,k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ denote, respectively, the set of ground Σ-terms with kind k and the set of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of kinded variables. In the following, Σ denotes the triple $\langle K, \Sigma, S\rangle$. Given the signature Σ, the atomic formulae are either a Σ-equation of the form $t = t'$, or a Σ-membership of the form $t : s$, with $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$. Σ-sentences are conditional formulae of the form $(\forall X)\ \varphi$ if $(\bigwedge_i p_i = q_i) \wedge (\bigwedge_j w_j : s_j)$ where φ is either a Σ-equation or a Σ-membership, and all the variables in φ, p_i, q_i, and w_j are in X.

A membership equational logic theory is a pair $\langle \Sigma, E\rangle$, with Σ a membership equational logic signature and E a set of Σ-sentences. Given a membership equational logic theory $\langle \Sigma, E\rangle$, its initial algebra is denoted $T_{\Sigma/E}$ and its

elements are the E-equivalen- ce classes of ground terms in T_Σ. An operator specification $f : s_1 \times \ldots s_n \rightarrow s$ corresponds to specifying f at the kind level and giving the axiom: $\forall x_1 : k_1, \ldots, x_n : k_n, f(x_1, \ldots, x_n) : s$ if $\bigwedge_{1 \leq i \leq n} x_i = s_i$. The equation specification $\forall x_1 : k_1, \ldots, x_n : k_n, t = t'$ if $\bigwedge_{1 \leq i \leq n} x_i : s_i$, corresponds to specifying $\forall x_1 : s_1, \ldots, x_n : s_n, t = t'$.

The logical deduction of the equational logic is the equational deduction using the equations of the theory, the axioms of the operators, and the five natural deduction rules: reflexivity, symmetry, transitivity, congruence and substitution [Chen et al., 2006].

A membership equational logic theory have some constraints regarding program execution [Meseguer and Roşu, 2007]: i) the sentences of the theory may be decomposed as $(E \cup A)$, where A is a set of equations that are used for computation modulo; for instance, A may include any combination of associativity, commutativity and idempotency laws for operators defined in Σ; and ii) the sentences E have to be A-confluent and A-terminating, in order to allow one to use the conditional equations in E as (equational) rewrite rules modulo A. Therefore, regarding program execution, a membership equational logic theory can be specified as the pair $(\Sigma, E \cup A)$, where the A axioms are used to rewrite modulo.

Maude has functional and system modules. The Maude's functional modules are theories in membership equational logic. Computation in a functional module is the equational deduction accomplished by rewriting, orienting the equations as rewriting rules from their left-hand side to their right-hand side, until a canonical form is found. Given the above constraints of the membership equational logic theory, the Maude's functional modules can be assumed to be confluent, terminating and sort-decreasing, and the conditional equational rewriting can be performed modulo the A axioms of associativity, commutativity and identity eventually considered [Clavel et al., 2007].

The axioms of associativity, commutativity and identity are specified in Maude as operator symbol attributes. As an example, the following Maude specification, where the multiset union operator (denoted by the empty string, i. e., the empty string is the operator symbol itself) has the attributes of commutativity and associativity:

```
op __ : State State -> State [comm assoc]
```

The attributes `comm` and `assoc` mean that the multiset union operator satisfies respectively the commutativity and associativity axioms that are specified

by the following equations without the need to give them explicitly [Clavel et al., 2005]:

```
X ( Y Z ) = ( X Y ) Z
X Y = Y X
```

Maude uses a multiset matching algorithm in which the multiset union is matched modulo associativity and commutativity. The rewriting performed onto terms of the multiset is multiset rewriting.

A rewriting logic specification or theory is a tuple $\langle \Sigma, E \cup A, R \rangle$, where $(\Sigma, E \cup A)$ is a membership equational theory with axioms A, and R is a collection of labelled and possibly conditional rewrite rules of the form: $r : (\forall X)$ $t \rightarrow t'$ if $(\bigwedge_i \ u_i = u'_i) \wedge (\bigwedge_j \ v_j : s_j) \wedge (\bigwedge_l \ w_l \rightarrow w'_l)$, where: i) the variables in the terms $t, t', u_i, u'_i, v_j, w_l$ and w'_l are all in X, ii) the terms in each equation or rewriting rule, have the same kind, and iii) the term v_j in the membership $v_j : s_j$ has the kind $[s_j]$. The rewriting rules are computationally interpreted as local transition rules in a possibly concurrent system. The rewriting logic theory specifies a concurrent system where the rewriting rules in R specify the concurrent transition rules between system states. These states are elements of the initial algebra $T_{\Sigma/E \cup A}$, specified by $(\Sigma, E \cup A)$.

The rewrite theory can prove sentences that are universally quantified rewrites of the form $(\forall X)\ t \rightarrow t'$, with $t, t' \in T_\Sigma(X)_k$ for some kind k, that are obtained by the finite application of the five inference rules: reflexivity, equality, congruence, replacement and transitivity [Meseguer and Roşu, 2007]. The rewrite rules are intuitively interpreted as inference rules in a logical system.

The Maude's system modules are rewrite theories $\langle \Sigma, E \cup A, R \rangle$. Rewriting in $\langle \Sigma, E \cup A, R \rangle$ happens modulo the equational axioms in A, that are specified as the equational attributes given to the function symbols in the program. Computation in system modules uses the equations to simplify the terms to its canonical form before applying the rules to generate transitions. The rules in the rewrite theory need not be confluent and need not be terminating, but they must be coherent with respect to its equations. Then, given a set of rules, many different rewriting paths are possible from a term. The rules are applied concurrently, hence a rewrite specification is a compact way to encode concurrent and non-deterministic transition systems [Chen et al., 2006].

In Maude functional modules, we can evaluate expressions with the `reduce` (`red`) comand. The `reduce` command simplifies the expression, a term,

to its canonical form, by using the equations and membership axioms in the module.

In Maude system modules, we can execute rewrite theories with the `rewrite` (`rew`) command. The `rewrite` command applies rules to a term until termination, in a top-down and rule-fair way giving a chance to all rules. However, before applying any rule to a term, it is reduced to its canonical form by using the equations of the theory. Because we can have non-terminating computations, the rewrite command can be given a numeric argument stating the maximum number of rewrite steps. Since the rewriting rules of the theory could correspond to a non-deterministic transition system, the `rewrite` command only gives one possible behavior among many.

To obtain all behaviors from an initial state, we can use the `search` command, which is Maude's search facility for doing breadth-first search with cycle detection. The `search` command looks for all the rewrites of a given state into states that match a given pattern satisfying some condition. When a `search` command terminates, the state graph is retained in memory so that it is possible to obtain the whole generated search graph to interrogate the state graph for the path(s) from the start term to any reachable state [Clavel et al., 2003]. Each path includes the states and the rewriting rules used in the transitions, and the state path graph can be used as a witness or certificate of the rewriting logic deduction.

Since equations are deterministic, the state space associated to a rewrite theory is determined in Maude only by the program rules. That is, rules and equations are applied in the same way, but Maude, by default, only keeps track of the rules applied and omits the information about the equations applied. Therefore, the number of rules and equations is relevant and the smaller the number of rules, the more efficient the verification analysis, since the search space is smaller.

Let us provide a concrete example. The Maude simple system module given in Figure 2.1, specifies a timer as a decreasing counter (seconds). The initial state specifies the time period start in whatever combination of seconds, minutes and hours, using at least one of their respective constructor operations `sec`, `min` and `hou`. The timer stops when it reaches the final state "`sec(0) min(0) hou(0)`".

The terms "`sec(7290)`", "`sec(89) min(127)`" and "`sec(45) min(49) hou(4)`" are valid initial states. The last of these terms is the only one in canonical form because it cannot be reduced by using the equations in the

```
mod TIMER is
  pr INT .
  sort Time  .
  op sec : Int -> Time .
  op min : Int -> Time .
  op hou : Int -> Time .
  op __ : Time Time -> Time [comm assoc] .
  vars I1 I2 I3 : Int .
--- EQUATIONS:
  eq sec(I1) sec(I2) = sec(I1 + I2) .
  ceq sec(I1) = sec(I1 rem 60) min(I1 quo 60) if I1 >= 60 .
  eq min(I1) min(I2) = min(I1 + I2) .
  ceq min(I1) = min(I1 rem 60) hou(I1 quo 60) if I1 >= 60 .
  eq hou(I1) hou(I2) = hou(I1 + I2) .
--- RULES:
  crl [seconds] :
      sec(I1)  => sec(I1 - 1)  if I1 > 0 .
  crl [minutes] :
      sec(0) min(I2)  => sec(59) min(I2 - 1)  if I2 > 0 .
  crl [hours] :
      sec(0) min(0) hou(I3) => sec(59) min(59) hou(I3 - 1) if I3 > 0 .
endm
```

Figure 2.1: Maude TIMER example.

module.

The equations reduce the terms of the sort `Time` to its standard time format in seconds, minutes and hours. The `reduce` command "`red sec(7290) .`" produces the canonical form of the term `sec(7290)`, which is "`sec(30) min(1) hou(2)`" , as follows:

```
Maude> red sec(7290) .
result Time: sec(30) min(1) hou(2)
```

The canonical form of the term `sec(89) min(127)` produced by the `reduce` command "`red sec(89) min(127) .`" is as follows:

```
Maude> red sec(89) min(127) .
result Time: sec(29) min(8) hou(2)
```

The term `sec(45) min(49) hou(4)` cannot be further reduced by the `reduce` command "`red sec(45) min(49) hou(4) .`" as shown in the following Maude execution excerpt:

```
Maude> red sec(45) min(49) hou(2) .
result Time: sec(45) min(49) hou(2)
```

The rules specify the time transitions as a decreasing counter second by second. The result of the application of the rewrite command to the initial state represented by the term `sec(45) min(49) hou(4)` is the following:

```
Maude> rew sec(45) min(49) hou(2) .
result Time: sec(0) min(0) hou(0)
```

The search command can be used to analyze the reachability of a given state from an initial one. For example the command "`search sec(888888) =>* sec(0) min(0) hou(0) .`" can be used to determine that the final state `sec(0) min(0) hou(0)` is reachable from the initial state `sec(888888)` as the following result shows:

```
Maude> search sec(888888) =>* sec(0) min(0) hou(0) .
Solution 1 (state 888888)
empty substitution
```

The following result shows a final state that represents a time count (`sec(45) min(23) hou(10)`) that cannot be reached from an initial state that represents a lower time count (`sec(7896)`):

```
Maude> search sec(7896) =>* sec(45) min(23) hou(10) .
No solution.
```

2.2 Abstract Interpretation

Abstract interpretation is a theory of semantic approximation in which concrete domain and operations are approximated by abstract domain and operations. In the following we briefly recall some key aspects of the abstract interpretation framework [Cousot and Cousot, 1977; Cousot and Cousot, 1979].

Let us begin by recalling some useful notions about sets, relations and transition systems.

Sets

A set S is a collection of its elements $S = \{s_1, s_2, \dots\}$ such that we can write $s_i \in S$ to denote that s_i is an element of S. There is a special set with no element, the empty set denoted by $\emptyset$.

A set S_1 is a subset of set S_2, written $S_1 \subseteq S_2$ if every element of S_1 is also element of S_2, i. e., for all $s \in S_1 \Rightarrow s \in S_2$. $\emptyset \subseteq S$ for all set S. Given a

set S, the set of all subsets of set S, written $\wp(S)$, is the set $\{SS \mid SS \subseteq S\}$. For every set S, $S \subseteq \wp(S)$ holds. Given two sets S_1 and S_2, their union, written $S_1 \cup S_2$, is the set S_3 such that $s \in S_3$ if $s \in S_1 \vee s \in S_2$. Given two sets S_1 and S_2, their intersection, written $S_1 \cap S_2$, is the set S_3 such that $s \in S_3$ if $s \in S_1 \wedge s \in S_2$.

Two sets S_1 and S_2 are equal if and only if they have the same elements: $S_1 = S_2 \Leftrightarrow$ for every $s \in S_1$ we have $s \in S_2$, and vice versa.

A set can be defined by a property (i. e., a predicate whose variables range over set elements): $S = \{s \mid P(s)\}$, the elements of S are such that they satisfy the property.

Relations and functions

A Cartesian product of two sets S_1 and S_2, written $S = S_1 \times S_2$, is the set whose elements are pairs formed with an element of each set: $S_1 \times S_2 = \{\langle s_1, s_2\rangle \mid s_1 \in S_1 \text{ and } s_2 \in S_2\}$. A relation R between two sets S_1 and S_2 is a set $R \subseteq (S_1 \times S_2)$. Two elements s_1 and s_2 of S_1 and S_2, respectively, are related by R if $\langle s_1, s_2\rangle \in R$.

A function f from the (domain) set S_1 to the (range) set S_2, written $f : S_1 \to S_2$ is a relation between the two sets, such that if $\langle s_1, s_2\rangle \in f$, written $f(s_1) = s_2$, there is not a $s_3 \in S_2$ such that $s_2 \neq s_3 \wedge f(s_1) = s_3$. A function $f : S_1 \to S_2$ is *surjective* (or onto) if for every $s_2 \in S_2$ there exist $s_1 \in S_1$ such that $f(s_1) = s_2$. A function $f : S_1 \to S_2$ is *injective* (or one-to-one) if for every $s_1, s_1' \in S_1$ with $s_1 \neq s_1'$ there exist $f(s_1), f(s_1') \in S_2$ such that $f(s_1) \neq f(s_1')$.

Complete lattices

A partially ordered set $(S, \leq)$ is a set S with an order relation $\leq$ such that, for all $x, y, z \in S$, it satisfies reflexivity ($x \leq x$), transitivity ($x \leq y \wedge y \leq z \Rightarrow x \leq z$), and antisymmetry properties ($x \leq y \wedge y \leq x \Leftrightarrow x = y$). A totally ordered set $(S, \leq)$ is a partially ordered set such that, for all $x, y \in S$, it satisfies $x \leq y \vee y \leq x$.

Given a partially ordered set $(S, \leq)$ and $X \subseteq S$, if for all $x \in X$, it holds $x \leq y$ with $y \in S$, y is an upper bound for X. If exists $y \in S$, such that y is an upper bound for X, and for every upper bound $y' \in S$ for X it holds $y \leq y'$, then y is the least upper bound for X, *lub* or *join*, denoted by $\sqcup X$. Dually,

given $X \subseteq S$, if for all $x \in X$, it holds $y \leq x$ with $y \in S$, y is a lower bound for X. If there exists $y \in S$, such that y is a lower bound for X, and for every lower bound $y' \in S$ for X it holds $y' \leq y$, then y is the greatest lower bound for X, *glb* or *meet*, denoted by $\sqcap X$.

A *complete lattice* is a partially ordered set $(S, \leq)$ such that, for every $X \subseteq S$ there exist $\sqcup X$ and $\sqcap X$. $\sqcup S$ is denoted by $\top$, and $\sqcap S$ by $\bot$. Such a complete lattice is denoted by $\langle S, \leq, \bot, \top, \sqcup, \sqcap \rangle$. For instance, the set of all possible subsets of any set S, $\wp(S)$ and the $\subseteq$ relationship form a complete lattice $\langle \wp(S), \subseteq, \emptyset, S, \cup, \cap \rangle$ where $\sqcup$ is set union, $\sqcap$ is set intersection, the $\top$ is S and $\bot$ is $\emptyset$.

A *semi lattice* is a partially ordered set $(S, \leq)$ such that, for every $X \subseteq S$, there exist either $\sqcup X$ (join semi lattice) or $\sqcap X$ (meet semi lattice), but not both.

Galois connection

Given two monotone functions $\alpha : A \to B$ and $\gamma : B \to A$, with $(A, \leq)$ and $(B, \leqslant)$, partially ordered sets, such that for all $a \in A$ and for all $b \in B$, $\alpha(a) \leqslant b$ if and only if $a \leq \gamma(b)$, then $\langle \alpha, \gamma \rangle$ is a *Galois connection*: $\langle A, \leq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle B, \leqslant \rangle$.

Galois connections satisfy the properties:

1. *Deflationary*: For all $a \in A$ it holds $a \leq \gamma(\alpha(a))$.
2. *Inflationary or extensive*: For all $b \in B$ it holds $\alpha(\gamma(b)) \leqslant b$.
3. *Idempotent*: For all $b \in B$, $\alpha(\gamma(\alpha(\gamma(b)))) = \alpha(\gamma(b))$.

Galois insertion

Given two functions $\alpha : A \to B$ and $\gamma : B \to A$, with $\langle A, \leq \rangle$ and $\langle B, \leqslant \rangle$, partially ordered sets, we can state that $\langle \alpha, \gamma \rangle$ is a Galois surjection or *Galois insertion*, i. e., $\langle A, \leq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle B, \leqslant \rangle$ if, for every $a \in A$ and $b \in B$, it holds $\alpha(a) \leqslant b \Leftrightarrow a \leq \gamma(b)$.

The α, γ functions of a Galois insertion satisfy the properties:

1. *Monotonicity* of α and γ: For all $a, a' \in A$ and for all $b, b' \in B$, it holds $a \leq a' \Rightarrow \alpha(a) \leqslant \alpha(a')$ and $b \leqslant b' \Rightarrow \gamma(b) \leq \gamma(b')$.

2. *Deflationary*: For all $a \in A$ it holds $a \leq \gamma(\alpha(a)))$.

3. Abstraction and concretization means no *information loss*: For all $b \in B$ it holds $b = \alpha(\gamma(b))$. This implies that α is *suryective* and γ is *inyective*.

If $\langle \alpha, \gamma \rangle$ is a Galois insertion, we also have: i) $\alpha \circ \gamma$ is idempotent, i. e., $\alpha \circ \gamma(b) = \alpha \circ \gamma(\alpha \circ \gamma(b))$, and ii) α is a *surjective* function, and γ is an *inyective* function.

A *Galois connection* where α is a *surjective* function and γ is an *inyective* function, is a *Galois insertion*.

Transition systems

Programs can be formalized as transition systems $\tau = \langle \Sigma, \Sigma_{ini}, t \rangle$, where Σ is a set of program states, Σ_{ini} is the set of initial states $\Sigma_{ini} \subseteq \Sigma$, and $t \subseteq \Sigma \times \Sigma$ is a transition relation between a state and its possible successors. A finite partial program execution trace is a sequence of states $s_0\ s_1 \ldots s_n$, where $s_0 \in \Sigma_{ini}$, $s_i \in \Sigma$ for $0 \leq i \leq n$, and $\langle s_i, s_{i+1} \rangle \in t$, for $0 \leq i \leq n-1$. A Program state s_i is defined by a mapping that holds the bindings of program variable identifiers to their domain values in the considered state, i. e., $s_i : Variable \rightarrow Value$. The (concrete) value $Val \in Value$ of a given variable $Var \in Variable$ in state s_i is obtained by the expression $s_i(Var)$, i. e., $Val = s_i(Var)$.

Example 1. For example, a finite partial execution trace corresponding to the Java program `int i = 0; while (i < 10) i++;` is the state sequence $s_0\ s_1 \ldots s_9$, such that `i` is a Java integer variable, i. e., $s_i(Var) \in \mathsf{Int}$. and $s_0(\mathtt{i}) = 0 \wedge s_j(\mathtt{i}) = s_{j-1}(\mathtt{i}) + 1$, for all $j \in 1 \ldots 9$.

If Σ_τ^n denotes the set of all finite partial execution traces of length n of the transition system τ, the collecting semantics of τ, denoted by $\Sigma_\tau^{\overrightarrow{*}}$, is the set of all partial traces of any finite length of τ, i. e., $\Sigma_\tau^{\overrightarrow{*}} = \bigcup_{0 \leq n} \Sigma_\tau^n$ [Cousot, 2004].

A terminating computer program can have an infinite number of different initial program execution states and also infinite number of partial finite execution traces. The corresponding collecting semantics can also have an infinite number of partial traces of any finite length. This state explosion makes infeasible to apply most program analysis techniques. However, there are program analysis techniques that correctly approximate program semantics in such a way that programs have finite initial program states and finite collecting semantics, so that their analysis is feasible in finite time. Moreover,

there are program properties that don't need every program execution state and every partial execution trace to be analyzed because they are too precise to express program properties under consideration. In these cases, we can use abstraction.

Abstraction

Abstract interpretation is concerned with a particular underlying structure of the concrete universe of computation. Abstract interpretation of programs produces a summary of some aspects of the concrete executions. This summary is in general inaccurate and incomplete but it allows one to answer questions which do not need full knowledge of the program execution [Cousot and Cousot, 1977]. For instance, if we are interested in the parity (an integer arithmetic property) of the final value of variable `i` in the `while` program given in Example 1, we can consider only the parity of the value of the variable `i` in the initial and final states, disregarding both, the intermediate states of the trace, and the specific concrete integer values of the program variable `i` in the initial and final states.

Given a set of objects O (either program states or finite partial execution traces) we represent the property P as the set $P^{\#} \subseteq O$, i. e., $P^{\#} = \{o \in O \mid P(o)\}$. Regarding program states, the property P of states $s \in \Sigma$ is represented by the set of states $P^{\#} \in \wp(\Sigma)$ which have the property P; i. e., $P^{\#} \subseteq \Sigma$ and $P^{\#} = \{s \in \Sigma \mid P(s)\}$. The correspondence between a state and its abstraction is specified by an abstraction function $\alpha : \wp(\Sigma) \mapsto \wp(\Sigma)$, that is based on the properties at hand. At the variable level, there is a corresponding abstraction function $\alpha : \wp(Value) \to \wp(Value)$, that maps sets of concrete values into properties or sets of concrete values.

For example, the properties *even* and *odd* of values of variable `i` can be represented as the sets of values $\#even, \#odd \subseteq \mathsf{Int}$, such that $\#even = \{\mathtt{i} \in \mathsf{Int} \mid \mathtt{i} \mod 2 = 0\}$, and $\#odd = \{\mathtt{i} \in \mathsf{Int} \mid \mathtt{i} \mod 2 \neq 0\}$. In this case, at the variable level we can use the abstraction function , $\alpha : \wp(\mathsf{Int}) \to \wp(\mathsf{Int})$, with $\alpha(\{\mathtt{Val}\}) = \#even$ where `Val` is the value of variable `i`, and it is such that $\mathtt{Val} \mod 2 = 0$ and $\alpha(\{\mathtt{Val}\}) = \#odd$ if `Val` is such that $\mathtt{Val} \mod 2 \neq 0$. We can extend homomorphically the α function from the variable level to program states by using the abstraction function $\alpha : \wp(\Sigma) \to \wp(\Sigma)$, such that given $S \subseteq \wp(\Sigma)$, we have: i) $\alpha(S) = \#even$ if for all $s \in S$, $s(\mathtt{i}) \ mod \ 2 = 0$; ii) $\alpha(S) = \#odd$ if for all $s \in S$, $s(\mathtt{i}) \ mod \ 2 \neq 0$; and iii)

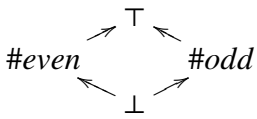

Figure 2.2: Abstract lattice on Int values regarding their parity.

$\alpha(S) = \mathsf{Int}$ if exists $s, s' \in S$ such that $s(\mathtt{i})\ mod\ 2 = 0$ and $s'(\mathtt{i})\ mod\ 2 \neq 0$. We also have that $\alpha(\emptyset) = \emptyset$.

The set $\#Parity = \{\bot, \#even, \#odd, \top\}$ is a complete lattice $\langle \#Parity, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ induced on integers by the parity property based on set inclusion $\sqsubseteq$, where $\bot$ denotes the set $\emptyset$ and $\top$ denotes the set Int (see Figure 2.2). The *lub* is $\top$ (i.e. $\sqcup \#Parity = \top$), the *glb* is $\bot$ (i.e. $\sqcap \#Parity = \bot$), $\#even \sqsubseteq \top$, and $\#odd \sqsubseteq \top$.

More generally, the set of properties of states in Σ is a complete Boolean lattice $\langle \wp(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap, \neg \rangle$ [Cousot, 2004].

There is another function that gives the concrete form of an abstract state: the concretization function $\gamma : \wp(\Sigma) \mapsto \wp(\Sigma)$. Function γ has also a corresponding concretization function at variable level $\gamma : \wp(Value) \rightarrow \wp(Value)$. In the example, $\gamma(\#even) = \#even$, $\gamma(\#odd) = \#odd$, $\gamma(\mathsf{Int}) = \mathsf{Int}$, and $\gamma(\emptyset) = \emptyset$.

An abstract interpretation $\langle \Sigma^{\#}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ of a program is a complete semi-lattice that is *consistent* with a concrete interpretation $\langle \wp(\Sigma), \subseteq, \bot, \top, \cup, \cap \rangle$ if the abstract state $s^{\#} \in \Sigma^{\#}$ resulting from the abstract interpretation is a correct approximation of the concrete state $\{s\} \in \wp(\Sigma)$ resulting from the concrete and more refined interpretation. Correctness of the approximation means that the abstract state contains at least the concrete state, i. e., $\{s\} \sqsubseteq s^{\#}$ [Cousot and Cousot, 1977]. Formally, this can be expressed by using the correspondence between concrete and abstract states (abstraction: $\alpha : \wp(\Sigma) \rightarrow \Sigma^{\#}$) and inversely between abstract and concrete states (concretization: $\gamma : \Sigma^{\#} \rightarrow \wp(\Sigma)$) and requiring [Cousot and Cousot, 1977]:

1. For all $S \in \wp(\Sigma)$ given $\alpha(S) = S^{\#}$, it holds that $S \subseteq \gamma(S^{\#})$ and $\alpha(S) \sqsubseteq S^{\#}$.

2. The α and γ functions are monotonic or order-preserving, i. e., for all $S, S' \in \wp(\Sigma)$, given $S \subseteq S'$ we have that $\alpha(S) \subseteq \alpha(S')$, and for all $S^{\#}, S_1^{\#} \in \Sigma^{\#}$, $S^{\#} \sqsubseteq S_1^{\#}$ implies that $\gamma(S^{\#}) \subseteq \gamma(S_1^{\#})$; this is necessary

because state inclusion must be preserved by the abstraction and concretization processes.

3. For all $S^{\#} \in \Sigma^{\sharp}$, i.e. $S^{\#} \in \wp(\Sigma)$, it holds $S^{\#} = \alpha(\gamma(S^{\#}))$; this means that concretization introduces no loss of information and implies α is surjective and γ is injective.

4. For all $S \in \wp(\Sigma), S \subseteq \gamma(\alpha(S))$; this introduces the idea of approximation: the abstraction of a set of concrete states S, $\alpha(S)$, may introduce some loss of information so that when concretizing again we may get a larger content $S \subseteq \gamma(\alpha(S))$.

Formally, $\Sigma^{\sharp}$ is an abstract interpretation of Σ ($\Sigma^{\sharp}$ approximates Σ or conversely Σ is a refinement of Σ) if and only if there exist functions $\alpha : \wp(\Sigma) \to \wp(\Sigma)$ ($\alpha : \wp(\Sigma) \to \Sigma^{\sharp}$) and $\gamma : \wp(\Sigma) \to \wp(\Sigma)$ ($\gamma : \Sigma^{\sharp} \to \wp(\Sigma)$), such that conditions 1, 2, 3 and 4 hold [Cousot and Cousot, 1977].

The required conditions means that α and γ functions form a *Galois surjection*[1] [Cousot and Cousot, 1979; Cousot and Cousot, 2002; Cousot, 2004].

Definition 1 (**Abstract interpretation** [Cousot and Cousot, 1979; Cousot and Cousot, 2002; Cousot, 2004])**.** Formally, $\Sigma^{\sharp}$ is an abstract interpretation of Σ if there exist functions $\alpha : \wp(\Sigma) \to \wp(\Sigma)$ ($\alpha : \wp(\Sigma) \to \Sigma^{\sharp}$) and $\gamma : \wp(\Sigma) \to \wp(\Sigma)$ ($\gamma : \Sigma^{\sharp} \to \wp(\Sigma)$), such that $\langle \alpha, \gamma \rangle$ is a *Galois surjection*, i. e., $\langle \Sigma, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow}} \langle \Sigma^{\sharp}, \sqsubseteq \rangle$.

There are two directions of approximation. In the above approximation, $S \in \wp(\Sigma)$ is over-approximated by $\bar{S}^{\#} \in \wp(\Sigma)$ such that $S \sqsubseteq \bar{S}^{\#}$. In the approximation from below, $S \in \wp(\Sigma)$ is under-approximated by $\underline{S}^{\#} \in \wp(\Sigma)$ such that $\underline{S}^{\#} \sqsubseteq S$ [Cousot, 2004]. The condition 4 means that $\Sigma^{\sharp}$ is an over-approximation.

If we use a concretization that means a loss of information, relaxing condition 3 such that instead of equality we have inclusion, i. e., for all $S^{\#} \in \Sigma^{\sharp}$ it holds $\alpha(\gamma(S^{\#})) \sqsubseteq S^{\#}$. In this, case conditions 1, 2, the relaxed condition 3 and condition 4 mean that the functions α and γ are such that $\langle \alpha, \gamma \rangle$ is a *Galois connection*: $\langle \Sigma, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \Sigma^{\sharp}, \sqsubseteq \rangle$ [Cousot and Cousot, 2002; Cousot, 2004].

[1] It is also called a Galois insertion

Other equivalent formalization uses closure operators in order to prove the soundness of the abstraction [Cousot, 2004; Cousot and Cousot, 1979].

Intuitively, abstraction at variable level means that any set of values in the concrete domain $Value$, an element $SV \in \wp(Value)$, can be approximated by any $SV' \in \wp(Value)$ such that $SV \subseteq SV'$. In the abstract domain $Value^\sharp = \wp(Value)$, $SV^\sharp$ can be approximated by any $SV \in Value^\sharp$ such that $SV \subseteq \gamma(SV^\sharp)$, but the most precise such abstract approximation is $SV^\sharp = \alpha(SV)$.

Abstraction soundness also requires the correctness of the abstract operations that need to be defined over abstract domains [Cousot and Cousot, 1977; Cousot and Cousot, 1979; Cousot and Cousot, 2002]. Abstract operations must corresponds to concrete operations regarding abstraction and concretization functions, even though they can be less precise.

Abstraction operation (and function) soundness can be stated as follows regarding binary operations. Abstract binary operation correctness regarding the abstraction function at variable level, i. e., $\alpha : \wp(\mathsf{Value}) \rightarrow \mathsf{Value}^\#$, means that, the $\oplus$ binary operation (over concrete Value domain) and the corresponding $\oplus^\sharp$ binary operation (over abstract $\mathsf{Value}^\sharp$ domain) are such that: given $Val_0^\sharp = \alpha_{Var}(\{Val_0\}), Val_1^\sharp = \alpha_{Var}(\{Val_1\}), Val_2^\sharp = \alpha_{Var}(\{Val_1\})$ with $Val_0^\sharp, Val_1^\sharp, Val_2^\sharp \in \mathsf{Value}^\sharp$ and $Val_0, Val_1, Val_2 \in \mathsf{Value}$, if $Val_2 = Val_0 \oplus Val_1$ then it holds that $Val_2^\sharp = Val_0^\sharp \oplus^\sharp Val_1^\sharp$.

Abstract binary operation correctness regarding the concretization function at variable level, $\gamma : \mathsf{Value}^\# \rightarrow \wp(\mathsf{Value})$, means that the abstract $\oplus^\sharp$ binary operation and the corresponding concrete $\oplus$ binary operation are such that the following holds: given $\gamma(Val_0^\sharp) = SetVal_0$, $\gamma(Val_1^\sharp) = SetVal_1$, and $\gamma(Val_2^\sharp) = SetVal_2$ with $Val_0^\sharp, Val_1^\sharp, Val_2^\sharp \in \mathsf{Value}^\sharp$ and $SetVal_0, SetVal_1, SetVal_2 \in \wp(\mathsf{Value})$, we have that, if $Val_2^\sharp = Val_0^\sharp \oplus^\sharp Val_1^\sharp$ then there exist $Val_0 \in SetVal_0, Val_1 \in SetVal_1, Val_2 \in SetVal_2$ such that $Val_2 = Val_0 \oplus Val_1$.

Regarding the above Java program given in Example 1 and the parity integer property, we can specify the abstract post increment operator over $\#Parity$ abstract values, as shown in Figure 2.3. The correctness of the specification in Figure 2.3 can be proved straightforward by simply using the specification of the corresponding abstraction function at variable level.

At the variable level, the abstraction of program variables can be done independently when the abstract value of any variable does only depend on the concrete value of the variable, and it does not depend on the values of any

Old value of `i`	`i++`	*New value of* `i`
$\top$	$\top$	$\top$
$\sharp even$	$\sharp even$	$\sharp odd$
$\sharp odd$	$\sharp odd$	$\sharp even$
$\bot$	$\bot$	$\bot$

Figure 2.3: Abstract post increment ++ integer operator.

other variable. In this case, the abstraction is not relational. When abstraction of program variables take into account the relations between the concrete values of the variables, we have a more precise abstraction called relational. Cousot et al. introduce relational abstraction regarding linear relations between program variables [Cousot and Halbwachs, 1978].

Chapter 3

The Java Rewriting Logic Semantics

In the following, we briefly describe the rewriting logic semantics of Java given in [Farzan et al., 2007] and used by the JavaFAN verification tool [Farzan et al., 2004a; Farzan et al., 2004b]. Its novelty and interest are based on the following four advantages: (i) formal specifications provide a rigorous semantic definition for a language that can be mathematically scrutinized; (ii) such formal specifications can be developed with relatively little effort [Şerbănuţă et al., 2009; Meseguer and Roşu, 2007], even for large languages like Java [Farzan et al., 2004a] and the JVM [Farzan et al., 2004b]; (iii) the Maude programming language [Clavel et al., 2007], which implements rewriting logic, provides a formal analysis infrastructure, so that its formal analysis tools (such as state-space breadth-first search and LTL model checking) become available for free for each programming language that is specified in Maude; and (iv) in spite of their generality, those formal analyses can be performed with competitive performance (see [Farzan et al., 2004a]).

In [Farzan et al., 2007; Meseguer and Roşu, 2007], a sufficiently large subset of the full Java 1.4 language is specified in Maude, including multithreading, inheritance, polymorphism, object references, and dynamic object allocation. However, Java native methods and most of the Java built-in libraries available are not supported. The specification of the Java operational semantics is a rewrite theory, that is, a triple $\mathcal{R}_{\text{Java}} = (\Sigma_{\text{Java}}, E_{\text{Java}}, R_{\text{Java}})$, with Σ_{Java} an order-sorted signature, $E_{\text{Java}} = \Delta_{\text{Java}} \uplus B_{\text{Java}}$ a set of Σ_{Java}-equational axioms where B_{Java} are algebraic axioms such as associativity, commutativity and identity and Δ_{Java} is a set of terminating and confluent (modulo B_{Java}) equations. Finally, R_{Java} is a set of Σ_{Java}–rewrite rules that are not required to be confluent nor terminating.

Intuitively, the sorts and function symbols in Σ_{Java} describe the static structure of the Java program state space as an algebraic data type, the equations in Δ_{Java} describe the operational semantics of its deterministic features, and

the rules in $\mathcal{R}_{\text{Java}}$ describe its concurrent features. Following the rewriting logic framework [Meseguer, 1992], we denote by $u \rightarrow^{r}_{\text{Java}} v$ the fact that concrete terms u, v, denoting Java program states, are rewritten (at the top position, see [Farzan et al., 2007]) by using r, which is either a rule in R_{Java} or an equation in Δ_{Java} both applied modulo B_{Java}. We simply write $u \rightarrow_{\text{Java}} v$ when no confusion can arise. We denote by $\rightarrow^{*}_{\text{Java}}$ the extension of $\rightarrow_{\text{Java}}$ to multiple rewrite steps, i.e., $u \rightarrow^{*}_{\text{Java}} v$ if there exist $u_1, \ldots, u_k$ such that $u \rightarrow_{\text{Java}} u_1 \rightarrow_{\text{Java}} u_2 \cdots u_k \rightarrow_{\text{Java}} v$. Associativity, commutativity and unity (written ACU) axioms of binary operations in B_{Java} allow us to elegantly and effectively define (and implicitly implement) the crucial infrastructure of the Java programming language, including environments, threads, memory, input/output, synchronization information, and stores as well as the lookup operations on them. All of them are implemented as a multiset union operation that builds up a "soup" of elements; see [Farzan et al., 2007; Meseguer and Roşu, 2007]. The Java operational semantics contains about 424 equations and only 7 rules, which considerably saves memory and execution time during the verification analysis of Java programs [Farzan et al., 2007].

3.1 The Java state

The rewrite theory $\mathcal{R}_{\text{Java}}$ is defined on terms of a concrete sort JavaState, with the main state attributes (represented by means of constructor symbols of the algebraic type JavaState). Each state attribute has a sort and uninterpreted operations, and the state itself is a multiset of its attributes. The state infrastructure is a two-level structure as shown in Figure 3.1, which distinguishes the global state (the circle node upside in the figure) from the local state of each thread (the middle circle node) because it was designed to allow multiple threads. The global state of a Java program has nine attributes (their sorts are the upper oval nodes) wrapped by the uninterpreted operations (the arc labels in the figure from left to right): 1) `out` for the output values list, 2) `in` for the input values list, 3) `t` for each thread local state, 4) the `store` for assignments of memory locations to their actual values, 5) `code` for the wrapped Java code, 6) `static` for storing all static variables, 7) `busy` for the locks used by all threads, 8) `nextLoc` for the last allocated memory location identifier, and 9) `nextTid` for the next thread identifier available. This state infrastructure allows one to identify a state component by just mentioning the

name of the corresponding attribute. This fact contributes to the modularity of the Java language definition.

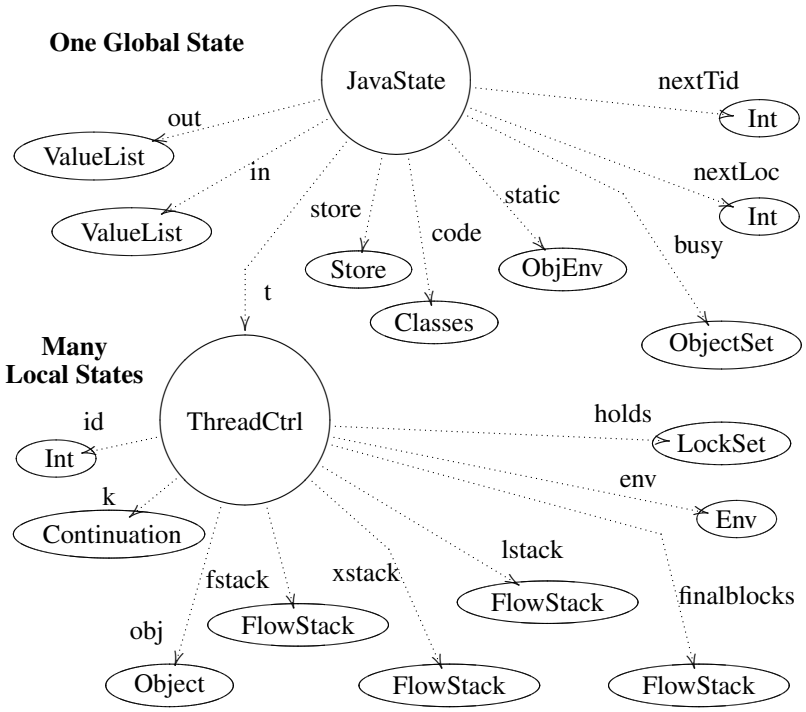

Figure 3.1: Java program state.

The local state thread control attribute (sort ThreadCtrl) is also a multiset and has nine parts or subattributes for each active thread, whose uninterpreted operations are (the arc labels in Figure 3.1 from left to right): 1) `id` is the unique thread identifier, 2) `k` is the thread continuation for handling control flow, 3) `obj` is the active object corresponding to the thread, 4) `fstack` is the function stack that is used for handling function calls, 5) `xstack` is the exceptions stack, 6) `lstack` is the loop stack that is used for loop handling, 7) `finalblocks` is the final block stack, 8) `env` is the environment that is used for allocation of non-static variables to memory locations and 9) `holds` keeps the locks held by the thread.

These functions define an algebraic structure which is parametric on a generic sort Value that defines all the possible values returned by Java functions, or stored in the memory, etc. For instance, the `int` and `bool` constructors respectively describe Java, integer and boolean values, and are defined in

Maude as "`op int : Int -> Value .`" and "`op bool : Bool -> Value .`", where `Int` and `Bool` are the internal built–in Maude sorts that define integers and booleans. Intuitively, equations in Δ_{Java} and rules in R_{Java} are used to specify the changes to the program state, i.e. the changes to the memory, threads, input/output, etc. The semantics of Java is defined modularly, i.e. different features of the language are defined in separate Maude modules so to ease extensions and maintenance; see [Farzan et al., 2007] for further details.

Since we consider only deterministic Java programs in Chapters 5, 6, and 7, our specification of the Java semantics in rewriting logic contains only equations and no rules, and the program state has only one thread (one local state), no locks are used, and no exceptions are thrown, as shown in Figure 3.2. The reader can find a RWL specification of the semantics of a programming language with threads in [Meseguer and Roşu, 2007; Farzan et al., 2007].

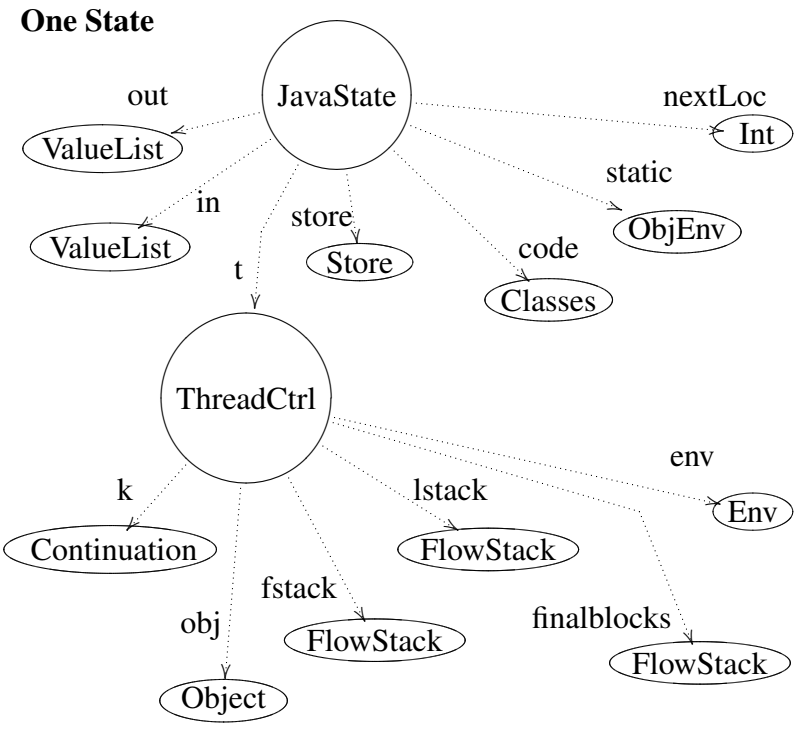

Figure 3.2: Sequential Java program state.

```
--- First evaluate arguments
eq k((E + E') -> K) = k((E, E') -> (+ -> K)) .
--- Once arguments are evaluated to integers, compute addition
eq k((int(I), int(I')) -> (+ -> K)) = k(int(I + I') -> K) .
```

Figure 3.3: Continuation-based equations for Java addition operator on integers.

3.2 Continuation-based semantics

The semantics of Java is defined in a *continuation-based style*. Continuations maintain the control context which explicitly specifies the next steps to be performed. Continuations are a typical technique to transform the uncontrollable control context into controllable data context, by stacking the sequence of actions that still need to be executed. Once the expression e on the top of a continuation (e -> k) is evaluated, its result will be passed to the remaining continuation k. Continuations significantly ease the definition of flow-control instructions, such as break, continue, return, and exceptions.

For instance, the Java addition operation on Java integers is specified[1] in Figure 3.3 using continuations, where `k` is the constructor symbol used to denote a continuation in a thread, -> is the constructor symbol used to concatenate continuations, `int` is the constructor symbol used to denote a Java integer, and + with[2] arity 2 and inside the constructor `int` is the Maude addition symbol, whereas + with arity 2 but outside the constructor `int` is the Java addition symbol, and + with arity 0 is a continuation symbol used to remember that the Java addition action is being stacked. For instance, consider the evaluation of the Java expression "`2 + 3`"; the constant literals "2" and "3" of the Java program are preprocessed into the Maude terms "`i(2)`" and "`i(3)`", respectively; these Maude terms are evaluated, respectively, to the values "`int(2)`" and "`int(3)`". The equational rewriting sequence (reduction) corresponding to the evaluation of the expression is given in Figure 3.4. The Java less-or-equal boolean operation on Java integers is specified in a similar way in Figure 3.5.

[1]The Maude syntax is almost self-explanatory. The general point is that each item: a sort, a subsort, an operation, an equation,etc., is declared with an obvious keyword: `sort`, `subsort`, `op`, `eq`, etc., with each declaration ended by a space and a period. We use uppercase letters to denote Maude variables and lowercase letters to denote Maude constructor symbols. See [Clavel et al., 2007] for details.

[2]The Maude syntax allows overloading of operators, with different arities.

```
k((i(2) + i(3)) -> K)
====> k((i(2), I(3)) -> (+ -> K))
====> k(i(2) -> ([ i(3), noExp | noVal ] -> (+ -> K)))
====> k(int(2) -> ([ i(3), noExp | noVal ] -> (+ -> K)))
====> k( i(3) -> ([ noExp | int(2) ] -> (+ -> K)))
====> k( int(3) -> ([ noExp | int(2) ] -> (+ -> K)))
====> k( ( int(2),int(3)) -> (+ -> K)))
====> k(int(2 + 3) -> K)
====> k(int(5) -> K)
```

Figure 3.4: Evaluation of ”2+3” Java expression.

```
--- First evaluate arguments
eq k((E <= E') -> K) = k((E, E') -> (<= -> K)) .
--- Once arguments are evaluated to integers, compute boolean
eq k((int(I), int(I')) -> (<= -> K)) = k(bool(I <= I') -> K) .
```

Figure 3.5: Continuation-based equations for Java less-or-equal operator on integers.

```
--- No new variable, end buildEnv continuation
eq k(buildEnv(noParameters, noValues) -> K) = k(K) .
--- New variable with name Var and value Val is assigned to Location I' + 1
eq t(k(buildEnv(((T d(Var)), Pl), (Val, Vl)) -> K) env(Env) TC) store(ST) nextLoc(I')
 = t(k(buildEnv(Pl, Vl) -> K) env([Var, l(I' + 1)] Env) TC)
   store([l(I' + 1), Val] ST) nextLoc(I' + 1) .
```

Figure 3.6: Continuation-based equations for building the environment.

```
---First obtain location in store from variable name
eq k(Var -> K) env([Var, Loc] Env) = k(#(Loc) -> K) env([Var, Loc] Env) .
---Then obtain value stored in this location
eq k(#(Loc) -> K) store([Loc,Value] Store) = k(Value -> K) store([Loc,Value] Store) .
```

Figure 3.7: Continuation-based equations for variable content retrieval.

A relevant construction in the Java semantics is the `buildEnv` continuation symbol shown in Figure 3.6 that gives a new location in the memory store to each new variable. It involves the following four elements of the Java state: the unique thread adding new variables (denoted by constructor `t`), the local environment of the thread (denoted by constructor `env`), the global store (denoted by constructor `store`), and a counter for the last used location in the store (denoted by constructor `nextLoc`).

Another important aspect of the semantics is the handling of Java variables. In Figure 3.7, we show how the contents of a Java variable are retrieved from the store (or memory) in the Java state. The semantics of the assignment operator for the Java variables is specified in Figure 3.8.

The semantics of the if-then-else statement is shown in Figure 3.9. The

```
---Obtain variable location and evaluate expression
eq k(Var = E -> K) env([Var, Loc] Env) = k(E -> =(Loc) -> K) env([Var, Loc] Env) .
---Once the expression is computed, assign to location
eq k(Val -> =(Loc) -> K) = k([Val -> Loc] -> (Val -> K)) .
---General procedure to update the memory
eq k([Val -> Loc] -> K) store([Loc,Val'] ST) = k(K) store([Loc,Val] ST) .
```

Figure 3.8: Continuation-based equations for the Java assignment operator.

```
--- Evaluates boolean expression keeping the then and else statements
eq k((if E S else S') -> K) = k(E -> (if(S, S') -> K)) .
eq k(bool(true) -> (if(S, S') -> K)) = k(S -> K) .
eq k(bool(false) -> (if(S, S') -> K)) = k(S' -> K) .
```

Figure 3.9: Continuation-based equations for the if-then-else statement.

```
--- Stack loop and transform while expression into while continuation
eq k((while E S) -> K) lstack(Lstack)
 = k(while(E,S) -> popLStack -> K) lstack(while(E,S) -> K, Lstack) .
--- A while continuation is transformed into an if-then-else
eq k(while(E,S) -> K) = k(E -> if(S while ( E , S ),{}) -> K) .
--- Add semantics for popLStack
eq k(popLStack -> K) lstack(LItem,Lstack) = k(K) lstack(Lstack) .
```

Figure 3.10: Continuation-based equations for the while statement.

```
--- The state is restored from the loop stack
eq k(break -> K) lstack(while(E,S) -> K', Lstack) = k(K') lstack(Lstack) .
```

Figure 3.11: Continuation-based equations for the while-break statement.

semantics of while statements (loops) is specified in Figure 3.10, where the term `while E S` denotes the Java iteration statement, the term `while(E,S)` denotes both the while continuation and the while statement that is expressed in terms of the `if(S,S')` continuation, and `lstack` denotes a stack of loops currently being executed, which is needed for a proper control of the Java `break` statement.

Figure 3.11 shows the semantic specification of the `break` statement, that simply pops the stack of loops. This is important for Chapters 6 and 7 since it can also abruptly change the information flow.

Instance method calls are shown in Figure 3.12. Their semantics is simply defined by the eager evaluation of all arguments of the method, the search for the corresponding method code, the creation of a new local empty environment, the creation of a new empty loop stack, and the replacement of the active object while keeping the local state in the function stack `fstack` (with the exception of the local state function stack). This local state will be re-

```
--- Evaluates target object  expression, while keeping
--- the method and the arguments expressions
eq k((E . (M < El >)) -> K) = k(E -> ((M < El >) -> K)) .
--- Evaluates arguments expressions keeping target object and method
eq k(Obj -> ((M < El >) -> K)) = k(El -> (call(Obj, M) -> K)) .
--- Look for the method while keeping the arguments values
eq t(k(Vl -> (call(o(orig(C') Oattr), M) -> K)) TC) code(Cl)
 = t(k(findMethod(C', M, getTypes(Vl), Cl) -> (call(o(orig(C') Oattr), M, Vl) ->
     K)) TC) code(Cl) .
--- Keep current local state in function stack, create an
--- empty new one, and create local environment.
eq t(k(m(Md, Pl, block, T) -> (call(o(curr(T') Oattr), M, Vl) -> K)) obj(Obj')
      fstack(Fstack) lstack(Lstack) env(Env) TC)
   = t(k(buildEnv(Pl, Vl) -> (block -> (return; -> noop))) obj(o(curr(T) Oattr))
     fstack(fsi(K, (obj(Obj') lstack(Lstack) env(Env) TC)) Fstack) lstack(noItem)
     env(noEnv) TC) .
```

Figure 3.12: Continuation-based equations for the instance method call statement.

```
--- If there is a value to be returned, first evaluate expression
eq k(return E ; -> K) = k(E -> return -> K) .
--- Release local environment, restore local state of calling method,
---  and keep returned value on continuation
eq t(k(V -> return -> K) fstack(fsi(K', ( env(Env) TC)) Fstack)  env(Env') TC')
   = t(k(releaseEnv(Env') ->  (V -> K')) fstack(Fstack) env(Env) TC) .
--- If there is no value to be returned, release local environment,
--- and restore local state of calling method.
eq t(k(return; -> K) fstack(fsi(K', (env(Env) TC)) Fstack) env(Env') TC')
   = t(k(releaseEnv(Env') ->  K') fstack(Fstack) env(Env) TC) [owise] .
```

Figure 3.13: Continuation-based equations for the return statement.

stored when the method returns (Figure 3.13). The first action to be executed before transferring control to the block of the called method is to build the local method environment with the formal parameters and their corresponding argument values.

Figure 3.13 shows the specification of "`return E`" and "`return`" statements. The specification of "`return E`" statement includes releasing the local environment, restoring the local state of the calling method, and keeping the returned value on top of the continuation.

3.3 Java execution

In order to interpret a Java program with this Java rewriting logic based operational semantics, the Java program has to be compiled into a Maude expression by using a Java language processor that applies a congruent transformation. We use the Java wrapper available at http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java.

In the following, we show how a Java program is transformed into a Maude term describing the initial configuration of the Java semantics corresponding to Figure 3.2. We also partially show the interpretation of a Java function call. A complete Java code and the corresponding Maude term that was produced by using the Java wrapper are illustrated as follows:

```
class Safe1Even3p1 {
        public  static void main(String[] args) {
               TestClass t;
               t = new TestClass();
               System.out.println(t.summation(0));    }
}
class TestClass {
 int summation(int n) {
   int sum = 0 ;
   int i = 0;
   while (i<=n)       {
            sum += i;
            i++;         }
   return sum;  }
}
```

The Maude term compiled from the Java source has two parts: i) some pre-processing code that is performed in order to remove the `import` Java compiler directives and to congruently transform the types of program classes, and, ii) the expression with the invocation to the `main` method that will initiate the program execution, after state initialization by the `java` operator. In this example, the `main` method invocation has no parameter (`noVal`).

```
java((
 preprocess((default class t('Safe1Even3p1) imports nil extends Object implements none {

  (public static) void 'main(t('String)[] d('args)) throws(noType) {
   (((((   t('TestClass) d('t) ;)
        5 @ ('t = new t('TestClass) < noExp > ;))
        6 @ ('System . 'out . 'println  < 't . summation < i(0) > > ;)) }

  default class t('TestClass) imports nil extends Object implements none {
   default int 'summation(int d('n))throws(noType)
    {((  (int d('sum) = i(0) ;)
```

```
        (int d('i) = i(0) ;))
         24 @ (while 'i <= 'n 24 @ {
            (22 @ ('sum += 'i ;))
             23 @ ('i ++ ;)}))
         25 @ return 'sum ;}}))

  noType . 'main < new string [i(0)] > noVal
    ))
```

From this initial Maude term, the `java` operator creates the corresponding initial program state, as specified in the equations below, that include: i) the initial state of the continuation with three actions to be executed in the given order: the building of the static environment as specified in the Java pre-processed code (`Cl`), then the invocation of the `main` method, and, finally, the stop action; ii) the initial empty state with no active object: the function stack, the exception stack, the loop stack, the environment, the set of locks used, the output, and the store are all empty; iii) the pre-processed code, the static environment (the static classes and members), and the given input value list; and iv) the next available store location identifier and the next available thread identifier. The default thread has the identifier 0.

```
eq java((Cl E Vl)) = run(t(k(buildS(Cl) -> (E -> stop)) obj(nullo) fstack(noItem)
  xstack(noItem) lstack(noItem) finalblocks(noItem) env(noEnv) id(-1) holds(nil))
  out(noOutput) in(Vl) store(noStore) code(Cl) static(onil) busy(noObj)
  nextLoc(0) nextTid(1)) .
```

Below, we show part of the trace that corresponds to the `t.summation(0)` method invocation.

The trace shows, first, the retrieval of the store location and the value of variable `t` (i.e. the object whose method is called), next the evaluation of the parameter `i(0)` (a Java integer literal), and finally the search for the method code (omitting some intermediate steps):

```
k('t . 'summation < i(0) > -> K)
===> k('t -> 'summation < i(0) > -> K)
===> k(#(l(2))  -> 'summation < i(0) > -> )
===> k(o(f([t(t('TestClass)),f(noEnv)]) curr(t('TestClass))
        orig(t('TestClass)))  -> 'summation < i(0) > -> K)
===> k(i(0) -> call(o(f([t(t('TestClass)),f(noEnv)])
       curr(t('TestClass)) orig(t('TestClass))), 'summation) -> K)
===> k(int(0)  -> call(o(f([t(t('TestClass)),f(noEnv)])
       curr(t('TestClass)) orig(t('TestClass))), 'summation) -> K)
===> k(findMethod(t('TestClass), 'summation, getTypes(int(0)), CODE) ->
       call(o(f([t(t('TestClass)),f(noEnv)])
       curr(t('TestClass)) orig(t('TestClass))), 'summation, int(0)) -> K)
===> k(m(default, int d('n), {(int d('sum) = i(0) ;)
```

```
        (int d('i) = i(0) ;)
        (24 @ (while 'i <= 'n 24 @ {
           (22 @ ('sum += 'i ;))
            23 @ ('i ++ ;)}))
            25 @ return 'sum ;}, t('TestClass)) ->
       call(o(f([t(t('TestClass)),f(noEnv)])
       curr(t('TestClass)) orig(t('TestClass))), 'summation, int(0)) -> K)
```

Finally, the trace below shows the building of the local method environment corresponding to the formal parameter n: the keeping of the context of the invocation in the function stack, and the creation and initialization of the variable n:

```
t( id(0)
k(buildEnv(int d('n), int(0)) -> {(int d('sum) = i(0) ;)
  (int d('i) = i(0) ;) (24 @ (while 'i <= 'n 24 @ {
    (22 @ ('sum += 'i ;)) 23 @ ('i ++ ;)})) 25 @ return 'sum ;} -> return; -> noop)
obj(o(f([t(t('TestClass)),f(noEnv)]) curr(t('TestClass)) rig(t('TestClass))))
fstack(fsi(call(o(curr(System) orig(System)), 'println) -> ; -> (( 7 @ ( 'System .
      'out . 'println < 't . 'sum < i(1) > > ;)) ( 8 @ ('System . 'out . 'println <
       't . 'sum < i(2) > > ;)) 9 @ ('System . 'out . 'println < 't . 'sum < i(3) > >
       ;)) -> e(['args,l(1)]) -> stop, id(0) obj(o(f(onil) curr(t( 'Safe1Even3p1))
       orig(t('Safe1Even3p1)))) xstack(noItem) lstack(noItem) finalblocks(noItem)
       env(['args,l(1)] ['t,l(2)]) holds(nil)))
xstack(noItem) lstack(noItem) finalblocks(noItem) env(noEnv) holds(nil))

store([l(1),a( string, anil),0] [l(2),o(f([t(t('TestClass)),f(noEnv)])  curr(
      t('TestClass)) orig(t('TestClass))),0]) static([t(t('Safe1Even3p1)),f(noEnv)]
      [t(t( 'TestClass)), f(noEnv)])
busy(noObj) nextLoc(2) nextTid(1)
===>
 t(
 k(buildEnv(noPara, noVal) -> {(int d('sum) = i(0) ;) (int d('i) = i(0) ;) (24 @
  (while 'i <= 'n 24 @ { (22 @ ('sum += 'i ;)) 23 @ ('i ++ ;)})) 25  @ return 'sum ;}
   -> return; -> noop)
 env(noEnv ['n,l(2  + 1)]) (obj(o(f([t(t('TestClass)), f(noEnv)]) curr( t('TestClass))
     orig(t('TestClass))))
 fstack(fsi(call(o(curr(System) orig(System)), 'println) -> ; -> ((7 @ ('System . 'out .
  'println < 't . 'sum < i(1) > > ;)) (8 @ ( 'System . 'out . 'println < 't . 'sum  <
  i(2) > > ;)) 9 @ ('System . 'out . 'println < 't . 'sum < i(3) > > ;))  -> e(['args,
  l(1)]) -> stop, id(0) obj( o(f(onil) curr(t('Safe1Even3p1)) orig(t('Safe1Even3p1))))
  xstack(noItem) lstack(noItem) finalblocks(noItem) env(['args,l(1)] ['t,l(2)])
  holds(nil)))
 xstack(noItem) lstack(noItem) finalblocks(noItem) holds(nil)) id(0))

 store(([l(1),a(string, anil),0] [l(2),o(f([t(t('TestClass)),f(noEnv)]) curr(
       t('TestClass)) orig(t('TestClass))),0])  [l(2 + 1),setTid(int(0), 0),0])
 nextLoc(2 + 1)
 code()
 static([t(t('Safe1Even3p1)),f(noEnv)] [t(t('TestClass)),  f(noEnv)]) busy(noObj)
 nextTid(1))
```

Now, we are ready to introduce the main aspects of the certification technique in Chapter 4, as follows.

Chapter 4

Certifying Java programs

The certification methodology formalized in this thesis is essentially as follows. Consider a source Java program together with a specification of the Java semantics written in Maude, as described in Chapter 3. The Java program is a concrete expression (i. e., a term) that represents the initial state of the Java interpreter running the considered Java program. Given a safety property (i.e., a system property that is defined in terms of certain events that do not happen [Manna and Pnueli, 1995]), the unreachability of the system states that denote the events that should never occur from the considered initial state allows one to infer the desired safety property. Unreachability analysis is performed by using the standard Maude (breadth–first) search command, which explores the entire state space of the program from an initial system state.

In the case where the unreachability test succeeds, the corresponding rewriting proofs that demonstrate that those states cannot be reached are delivered as the expected outcome certificate. Very often, the unreachability test does not succeed because there is an infinite search space; thus, to achieve a finite search space, abstraction is used [Cousot and Cousot, 1977].

The safety certificate essentially consists of the set of rewriting proofs of the form $t_1 \rightarrow^{r_1}_{\text{Java}^\#} t_2 \cdots \rightarrow^{r_{k-1}}_{\text{Java}^\#} t_k$ that describe the abstract program states which can and cannot be reached from a given a (abstract) initial state. The certificates, i.e. the encoded (abstract) rewriting sequences, together with an encoding of the abstraction in Maude, can be checked by standard reduction. This methodology is an instance of the Proof–Carrying Code (PCC) paradigm, a mechanism originated by Necula [Necula, 1997] for ensuring the secure behavior of programs.

Since these proofs correspond to the execution of the abstract Java semantics specification, which is made available to the code consumer, the certificate can be inexpensively checked on the consumer side by any standard rewrite engine by means of a rewriting process that can be very simplified. Actually, it suffices to check that each abstract rewriting step in the certificate is valid and that no other valid rewritings have been disregarded, which essentially

amounts to use the matching infrastructure within the rewriting engine.

4.1 The Java Modeling Language JML

The safety property is specified by using a subset of the Java Modeling Language JML syntax [Leavens et al., 2006]. The Java code is annotated with the JML-like safety specifications that are pre-processed and compiled to a Maude term that encodes the program initial state.

The Java Modeling Language [Leavens et al., 2006] is a behavioral and interface specification language that allows Java programmers to write specifications of Java classes, interfaces, and modules without the difficulty of learning a language-independent formal specification language like OCL (Object Constraint Language) (see c. f. [Burdy et al., 2005]). JML has been designed as an easily accessible specification language that combines the design by contract method and the model-based approach to specification in order to guarantee that a program satisfies its specification. Java developers can specify within JML the functional properties of their programs in a generalization of Hoare logic, tailored to Java. JML uses Java's expression syntax in assertions, with some specific keywords added for specification purposes of the Java modules interfaces and behavior, so that the JML notation is easy to learn for Java programmers.

As an interface specification language, JML can describe the names and static information found in Java declarations of Java modules with pre-conditions (in `requires` clauses), normal post-conditions (in `ensures` clauses), invariants and exceptional post-conditions (with the `signals` clauses), that express first-order logic statements (Figure 4.1). JML notation includes quantifiers `\forall` and `\exists` and specification-only fields and methods that allow more precise and complete specifications.

```
/*@  requires <precondition>;
  @  ensures <postcondition if no exception raised>;
  @  signals(E) <postcondition when exception E raised>;
  @  assignable <modifiable fields and variables>      @*/
```

Figure 4.1: JML method specification.

As a behavior specification language, JML can also describe how the module will behave when used with assertions intermixed with the Java code, us-

ing `assert` statements (Figure 4.2). These assertions can be checked at run time.

```
//@  assert <expression>;
```

Figure 4.2: JML assert statement.

JML comes with a library with Java types that can be used for mathematically describing the behavior of sets, sequences and relations.

JML annotations may include class, field and method definitions that are only used for specification purposes, i.e. they are not used in executable Java code. These JML annotations include the modifier `model` within fields, methods, and class declarations [Leavens et al., 2008]. For instance, a `model` method is a method that can be invoked in JML annotations, but cannot be invoked in executable Java code. Example 4.1, borrowed from [Leavens et al., 2006], shows a class with a model field.

```
public abstract class  UnBoundedStack  {
/*@  public model JMLObjectSequence theStack;
  @  public initially theStack != null && theStack.isEmpty();
  @*/
  ...
}
```

Figure 4.3: JML specification clauses for a class with a model field.

The JML specifications of a Java program can either be written as code annotations in Java program files or in separate files. The JML specifications given as code annotations are treated like Java comments that are ignored by the compiler. The text of an annotation could be either in one line, after the marker `//@` or, in many lines enclosed between the markers `/*@` and `@*/`.

4.1.1 JML tools

The correctness of JML specifications can be currently verified either during runtime or statically. The most basic static tool support for JML is the JML checker (`jmlc`) that performs type checking and parsing of Java programs with their JML annotations (see [Burdy et al., 2005]). There are several tools for static verification of Java programs that use JML as a specification

language. The main differences between these tools are its soundness, its level of automation, its language coverage and whether they are proof tools or just validation tools.

The ESC/Java tool [Flanagan et al., 2002] offers a higher level of automation without any user interaction and relies on an automatic prover to check null pointers or array bound limits but it is unsound and incomplete and uses its own specification language. The ESC/Java2 tool [Chalin et al., 2005] extends ESC/Java to support Java 1.4 code with standard JML annotations, but it is also unsound and incomplete. However, the ESC/Java2 tool cannot generate checkable proofs when verification succeeds.

JML4, an Integrated Verification Environment (IVE) for Java [Chalin et al., 2007; Chalin et al., 2008], is the first IVE that supports a full range of verification technologies for a mainstream programming language. JML4 integrates a non-null type system for JML, a run–time assertion checking component, an Extended Static Checking (ESC) tool and a Full Static Program Verification (FSPV). JML4 is built upon the Eclipse's Java Development Tooling (JDT) and has up-to-date support for Java. The JML4 tool has found bugs that were not previously detected by `ESC/Java2` because the size of the analyzed program is so big that the corresponding verification condition is too large for `ESC/Java2`.

The ESC component of JML4, called ESC4 can also verify more kinds of methods and is generally more efficient than its predecessor. It can also be used in off-line user-assisted mode, a new form of verification that lies between ESC and Full Static Program Verification (FSPV) [Janota et al., 2007]. ESC4 produces proof obligations for the first order provers Simplify and CVC3 as well as Isabelle/HOL. The JML4 FSPV tool, called the FSPV Theory Generator (TG), generates theories written in the Hoare Logic of SIMPL which is an Isabelle/HOL based theory designed for the verification of imperative sequential programs. To prove the correctness of such theories, a user can interactively get their proof using the Eclipse version of Proof General (PG). The proof can then be validated by Isabelle [Chalin et al., 2008].

Other static and proof tools that are sound but require user interaction and are targeted for the Java Card language (a sequential subset of the Java language) are Krakatoa [Marché et al., 2004], JACK (the Java Applet Correctness Kit) [Barthe et al., 2007a], Jive (Java Interactive Verification Environment) [Meyer and Poetzsch-Heffter, 2000; Ádám Darvas and Müller, 2007], KeY [Ahrendt et al., 2007; Beckert et al., 2007] and LOOP [van den Berg

and Jacobs, 2001].

These proof tools generate proof obligations that can be discharged by using automatic provers or proof assistants in which the Java semantics is specified: LOOP generates proof obligations for PVS [Owre, 2007] and Isabelle. Jive for PVS, Isabelle/HOL and Simplify [Dowek et al., 2001; Ádám Darvas and Müller, 2007]. Krakatoa uses the Why tool, which can deliver a multi-prover output, to generate the proof obligations for Coq, Simplify, Alt-Ergo, CVC3, PVS, Isabelle/HOL, etc. See [Filliâtre and Marché, 2007] for more information. JACK can interface with several theorem provers: AtelierB, Simplify, Coq and PVS [Burdy and Pavlova, 2006].

JACK considers only Java sequential programs without dynamic loading [Marché et al., 2004]. The JACK tool is integrated with Eclipse [eclipse, 2011] and provides an easy-to-use user interface. Moreover, it works both for source code and for bytecode, but JACK verification takes place at the source code level. JACK implements a weakest precondition calculus adapted for Java, and automatically generates proof obligations that can be discharged both by automatic and interactive theorem provers. JACK generates the proof obligations in first-order logic by using an abstract formula language.

Krakatoa is a translator that reads Java files annotated with pre and post-conditions, produces input specifications for the considered theorem provers and a representation of the Java program to the Why tool. The Why tool produces proof obligations based on the code and the pre and post-conditions. Krakatoa takes into account a subset of JML, w.r.t. clause `invariant` for classes, clauses `requires, assignable, ensures` and `signals` for methods, and loop invariants and `decreases` clauses for `while` and `for` loops [Marché et al., 2004]. Besides correctness, Krakatoa can prove loop termination, but not recursion termination. The Krakatoa tool was improved to consider Java card transactions, Java floating point arithmetic, pointer aliasing analysis based on separation logic, support for SMT provers and the Caduceus extension for the analysis of `C` programs [Filliâtre and Marché, 2007].

The JACK and Krakatoa Coq proofs can be checked by type-checking the generated Coq files by using the Coq small certification Kernel. This means that Krakatoa and JACK could be used within a PCC framework, where the tools and the Coq prover can be used by the code producers and the Coq type-checker can be used by the code consumers.

Jive works with a sequential subset of Java [Poetzsch-Heffter and Müller, 2011]. The Jive tool uses the ANJA (annotated Java) specification language.

Jive has its own program prover component that proves program-dependent properties, while program-independent proof obligations are solved by using a theorem prover, i.e. PVS [Meyer and Poetzsch-Heffter, 2000]. These proof obligations include type-checking and non-Hoare formulas. The program prover component implements a Hoare logic by basic proof operations supporting forward and backward proving. Proof tactics can be defined by using these proof operations. Jive also supports most JML level 0 constructs and can generate program-independent proof obligations for the Isabelle/HOL and Simplify theorem provers [Ádám Darvas and Müller, 2007]. However, PVS, Isabelle/HOL and Simplify produce proofs that cannot be independently checked.

The KeY tool can be used for the formal specification and semi-automatic verification of Java Card programs (programs with no floats, concurrency nor dynamic class loading) [Ahrendt et al., 2007]. KeY uses its own deductive verification prover, which also can be used as a stand-alone prover. KeY is based on a free-variable sequent calculus for a first-order dynamic logic for Java [Ahrendt et al., 2007]. KeY supports the OCL and JML specification languages and is integrated both in Eclipse and Together [Together, 2011] IDEs. However, KeY cannot generate proofs that could be checked without relying on the KeY prover itself.

The LOOP compiler can be used in the verification of sequential Java programs. LOOP reasoning combines the application of semantic-based Hoare logic and automatic rewriting. LOOP compiler uses Higher Order logic in order to translate the Java code and the JML specifications into its semantics. This semantics is a series of logic theories that are specified by using an abstract syntax. The abstract syntax theories are used to generate concrete syntax theories for different theorem provers, one for PVS and one for Isabelle [van den Berg and Jacobs, 2001].

There are some JML-based verification tools for Java that are based on model checking instead of theorem proving. For instance, Bogor, which is an extensible framework designed to support both general purpose and domain-specific software model checking. Bogor can model-check standard complete heavy-weight JML specifications of sequential and concurrent Java programs [Robby et al., 2006]. Bogor implements explicit-state model checking through use of sophisticated state-space reductions. Bogor reduces the space required to store a state by sharing common parts of distinct states. Besides, Bogor reduces the set of paths that need to be explored in the state-space by

using partial order reductions. The combination of these techniques in Bogor yields orders of magnitude of reduction in the space and time required for model checking. However, Bogor does not generate any proof when verification succeeds. Bogor is a validation tool, but not a proof tool, and it cannot be used in a PCC framework.

One of the most popular certification approaches that apply to Java or JVM is based on "types and effects" inference systems [Necula and Lee, 1996; Necula and Rahul, 2001]. As we already mentioned, our certification methodology is different in its rewriting logic nature to all the above methods.

4.2 Full certificates

Our methodology can generate full certificates that include the initial program state and all performed rewriting steps, showing for each of the rewriting steps, the equation or rule used, the subterm to be rewritten (i.e. the redex), the corresponding matching substitution, and the rewritten term. The full certificate may include also the full term that represents the entire program state before and after each rewriting step. These full certificates contain all possible program traces, such that full certificate sizes grow depending on program size and cyclomatic complexity, i.e. the number of program-independent execution paths. However, the full certificates do not grow exponentially as a function of the code size (see numeral 2 on page 6), and it could be reduced without decreasing its information content, by eliminating its redundant content similarly to the LFi reduced version of the LF type derivations [Necula and Lee, 1997a] (see last paragraph of Section 1.1.1 on page 8). For instance, the program code in the Java program state does not change at all, i.e. it is the same in all program states. The left-hand side of Figure 8.3 (on page 158) has a screen snapshot that shows part of a full certificate.

Not only the size of full certificate is higher than the size of the corresponding reduced one. The corresponding full certificate generation time is also higher than the corresponding reduced certificate generation time because the trace option must be set on, slowing Maude's execution. On the other hand, full certificate checking time is smaller. Actually the time needed to check the rewriting sequence is *minimal*, because it is enough to check if the redex is a subterm of the current program state, if the redex matches the left hand side of the applied rule or equation using the given matching

substitution, and if the rewritten term corresponds to a rewriting step using the applied rule or equation and the corresponding substitution. In this case, for each given program state it suffices to apply pattern matching, and then one reduction step, by using the corresponding given equation or rule in the certificate.

In order to validate if there is some rewriting sequence that was disregarded and it is not in the full certificate, we need to check if there is an equation or a rule that could be applied and it is not. We would like to emphasize that, in this case pattern matching is also enough.

4.3 Reduced certificates

We have considered two kinds of reduced certificates that only include the rewrite steps corresponding to the applied rules, while excluding the used equations. In this case, the Maude trace option is set off, taking advantage of the efficient handling of equations by Maude:

- Reduced-rule certificate, that includes the used rules, the redexes, the corresponding matching substitutions, and the program states before and after rule application. The right-hand side of Figure 8.3 (on page 158) shows part of a reduced-rule certificate.

- Reduced-rule label certificate, that only includes the labels of the used rules, while omitting the used rules, the corresponding matching substitutions and the program states.

These reduced certificates contain less information than the full certificates, since they omit most rewriting steps (the rewriting steps that correspond to equational rewriting) of the rewriting logic deduction. In this way, the reduced certificates can be seen as the proof witnesses of FPCC [Appel et al., 2003; Wu et al., 2003] and the oracles of OPCC [Necula, 2001a; Necula and Rahul, 2001], but our reduced-rule certificates size (length of the text string) is ten times the program size (bytes) while the size of the proof witnesses of FPCC can be thousand times the program size (see FPCC section on page 10). Moreover, the reduced-label certificates size is unit times the program size. However, the reduced (rule and label) certificates have higher relative size than OPCC oracles which are 12% the size of the code (see OPCC section on page 14).

Generation time and size of any reduced-rule certificate is smaller than the generation time and size of the corresponding full certificate, because Maude does not record the rewriting steps that correspond to equation application, neither the corresponding redexes, matching substitutions, and terms that correspond to intermediate program states.

On the other hand, the time needed to check reduced-rule certificates is higher than the checking time of full certificates because it is necessary to reduce the terms that represent program states by using the equations before considering a rewriting step using any given rule, without no information on the equations used and the corresponding substitutions.

This means that standard reduction is enough to validate the reduced-rule certificate. Moreover, in order to check in this case if there are no equations and rules that are disregarded we also need standard reduction.

The size of the reduced-rule label certificate is the smallest one but its validation time is the highest, because the certificate only includes the initial program state, the final program states and the labels of the applied rules, but no corresponding substitutions and no intermediate program states are delivered.

The higher validation times of reduced certificates are similar to the FPCC with witnesses, OPCC and ACC approaches. The use of witnesses in the FPCC approach reduces the certificate sizes but increases the certificate validation times; the validation time grows slower than the code size [Wu, 2005] (see FPCC section on page 11). In OPCC, the validation times of the oracles are higher than the validation times of the full LFi certificates (see OPCC section on page 14); while in ACC, the validation time of the reduced certificates are 6% higher than the validation time of the full corresponding certificates [Albert et al., 2011](see ACC section on page 19).

In order to validate the reduced-rule certificates, we also need standard reduction.

The experiments reported in Sections 5.3, 6.5 and 7.5, include a comparison among full certificates, reduced-rule certificates and reduced-rule label certificates, regarding arithmetic integer properties, non–interference policies and erasure policies, respectively.

Other versions of reduced certificates could be obtained by means of automatic transformation techniques such as trace slicing. Trace slicing is a widely used technique for execution trace analysis that is effectively used in program debugging, analysis and comprehension. In [Alpuente et al., 2011],

a backward trace slicing technique is proposed that can be used for the analysis of rewriting logic theories. The trace slicing technique allows one to systematically trace back rewrite sequences modulo equational axioms (such as associativity and commutativity) by means of an algorithm that dynamically simplifies the traces by detecting control and data dependencies, and dropping useless data that do not influence the final result. The methodology is particularly suitable for analyzing complex, textually-large system computations such as those delivered as counter-example traces by Maude model-checkers. Trace slicing could be used to reduce the trace-based certificates generated by our methodology at the code producer side, but also it can be used to check them at the code consumer side. This is immediate thanks to availability of the trace slicing facility for a wide class of rewrite logic theories.

4.4 Certificate checking vs. generation

In this section, we demonstrate how the code consumer can check that the proof carried by the program is valid, so that the consumer can execute the code. We also include some experiments that demonstrate that validation at the consumer side is lighter than the proof generation effort at the code producer side.

As explained above, certificate validation is faster than certification generation basically because the certificate contains information that avoids to look for the equations and rules that could be applied. In the case of the full certificates, they have the smallest validation times because their checking only requires matching, whereas the reduced-rule certificate and the reduced-rule label certificate require standard reduction, i.e. we have the information about the applied rules (at least their labels), but we have no information about applied equations, and thus, we have to look for the equations that could be applied.

Therefore, the main difference between the validation time of full and reduced certificates is based on the difference between matching and standard reduction. This means that the generation of the program state space during the verification process at the producer's side requires more time than state space exploration during the validation process at the consumer's side.

In order to analyze proof checking (certificate validation) time at the consumer side, we have developed an experiment involving state space explo-

ration that allows us to find out the relative difference introduced by state condition evaluation given a number of states. The experiment consists of a Maude program that generates a very large search space that is then traversed in different ways: one corresponding to proof generation, another corresponding to proof verification (what the code consumer would be performing). The Maude program generating the large search space is as follows:

```
mod TEST-PCC is
  protecting NAT .
  sort Conf .
  op [_|_] : Nat Nat -> Conf .
  vars N X Y Max : Nat .
  crl [inc] : [ Max | N ] => [ Max | N + 1 ] if N < Max .
  crl [dec] : [ Max | N ] => [ Max | sd(N,1) ] if N > 0 .
endm
```

In this specification[1], there is an upper bound and a counter stored in a configuration of the form "`[ Upper | Counter ]`". There are also two transition rules, one incrementing the counter without reaching the upper bound and another decrementing it without reaching the natural number zero.

First, we consider a search command within this specification that forces two different ways of generating the search space: just generate all states, or generate all states while also checking some extra condition on each generated state. Figure 4.4 shows a table with the results for upper bounds 10000 and 100000, which also implies that there are 10000 and 100000 states in the search space, respectively. The column "Generate All" corresponds to the search command:

```
search [ 10000 | 0 ] =>* [10000 | 10000] .
```

Whereas the column "All & Condition" corresponds to the search command:

```
search [ 10000 | 0 ] =>* [10000 | N]  such that N > 9999 .
```

We can easily see that the time associated to generating the search space and to generating the search space while checking some property is remarkable. Indeed, traversing the state space takes just 15% extra time than the time necessary to generate the search space.

Furthermore, we have defined another experiment where we increase the execution time of the property to be checked throughout the search space. We

[1]The symbol `sd` denotes the subtraction operation

States	Generate All	All & condition	Difference	%
10000	147ms	173ms	26ms	17.69
100000	1585ms	1835ms	250ms	15.77

Figure 4.4: Time differences with a simple condition.

consider the simple condition and a Fibonacci number computation, whose results are shown in Figure 4.5.

The search command used for generating the search space now is:

```
search [ 10000 | 0 ] =>* [10000 | 10000] such that  fib(20) >= 0 .
```

Whereas the search command with the extra condition is:

```
search [ 10000 | 0 ] =>* [10000 | N] such that  N > 9999 /\ fib(20) >= 0 .
```

In this case, the time associated to checking the extra condition"`N > 0`" throughout the search space is very similar to the previous case (15% extra time of the generation time) which is reasonable because the condition is exactly the same though the generation time is bigger due to the condition "`fib(20) >= 0`".

States	All & fib(20)	All & fib(20) & condition	Difference	%
10000	174ms	200ms	26ms	14.94
100000	1631ms	1902ms	271ms	16.62

Figure 4.5: Time differences with a simple condition and a costly computation.

Next, we consider the simple condition and a higher Fibonacci number computation, whose results are shown in Figure 4.6. With this other experiment we can see that the time associated to traversing the search space can be negligible (1%) with respect to generating the states when the generation time increases.

States	All & fib(30)	All & fib(30) & condition	Difference	%
10000	3821ms	3875ms	54ms	1.41
100000	6619ms	6856ms	237ms	3.58

Figure 4.6: Time differences with a simple condition and a more costly computation.

Finally, we can conclude that state generation and state exploration have clearly different costs and that most of the time is devoted to state generation. Also, validation time for reduced-rule and label certificates is higher than full certificates because full certificates contain all the data necessary for the validation.

CHAPTER 5

Analyzing Arithmetic Properties of Java Programs

5.1 Introduction

In this chapter, we consider some simple integer arithmetic properties of methods that accept integer parameters and return integer values (i.e. functions from integers to integers). We use here two simple JML-like clauses: the `ensures` clause in order to indicate the integer outcome of a function that is expected by the code consumer, and the `requires` clause to indicate any precondition on an input integer parameter of a function. We also use a JML-like statement, the `assert` statement, to indicate conditions on the local variables of the method. These integer arithmetic properties were initially analyzed in [Alba-Castro et al., 2008].

Let us motivate our work by focusing on some simple Java programs borrowed from the related literature, that we are interested to certify.

Example 2. Consider a simple Java program, introduced in [Wu et al., 2003], with the requirement to produce an even number as a result. We express this requirement as a safety policy in the specification language JML-style by using the `ensures` clause and the JML operator `\result`. Namely, we require that the Java outcome is not an odd number when the execution of the method is completed.

```
/*@    requires true;
  @    ensures AbsValue(\result)  == #even ;         @*/

static int even16()  {
       int x = 4; int y = x + 8;
       return x+y;
 }
```

A dedicated, standard verification tool for Java such as JavaFAN [Farzan et al., 2004a] can help verify the program above since there is only one initial

state and its space state is finite. JavaFAN [Farzan et al., 2004a] is a Java program analysis framework that can symbolically execute multithreaded programs, detect safety violations by exploring an unbounded state space, and verify finite state programs by explicit state model checking. Both Java source language and JVM bytecode analyses are possible. JavaFAN's implementation consists of a Maude program, specifying formally the semantics of Java and JVM in rewriting logic. Unfortunately, no safety certificate would be delivered by JavaFAN that could be inexpensively tested at the consumer side.

The Bogor explicit-state model checking JML tool can also verify this program because the state space is finite. However, Bogor cannot produce a formal proof that would be used as a formal certificate (see Section 4.1.1 on page 59 for a discussion on the Bogor tool).

The ESC/Java2 tool JML tool can be used to verify the above example (see Section 4.1.1 for a discussion on ESC/Java2). Other similar static verifiers based on theorem proving and targeted for the Java Card Language with JML-like annotations such as JML4, JACK, Krakatoa, KeY, Jive and LOOP can also be used to verify the above example (see Section 4.1.1). However, ESC/Java2 can not generate certificates upon the success of the validation. JML4, KeY, LOOP, and Jive proofs can be checked by relying on full theorem provers themselves that were used to produce the proofs. JACK and Krakatoa Coq-generated proofs can be used as certificates that can be validated by using the Coq type-checker, a small part of the theorem prover. However, Coq proofs cannot be independently checked.

There are other verification tools based on theorem proving that do not rely on the JML functional specification language. For instance, the KIV tool (Karlsruhe Interactive Verifier). KIV is a tool for formal systems development based on theorem proving that can be used with the functional high-order algebraic specification language CASL (CoFI algebraic specification language). KIV can be also used with dynamic abstract state machine specifications [Balser et al., 2000]. KIV deduction is based on a sequent calculus with proof tactics for first-order reasoning, and a proof strategy based-on symbolic execution and induction for dynamic logic. KIV offers heuristics for automated reasoning and includes a simplifier that is based on conditional rewriting. The user can choose the heuristics and the rewriting rules. KIV was used to verify sequential Java applications, including a full Java card application, a decimal numbers program, and a linked list program [Stenzel, 2005]. KIV was also used to verify the Java implementation of the Mondex

electronic purse [Grandy et al., 2008]. The KIV system can also be used to verify the above example, but it cannot produce an independent checkable proof.

The proposal in [Wu et al., 2003] is the first FPCC approach [Appel and Felty, 2000; Appel, 2001; Appel and Mcallester, 2001; Felty, 2005](see Section 1.1.1) that considered the verification and certification of integer arithmetic properties of Java methods like the Example 2 above, but for the case of a low-level language and by relying on type systems. This work uses a simple integer type system based on the even/odd parity property.

The following example illustrates the mechanization of the Java semantics of Chapter 3 by using the Java program of Example 2.

Example 3. Consider the Java program of Example 2 together with the following Java main function:

```
void main() { System.out.println(even16()); }
```

The Maude command `search` provides us with built–in breadth-first search, i.e., it provides all the sequences of rules (recall that the application of equations is omitted throughout the whole search space) from an initial term (without variables) to a final term (possibly with variables) [Clavel et al., 2007]. Note that the ground initial term describes a concrete initial Java state whereas the final term (possibly with variables) describes a (possibly infinite) set of final Java states. In the search command below, we ask for all possible values returned by the `main` Java function of Example 2, and therefore, a variable term denotes the goal state to be reached. Note that the code of the two Java functions `even16` and `main` is embedded within the search command, as well as the initial call to `main` (see Section 3.3 and [Farzan et al., 2007] for details on how to build an initial Java state).

```
search in PGM-SEMANTICS : java((preprocess(default class 'Safe1Even1
extends Object implements none {
 (default static) int 'even16(noPara)throws(noType)
  {int d('x) = i(4) ; int d('y) = 'x + i(8) ; 12 @ return 'x + 'y ;}
 (public static) void 'main(t('String)[] d('args))throws(noType)
  {5 @ ('System . 'out . 'println < 'even16 < noExp > > ;)} })
t('Safe1Even1) . 'main < new string [i(0)] > noVal))
=>! X:Output .

Solution 1 (state 0)
X:Output --> pl(int(16))
```

The search command returns that one unique possible Java execution trace is possible, which leads to the Java value 16 as the outcome of the Java instruction "`System.out.println(even16());`". The whole rewriting sequence leading to this Java value is also delivered by Maude.

Example 4. Consider a more elaborated Java program together with a similar "even" safety policy required on both, the input and the output of the function. The JML-like statement "`assert AbsDomain(u) == EvenOdd;`" states that variable `u` should be "even" or "odd".

```
/*@    requires AbsValue(j) == #even;
  @    ensures AbsValue(\result)  == #even ;         @*/

static int evenOdd(int j)  {
       int u = 3; int v,z = 4;
/*@    assert AbsDomain(u) == EvenOdd ;
  @    assert AbsDomain(v) == EvenOdd ;
  @    assert AbsDomain(z) == EvenOdd ;             @*/
       z += 30;
       v = u*8 + j;
       return z - v;
}
```

In this case, an infinite number of initial states are considered, although the search space is finite for each of them. Since Java verification tool JavaFAN does not support program abstraction, all concurrent computations of the program need to be explored. This can be done either by symbolic simulation or by explicit-state model checking of the property (specified in linear temporal logic). Thus, for the infinite-state program of Example 4 above, the JavaFAN and Bogor tools can only be used as semi-decision procedures to look for safety violations starting from specific initial states. Similarly, the search command of Maude using the concrete Java semantics of Chapter 3 could not be used either.

There are some model checking tools that reduce the state search space by using abstraction and do not use the JML specification language. For instance, the Bandera tool generates finite-state models from concurrent Java source code annotated with BSL (Bandera Specification Language) statements [Corbett et al., 2000b; Corbett et al., 2000a]. Bandera uses abstract techniques such as abstract interpretation, program slicing and specialization in order to reduce the state space, to be model checked by SPIN and SMV [Corbett et al., 2000a]. BSL statements behave as Javadoc comments and eases the model checking process by avoiding property specification in LTL and CTL logics

which may be difficult to use for non-experimented users. BSL has an assertion sublanguage that allows developers to define constraints on the program context in familiar assertion-style notation. BSL has also a temporal property sublanguage which provides support for defining predicates on Java control points (method invocation and return, and labels of break an continue statements) using selected assertions written in the assertion sublanguage. BSL examples consider proper shutdown pipeline computation properties [Corbett et al., 2000b], bounded buffer non-temporal properties (positive bound, add to end, index range invariant) and temporal properties (full buffers eventually get emptied and empty buffers must be added before being taken) [Corbett et al., 2000b], and readers and writers correctness properties [Corbett et al., 2000a].

Bandera uses abstract interpretation to reduce the size of the variable domains. Besides, Bandera has predefined abstractions for some concrete types; for instance, the abstractions for integers that are organized by families, include *range*, *set*, *modulo*, and *point* families [Dwyer et al., 2001]. Then, Bandera can be used to verify the Example 4 by model checking the corresponding finite-state search space by using the *modulo*-`k` family for integer variables, with `k = 2`. On the other hand, Bandera is just a validation tool; then, after the successful verification, it cannot deliver any proof that would be used as a formal certificate.

The JML-based theorem proving tools ESC/Java2, JML4, Jack, Krakatoa, KeY, LOOP and Jive can indeed be used to verify the above example, as well as the KIV tool, but they cannot produce independent checkable proofs. These tools generate proofs that need the full prover in order to be validated. However, Jack and Krakatoa Coq proofs can be validated by using the Coq type-checker, without the need for the full theorem prover.

Appel et.al. type system [Wu et al., 2003](see 1.1.1) can also be used to verify Example 4.

Our last example is more involved, containing loops and conditionals, as follows.

Example 5. Consider a more realistic Java program, requiring a more involved condition on the input to ensure the fulfilment of a specific safety property [Alba-Castro et al., 2008]. The parity of the output is again ensured to be "even" under a more complex "modulo 4" safety policy on the input parameter: the `requires` clause states that the input parameter should be equal to `0` or `3` "modulo 4".

```
/*@    requires AbsValue(n) == #0 || AbsValue(n) == #3;
  @    ensures AbsValue(\result)  == #even ;              @*/

static int summation(int n) {
       int sum = 0 ;
//@    assert AbsDomain(sum) == EvenOdd ;
       int i = 0;
//@    assert AbsDomain(i) == Mod4;
       while (i<=n)   {
           sum += i;
           i++;
       }
       return sum;
}
```

In order to deal with the condition of the `while`, we have to specify for the counter variable `i`: the JML-like clause "`assert AbsDomain(i) == Mod4;`" states that the value of variable `i` "modulo 4" should range from `0` to 3.

Although the Java semantics of [Farzan et al., 2007] considers non-deterministic Java programs, in this chapter we consider deterministic Java programs, i.e. programs with no threads, and no exceptions.

5.2 The abstract RL semantics of Java for the arithmetic domain

In this section, we develop an abstract version of the rewriting logic semantics of Java described by the rewrite theory $\mathcal{R}_{\text{Java}^\#} = (\Sigma_{\text{Java}^\#}, E_{\text{Java}^\#}, R_{\text{Java}^\#})$, $E_{\text{Java}^\#} = \Delta_{\text{Java}^\#} \uplus B_{\text{Java}^\#}$ and its corresponding $\rightarrow_{\text{Java}^\#}$ rewriting relation. Recall that the rewrite theory $\mathcal{R}_{\text{Java}}$ of Chapter 3 is defined on a generic sort Value. Our approach consists in extending $\mathcal{R}_{\text{Java}}$ (taking advantage of its modularity) by creating abstract domains as subsorts of the sort Value and adding the appropriate versions of the Java constructions and operators for the corresponding abstract domains.

In our approach, the code consumer can assign a different abstract domain to each variable in the Java source code in order to obtain a finite-state model of the program. This is an important point, since a potential user of the tool only has to select some source variables to be abstracted together with the selected abstraction. A graphical interface equipped with user–friendly advisory

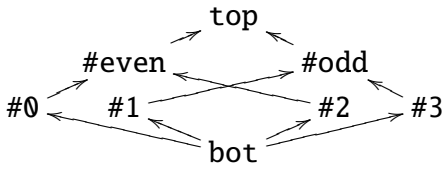

Figure 5.1: Lattice of integers for the *mod*2 and *mod*4 abstractions.

facilities can help her in this process. Furthermore, the user could simply annotate the source code with JML-like assertions encoding the required safety policy so that the critical variables (together with their appropriate abstract domains) might be automatically inferred, although in this case the abstraction might be less accurate.

For the process of assigning an abstract domain to each source variable, we have to consider both the theoretical and practical aspects of the problem. On the theoretical side, we define an abstract function for each Java variable name `x`, e.g., $\alpha_{\texttt{x}} : \wp(\mathsf{Int}) \to \wp(\mathsf{Int})$, and homomorphically extend those abstract functions to an abstract function $\alpha : \wp(\mathsf{State}) \to \wp(\mathsf{State})$. Indeed, for each variable `x`, function α abstracts the values stored in the Java memory for `x` using $\alpha_{\texttt{x}}$, which can be the identity function if no abstract domain is selected. As mentioned before, these assignments of an abstract domain to a source variable can be inferred from the JML-like annotations, e.g. "`AbsValue(\result) == #even`" or "`assert AbsDomain(u) == EvenOdd`", in the Java source code. The following example shows some abstract domains which are relevant to this work [Alba-Castro et al., 2008].

Example 6. Let us consider an abstract function that classifies Java integers into even and odd classes, i.e., $\texttt{mod2} : \wp(\texttt{int}(\mathsf{Int})) \to \wp(\texttt{int}(\mathsf{Int}))$ where $\texttt{int}(\mathsf{Int})$ denotes the Maude terms of sort Value that correspond to the Java integers. This abstraction is relevant for Examples 2, 4, and 5. We can choose the following abstract symbols $\texttt{EvenOdd} = \{\texttt{bot}, \texttt{\#even}, \texttt{\#odd}, \texttt{top}\}$ to denote the following subsets $\texttt{top} = \texttt{int}(\mathsf{Int})$, $\texttt{bot} = \emptyset$, $\texttt{\#even} = \{\texttt{int}(\texttt{n}) \mid \texttt{n}\ \mathit{mod}\ 2 = 0\}$, and $\texttt{\#odd} = \{\texttt{int}(\texttt{n}) \mid \texttt{n}\ \mathit{mod}\ 2 = 1\}$. And we consider the abstraction function $\texttt{mod2} : \wp(\texttt{int}(\mathsf{Int})) \to \texttt{EvenOdd}$. We can even refine such an abstract domain by including the abstraction for Java integers modulo 4, i.e., $\texttt{mod4} : \wp(\texttt{int}(\mathsf{Int})) \to \wp(\texttt{int}(\mathsf{Int}))$. This abstraction is suitable for Example 5. Let us introduce the following abstract symbols $\texttt{Mod4} = \{\texttt{bot}, \texttt{\#0}, \texttt{\#1}, \texttt{\#2}, \texttt{\#3}, \texttt{top}\}$, where $\texttt{\#k} = \{\texttt{int}(\texttt{n}) \mid \texttt{n}\ \mathit{mod}\ 4 = k\}$ for $k \in \{0, 1, 2, 3\}$, and the abstraction

```
--- Define abstract domains
sorts EvenOdd Mod4 . subsort EvenOdd Mod4 < Value .
ops even odd : -> EvenOdd .
op #_: Int -> Mod4 .
--- Define abstraction functions
op mod2 : Value -> EvenOdd .
eq mod2(int(I)) = if (I rem 2 == 0) then even else odd fi .
op mod4 : Value -> Mod4 .
eq mod4(int(I)) = #(I rem 4) .
--- Equations for abstracting concrete values
op inAbsDomain : Qid Value -> Value .
eq inAbsDomain('x,int(I)) = mod2(int(I)) . --- Examples 1,2
eq inAbsDomain('y,int(I)) = mod2(int(I)) . --- Examples 1,2
eq inAbsDomain('n,int(I)) = mod4(int(I)) . --- Example 3
eq inAbsDomain(Var,Value) = Value [owise] .
```

Figure 5.2: Abstract domain and association of abstract domain to variable name.

```
--- BuildEnv modified equation
eq t(k(buildEnv(((T d(Var)), Pl), (Value, Vl)) -> K) env(Env) TC) store(ST) nextLoc(I')
 = t(k(buildEnv(Pl, Vl) -> K) env([Var, l(I' + 1)] Env) TC)
   store([l(I' + 1), inAbsDomain(Var,Value)] ST) nextLoc(I' + 1) .
--- Assignment modified equations
op = : Exp Qid -> Continuation . --- new definition
op = : Location Qid -> Continuation . --- new definition
eq k((Var = E) -> K) = k(getLoc(Var) -> (=(E,Var) -> K)) .
eq k(Loc -> (=(E,Var) -> K)) = k(E -> (=(Loc,Var) -> K)) .
eq k(Val -> (=(Loc,Var) -> K)) = k([inAbsDomain(Var,Val) -> Loc] -> (Val -> K)) .
```

Figure 5.3: Modified continuation-based equations for building environments and Java assignment.

function `mod4` : $\wp(\texttt{int}(\mathsf{Int})) \rightarrow \texttt{Mod4}$. The lattice induced by the relation $\subseteq$ on sets of Java integer values is shown in Figure 5.1.

On the practical side, we have to supplement the original Java semantics with a new Maude function, called `inAbsDomain`, that records the abstract domain associated to each variable name and that will be used in two execution points: when the variable is initially created in the Java memory and everytime its value is updated in the memory. For instance, Figure 5.2 shows the code of the `inAbsDomain` function for variables `x`,`y` of Example 2, also for the variable `n` of Example 5, according to the JML-like annotations, together with the Maude code for the abstract functions `mod2` and `mod4`. We also have to add a call to the `inAbsDomain` function in the `buildEnv` continuation symbol of Figure 3.6 and the Java assignment operator of Figure 3.8; all these modifications are shown in Figure 5.3. Obviously, we have to provide abstract versions of all the Java operators in the Java semantics dealing

```
--- Add abstract mod2 values
eq k((even, even) -> (+ -> K)) = k(even -> K) .
eq k((even, odd) -> (+ -> K)) = k(odd -> K) .
eq k((odd, even) -> (+ -> K)) = k(odd -> K) .
eq k((odd, odd) -> (+ -> K)) = k(even -> K) .
--- Add mod2 values with standard integer values
var Val EvenOdd .
eq k((int(I), Val) -> (+ -> K)) = k((mod2(int(I)), Val) -> (+ -> K)) .
eq k((Val, int(I)) -> (+ -> K)) = k((Val, mod2(int(I))) -> (+ -> K)) .
--- Add mod2 with Mod4 values
eq k((# I, Val) -> (+ -> K)) = k((mod2(int(I)), Val) -> (+ -> K)) .
eq k((Val, # I) -> (+ -> K)) = k((Val, mod2(int(I))) -> (+ -> K)) .
```

+	even	odd
even	even	odd
odd	odd	even

Figure 5.4: Abstract definition and equations for the abstract Java addition operator with `EvenOdd` values.

with such kind of values, e.g., we must provide an approximation of integer addition, less-or-equal boolean operator, etc. dealing with the new abstract domains for integers. For instance, given the abstract function `mod2`, the addition operation on integers is specified in Figure 5.4.

In abstract interpretation, it is common to compress several computation steps into one abstract computation step, in order to reflect the fact that several distinct behaviors are mimicked by an abstract state. Consider for instance the Java less-or-equal operator <= of Figure 3.5 and the abstract function `mod2`. For the case when we compare two even expressions with <=, an (inaccurate) approximation of the result is the union of `true` and `false`, which is denoted by the symbol `top`. A naïve implementation of this idea would consist in including the following equation into the abstract Java semantics $\mathcal{R}_{\text{Java}^\#}$ (following the definition of operator <= in Figure 3.5):

```
eq k((even, even) -> (<= -> K)) = k(top -> K) .
```

This instrumentalization of the Java semantics for dealing with abstraction implicitly means too many modifications of the semantics, since completely different Java states could be generated that have to be packed together into a unique abstract Java state. For instance, consider a Java expression "`if eb then et else ef`" such that the expression `eb` returns `top` so that we have to represent within a single Java state both, the case when we reach a Java state continued by executing instruction `et` and also the case when we reach a Java state continued with the instruction `ef`. This would amount to a deep modification of the whole Java semantics, in order to cope with sets of Java states. Therefore, we adopt a different approach. When several $\rightarrow_{\text{Java}}$ rewrite

```
--- EvenOdd
vars Eo1 Eo2 : EvenOdd .
rl k((Eo1, Eo2) -> (<= -> K)) => k(bool(true) -> K) .
rl k((Eo1, Eo2) -> (<= -> K)) => k(bool(false) -> K) .
--- Mod4
rl k((# I, # I') -> (<= -> K)) => k(bool(true) -> K) .
rl k((# I, # I') -> (<= -> K)) => k(bool(false) -> K) .
```

Figure 5.5: Continuation-based equations for Java less-or-equal abstract operator on EvenOdd and Mod4 abstract integers.

steps are mimicked by an abstract Java state and those rewrite steps apply different rules or equations, we use concurrency at the Maude level. That is, we add rules to $R_{\text{Java}^\#}$ to reflect the different possible evolutions of the system. Following this approach, the Java less-or-equal operator is defined as shown in Figure 5.5, describing that the comparison operator <= can indifferently return `true` or `false` [Alba-Castro et al., 2008].

The specifications given in Chapter 3 for the variable content retrieval (Figure 3.7), the if-the-else statement (Figure 3.9), the while statement (Figure 3.10), and the break statement (Figure 3.11) do not need any modification. The specification of instance method invocation shown above in Figure 3.12 can also be used here with no modification, given the specification for abstract environment building provided in Figure 5.3. The local environment of the method is built by applying the generic abstraction function `inAbsDomain` specified in Figure 5.2 above to all method parameters.

5.2.1 Abstract rewriting formalization

Now, we are ready to formalize the abstract rewriting relation $\rightarrow_{\text{Java}^\#}$, which intuitively develops the idea of applying only one rule or equation from the concrete Java semantics to an abstract Java state while exploring the different alternatives in a non-deterministic way. By abusing notation, we denote the abstraction of a rule $\alpha(\{l\}) \rightarrow \alpha(\{r\})$ by $\alpha(\{l\} \rightarrow \{r\})$.

Definition 2 (Abstract rewriting)**.** Let $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ be an abstraction. We define the approximated version of rewriting $\rightarrow_{\text{Java}^\#} \subseteq \wp(\mathsf{State}) \times \wp(\mathsf{State})$ by:

$$SSt_1 \rightarrow_{\text{Java}^\#} SSt_2 \text{ using } \alpha(\{l\} \rightarrow \{r\}) \in (R_{\text{Java}^\#} \cup \Delta_{\text{Java}^\#})$$
$$\text{iff } \exists u \in \alpha(SSt_1), \exists v \in SSt_2 \text{ s.t. } u \rightarrow_{\text{Java}} v, \text{using } l \rightarrow r \in R_{\text{Java}} \cup \Delta_{\text{Java}}.$$

We denote by $\rightarrow^{*}_{\text{Java}^{\#}}$ the extension of $\rightarrow_{\text{Java}^{\#}}$ to multiple rewrite steps.

In order to guarantee that the abstract semantics correctly (over-) approximates the concrete semantics, we have to demonstrate that:

1. The abstraction is sound regarding the relation between the concrete and abstract domains of program states, based on the abstraction function α. This can be done by proving that this abstraction function α and a corresponding concretization function γ, i.e. the pair $\langle \alpha, \gamma \rangle$ is a *Galois insertion* [Cousot and Cousot, 1979; Cousot and Cousot, 2002; Cousot, 2004]. This result is formalized in the Theorem 1 below.

2. All concrete program traces have corresponding abstract program traces, such that no concrete program trace is disregarded. The transformation of a set of equations (which are confluent and terminating modulo axioms) into rules preserves the execution traces. The abstraction of the value component of states, i.e. the replacement of the concrete value by the abstract one, does not eliminate execution traces either. This result is formalized in the Theorem 2 on page 85.

Theorem 1. Given an abstraction function $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ (or $\alpha : \wp(\mathsf{State}) \rightarrow \mathsf{State}^{\#}$) and the corresponding concretization function $\gamma : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ (or $\gamma : \mathsf{State}^{\#} \rightarrow \wp(\mathsf{State})$) it holds that, for all $S \in \wp(\mathsf{State})$ and $S^{\#} \in \mathsf{State}^{\#}$, $\alpha(S) \sqsubseteq S^{\#}$ if and only if $S \subseteq \gamma(S^{\#})$.

Proof. In order to prove that the pair of functions $\langle \alpha, \gamma \rangle$ is a Galois insertion, it is enough to prove that the pair $\langle \alpha, \gamma \rangle$ satisfy the following conditions: i) *monotonicity*, i.e. for all $S, S' \in \wp(\Sigma)$, given $S \subseteq S'$ we have that $\alpha(S) \sqsubseteq \alpha(S')$, and for all $S^{\#}, S^{\#}_1 \in \Sigma^{\#}$, $S^{\#} \sqsubseteq S^{\#}_1$ implies that $\gamma(S^{\#}) \subseteq \gamma(S^{\#}_1)$, ii) the *deflationary* property, $S \subseteq \gamma(\alpha(S))$, and iii) the *non-information loss* property, $\alpha(\gamma(S^{\#})) = S^{\#}$ [Cousot and Cousot, 1979; Cousot and Cousot, 2002; Cousot, 2004].

Next, we prove that the pairs $\langle \alpha, \gamma \rangle$ satisfy the *monotonicity*, *deflationary* and *non-information loss* properties at the variable level.

The concretization functions corresponding to the two abstraction functions `mod2` and `mod4` are the functions $\texttt{mod2}^{\sharp}$ and $\texttt{mod4}^{\sharp}$, in Figures 5.6 and 5.7, respectively.

Recall that the lattice in Figure 5.1 is a complete lattice of parity properties $\langle \mathsf{Int}^{\#}, \sqsubseteq, \texttt{bot}, \texttt{top}, \sqcup, \sqcap \rangle$ with $\mathsf{Int}^{\#} = \{\texttt{bot}, \texttt{\#even}, \texttt{\#0}, \texttt{\#2}, \texttt{\#odd}, \texttt{\#1}, \texttt{\#3}, \texttt{top}\}$, and that the set of integers is also a complete lattice regarding set inclusion,

$$
\begin{aligned}
\texttt{mod2}^\sharp(\texttt{bot}) &= \emptyset \\
\texttt{mod2}^\sharp(\texttt{top}) &= \mathsf{Int} \\
\texttt{mod2}^\sharp(\#\texttt{even}) &= \{\texttt{int(i)} \in \mathsf{Int} \mid \texttt{i mod } 2 = 0\} \\
\texttt{mod2}^\sharp(\#\texttt{odd}) &= \{\texttt{int(i)} \in \mathsf{Int} \mid \texttt{i mod } 2 \neq 0\}
\end{aligned}
$$

Figure 5.6: Concretization function $\texttt{mod2}^\sharp$.

$$
\begin{aligned}
\texttt{mod4}^\sharp(\texttt{bot}) &= \emptyset \\
\texttt{mod4}^\sharp(\texttt{top}) &= \mathsf{Int} \\
\texttt{mod4}^\sharp(\#0) &= \{\texttt{int(i)} \in \mathsf{Int} \mid \texttt{i mod } 4 = 0\} \\
\texttt{mod4}^\sharp(\#1) &= \{\texttt{int(i)} \in \mathsf{Int} \mid \texttt{i mod } 4 = 1\} \\
\texttt{mod4}^\sharp(\#2) &= \{\texttt{int(i)} \in \mathsf{Int} \mid \texttt{i mod } 4 = 2\} \\
\texttt{mod4}^\sharp(\#3) &= \{\texttt{int(i)} \in \mathsf{Int} \mid \texttt{i mod } 4 = 3\}
\end{aligned}
$$

Figure 5.7: Concretization function $\texttt{mod4}^\sharp$.

i.e. $\langle \mathsf{Int}, \subseteq, \bot, \top, \cup, \cap \rangle$. We abuse of notation by using the same term to denote both an abstract value in $\mathsf{Int}^\#$ and the set of integer numbers that satisfy the corresponding property, for instance #even denote the property and the set $\{\texttt{int(i)} \in \mathsf{Int} \mid \texttt{i mod } 2 = 0\}$.

i) *monotonicity* of γ, for all $S^\#, S_1^\# \in \Sigma^\#$, $S^\# \sqsubseteq S_1^\#$ implies $\gamma(S^\#) \subseteq \gamma(S_1^\#)$: given the specification of this function and the two lattices, it is easy to see that whenever it holds $\mathsf{Int}_1^\# \sqsubseteq \mathsf{Int}_2^\#$ with $\mathsf{Int}_1^\#, \mathsf{Int}_2^\# \in \mathsf{Int}^\#$, it also holds that $\gamma(\mathsf{Int}_1^\#) \subseteq \gamma(\mathsf{Int}_2^\#)$; therefore, γ is monotonic; *monotonicity* of α, for all $S, S' \in \wp(\Sigma)$, given $S \subseteq S'$ we have that $\alpha(S) \sqsubseteq \alpha(S')$:

1. If $S_1 = \emptyset$ we have that $S_1 \subseteq S_2$ for all $S_2 \in \wp(\mathsf{Int})$, then we have that $\alpha(S_1) \sqsubseteq \alpha(S_2)$ holds, regarding both abstraction functions, mod2 and mod4;

2. if $S_2 = \mathsf{Int}$ we have that $S_1 \subseteq S_2$ for all $S_1 \in \wp(\mathsf{Int})$, then we have that $\alpha(S_1) \sqsubseteq \alpha(S_2)$ also holds in this case, regarding both abstraction functions.

3. Given $S_1, S_2 \in \wp(\mathsf{Int})$ such that $S_1 \neq \emptyset$, $S_2 \neq \mathsf{Int}$, and $S_1 \subseteq S_2$, we have several cases depending on the inclusion relation of these sets S_1 and S_2 with the proper subsets of Int different from $\emptyset$, i.e. #even, #odd, #0, #1, #2 and #3;

(a) If $\emptyset \subset S_1 \subseteq$ #even, and $\emptyset \subset S_2 \subseteq$ #even, its easy to see that regarding the mod2 abstraction function, $\alpha(S_1) =$ #even and $\alpha(S_2) =$ #even, therefore $\alpha(S_1) \sqsubseteq \alpha(S_2)$ also holds in this case;

(b) The precedent case 3a is very similar to the case with $\emptyset \subset S_1 \subseteq$ #odd, and $\emptyset \subset S_2 \subseteq$ #odd;

(c) The cases 3a and 3b are very similar to some cases regarding mod4 abstraction function: $\emptyset \subset S_1 \subseteq$ #0, and $\emptyset \subset S_2 \subseteq$ #0, $\emptyset \subset S_1 \subseteq$ #1, and $\emptyset \subset S_2 \subseteq$ #1, $\emptyset \subset S_1 \subseteq$ #2, and $\emptyset \subset S_2 \subseteq$ #2, and $\emptyset \subset S_1 \subseteq$ #3, and $\emptyset \subset S_2 \subseteq$ #3.

(d) If $\emptyset \subset S_1 \subseteq$ #0, and $\emptyset \subset S_2 \subseteq$ #even, its easy to see that $\alpha(S_1) =$ #0 and $\alpha(S_2) =$ #even, therefore given the lattice of Figure 5.1, the inclusion $\alpha(S_1) \sqsubseteq \alpha(S_2)$ also holds in this case, regarding abstraction functions mod2 and mod4 over S_2 and S_1, respectively.

(e) The case 3d is similar to the cases where $\emptyset \subset S_1 \subseteq$ #2, and $\emptyset \subset S_2 \subseteq$ #even, $\emptyset \subset S_1 \subseteq$ #1, and $\emptyset \subset S_2 \subseteq$ #odd, and $\emptyset \subset S_1 \subseteq$ #3, and $\emptyset \subset S_2 \subseteq$ #odd.

(f) If $\emptyset \subset S_1 \subseteq$ #even, and $\emptyset \subset S_2 \subseteq$ #0 and $S_1 \subseteq S_2$, means that the integer numbers that are elements of the set S_1 are such that all them are even but also that all them are multiples of 4 (i.e. int(i) $\in$ Int and i mod 4 = 0), thus $S_1 \subseteq$ #0 and this case is reduced to one sub-case of case 3c above;

(g) The case 3f is similar to cases $\emptyset \subset S_1 \subseteq$ #even, and $\emptyset \subset S_2 \subseteq$ #2, $\emptyset \subset S_1 \subseteq$ #odd, and $\emptyset \subset S_2 \subseteq$ #1 and $\emptyset \subset S_1 \subseteq$ #odd, and $\emptyset \subset S_2 \subseteq$ #3.

Therefore, we have that α is also monotonic.

ii) the *deflationary* property $S \subseteq \gamma(\alpha(S))$: we consider the cases corresponding to the inclusion relation $\subseteq$ between S and the sets of integers $\emptyset$, #even, #0, #2, #odd, #1, #3, and Int.

1. The mod2 abstraction:

   (a) If $S \subseteq \emptyset$, $S \subseteq$ #even, $S \subseteq$ #odd and $S \subseteq$ Int, then $\alpha(S) =$ bot, #even, #odd and top, respectively.

   (b) Given the abstraction function $\texttt{mod2}^\sharp$ (Figure 5.6), therefore we also have that $\gamma(\alpha(S)) = \emptyset$, #even, #odd, and Int, respectively.

(c) Given 1a and 1b, we have that $\gamma(\alpha(S)) = \emptyset$, if $S \subseteq \emptyset$, $\gamma(\alpha(S)) =$ #even, if $S \subseteq$ #even, $\gamma(\alpha(S)) =$ #odd, if $S \subseteq$ #odd, and $\gamma(\alpha(S)) =$ Int, if $S \subseteq$ Int. Therefore, we have that $\gamma(\alpha(S)) = S'$, if $S \subseteq S'$ hence $S \subseteq \gamma(\alpha(S))$.

2. The mod4 abstraction:

(a) If $S \subseteq \emptyset$, $S \subseteq$ #0, $S \subseteq$ #2 $S \subseteq$ #1, $S \subseteq$ #3, and $S \subseteq$ Int then we have that $\alpha(S) =$ bot, #0, #2, #1, #3, and top, respectively.

(b) Given the abstraction function $\texttt{mod4}^\sharp$ (Figure 5.7) we also have that $\gamma(\alpha(S)) = \emptyset$, #0, #2, #1, #3, and Int, respectively;

(c) Given 2a and 2b, and similarly to case 1c we have that $\gamma(\alpha(S)) = S'$ if $S \subseteq S'$hence $S \subseteq \gamma(\alpha(S))$.

iii) *non-information loss*, $\alpha(\gamma(S^\#)) = S^\#$:

1. The mod2 and $\texttt{mod2}^\#$ functions:

   If $S^\# =$ bot, #even, #odd and top $\in \mathsf{Int}^\#$ then $\gamma(S^\#) = \emptyset$, #even, #odd and Int $\in \wp$(Int), respectively.

   Therefore, we have that $\alpha(\gamma(S^\#)) =$ bot, #even, #odd and top $\in \mathsf{Int}^\#$, respectively.

   We conclude $\alpha(\gamma(S^\#)) = S^\#$.

2. The mod4 and $\texttt{mod4}^\#$ functions:

   If $S^\# =$ bot, #0, #2, #1, #3 and top $\in \mathsf{Int}^\#$ then $\gamma(S^\#) = \emptyset$, #0, #2, #1, #3 and Int $\in \wp$(Int), respectively.

   Therefore, we have that $\alpha(\gamma(S^\#)) =$ bot, #0, #2, #1, #3 and top $\in \mathsf{Int}^\#$, respectively.

   The result $\alpha(\gamma(S^\#)) = S^\#$ follows, regarding mod4 and $\texttt{mod4}^\#$ functions.

□

This result for Int (and $\mathsf{Int}^\#$) can be homomorphically extended to State (and $\mathsf{State}^\#$) so that, the α : State $\rightarrow$ $\mathsf{State}^\#$ and its corresponding γ : $\mathsf{State}^\# \rightarrow$ State functions, satisfy the *monotonic*, *deflationary* and *non-information loss* properties.

Alternatively, this can also be done by proving that the given abstraction function α is an upper closure operator with monotonicity, idempotency, and extensitivity [Cousot, 2004; Cousot and Cousot, 1979].

Theorem 2 (Correctness)**.** Let $\alpha : \wp(\mathsf{State}) \to \wp(\mathsf{State})$ be an abstraction. Let $St_1, St_2 \in \mathsf{State}$. If $St_1 \to^*_{\text{Java}} St_2$, then there exists $SSt_3 \subseteq \wp(\mathsf{State})$ s.t. $\alpha(\{St_1\}) \to^*_{\text{Java}^\#} SSt_3$ and $St_2 \in SSt_3$.

Proof. The proof is done by induction on the length n of the concrete program trace or rewriting sequence denoted by $\to^*_{\text{Java}}$.

1. ($n = 1$). There is only one rewriting step $\langle P_{\text{Java}}, St_1\rangle \to_{\text{Java}} \langle St_2\rangle$. Given $St_1 \in \alpha(\{St_1\})$ and $St_2 \in \alpha(\{St_2\})$, if $SSt_1 = \alpha(\{St_1\})$ and $SSt_2 = \alpha(\{St_2\})$, then by Definition 2 (on page 80) it holds that $\langle P_{\text{Java}}, \alpha(\{St_1)\}\rangle \to_{\text{Java}^\#} \langle \alpha(\{St_2)\}\rangle$. Thus, the result holds for $n = 1$.

2. ($n > 1$). The program trace of length n, $\langle P_{\text{Java}}, St_1\rangle \to^*_{\text{Java}} \langle St_2\rangle$, can be split into two sub–traces of length $n - 1$ and 1 respectively:

$$\underbrace{\langle P_{\text{Java}}, St_1\rangle \to^*_{\text{Java}} \langle P_{\text{Java}^{\text{int}}}, St_{2'}\rangle}_{\text{length } n-1} \to_{\text{Java}} \langle St_2\rangle$$

   By the induction hypothesis, the extended sub–trace of length $n - 1$ has a corresponding abstract sub–trace:

$$\langle P_{\text{Java}}, \alpha(St_1)\rangle \to^*_{\text{Java}^\#} \langle P_{\text{Java}^{\text{int}}}, \alpha(St_{2'})\rangle.$$

   Since $St_1 \in \alpha(\{St_1\})$ and $St_{2'} \in \alpha(\{St_{2'}\})$, by Definition 2, the rewriting step $\langle P_{\text{Java}^{\text{int}}}, St_{2'}\rangle \to_{\text{Java}} \langle St_2\rangle$, has a corresponding abstract rewriting step: $\langle P_{\text{Java}^{\text{int}}}, \alpha(St_{2'})\rangle \to_{\text{Java}^\#} \alpha(\langle St_2\rangle)$. Then, the conclusion also holds for $n > 1$. □

The breadth-first search for the abstract finite state system (finite due to the use of finite abstract domains) gives us a useful tool for symbolic execution, while keeping simple the modifications of the Java semantics in Maude. Actually, verification in our framework simply boils down to the exploration of all the rewriting sequences.

Example 7. Consider the Java functions `even16` and `main` of Example 3, and the abstract Java semantics shown above with the `inAbsDomain` function of Figure 5.2. Note that, for the search command, the only change we need in

this case is the replacement of PGM-SEMANTICS with PGM-SEMANTICS-ABSTR, since the considered Java function even16 of Example 2 has no input parameters. Now we invoke function main as follows.

```
search in PGM-SEMANTICS-ABSTR : java((preprocess(
default class 'Safe1Even1  extends Object implements none  {
   (default static) int 'even16(noPara)throws(noType) {
       ((int d('x) = i(4) ;)
        (int d('y) = 'x + i(8) ;))
      12 @ return 'x + 'y ;}
   (public static) void 'main(t('String)[] d('args))throws(noType) {
      5 @ ('System . 'out . 'println < 'even16 < noExp > > ;)}})

t('Safe1Even1) . 'main < new string [i(0)] > noVal))
=>! X:Output .
```

The outcome of this search command is the following result, meaning that exactly one abstract Java execution trace is proven, which returns the abstract value even as a result of the Java instruction "System.out.println(even16());":

```
Solution 1 (state 0)
X:Output --> pl(even)
```

and therefore every real execution of the Java program of Figure 2 also returns an even value, according to Theorem 2.

However, the abstraction defined in Example 6 is not accurate enough for the Java program of Example 5, as shown in the following example.

Example 8. Consider the Java code of Example 5 together with the following function main:

```
void main() { System.out.println(summation(0)); }
```

We provide the following assignment of abstract domains for the variables in the Java program:

```
op inAbsDomain : Qid Value -> Value .
eq inAbsDomain('n,int(I)) = mod4(int(I)) .
eq inAbsDomain('i,int(I)) = mod4(int(I)) .
eq inAbsDomain('sum,int(I)) = mod2(int(I)) .
eq inAbsDomain(Var,V) = V [owise] .
```

When we search for all the results of the function main

```
search in PGM-SEMANTICS-ABSTR : java((preprocess(
default class 'Safe1Even1 extends Object implements none {
  (default static) int 'summation(int d('n))throws(noType) {
     (((int d('sum) ;)
       (int d('i) = i(0) ;))
     17 @ (while 'i <= 'n 17 @ {
          (15 @ ('sum += 'i ;))
           16 @ ('i ++ ;)}))
     18 @ return 'sum ;}
  (public static) void 'main(t('String)[] d('args))throws(noType) {
      7 @ ('System . 'out . 'println < 'summation < i(0) > > ;)}})
t('Safe1Even1) . 'main < new string [i(0)] > noVal))
=>! X:Output .
```

Maude delivers the following two results

```
Solution 1 (state 2)                    Solution 2 (state 5)
X:Output --> pl(even)                    X:Output --> pl(odd)
```

which are useless since both, an even and an odd output values are equally possible. The problem is that the boolean condition (`i <= n`) returns both `true` and `false` (in a non-deterministic way) under the `mod2` and `mod4` abstraction operators in too many situations.

5.2.2 Extending the approach to relational domains

In order to improve accuracy, we define a new, more precise abstract domain $\texttt{leq}^{\#}_{x,y}$ that is parametric w.r.t. two Java variable names x, y (which have different abstraction domains). For the previous example, this can be used to abstract variable `i` w.r.t. `n`. On the theoretical level, there are two abstract domains $\alpha_x, \alpha_y : \wp(\mathsf{Int}) \rightarrow \wp(\mathsf{Int})$ that are used for the values stored in the Java memory for variables x, y, respectively. The extension $\texttt{leq}^{\#}_{x,y} : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ takes the abstract domains α_x, α_y and captures also whether $x \leq y$ or $x > y$. On the practical level, we use the abstract symbols `leq#` and `gt#` defined in Maude as "`leq# : Abst Qid -> AbstLeqN`" and "`gt# : Abst Qid -> AbstLeqN`" where the first argument denotes the abstract domain for variable x (i.e., α_x) and the second argument is just y (the name of the second variable). For instance, for the previous example we will have an abstract expression that approximates variable `i` as `leq#(#0,'n)` which denotes that the current value of variable `i` modulo 4 is 0 and that variable `i` is less than or equal to variable `n`, whatever value `n` has been assigned during the execution; the corresponding JML-like annotation is "`//@ assert AbsDomain(i) == (Mod4,(<=,n));`". Note that we cannot use

<=	*any value of* Var
leq#(Val, Var)	true
gt#(Val, Var)	false

<=	leq#(Val, Var)	gt#(Val, Var)
any value of Var	true, false	true

Figure 5.8: Java less-or-equal operator on integers.

```
--- If  two integer variables are compared using a relational operator,
--- first, keep the variables in the continuation and evaluate them
ceq k((Var,Var') -> ( Con -> K))
    = k((Var,Var') -> (RelOp(Var,Var') -> ( Con -> K))) if RelOp(Con) .
--- if the value of one is related to the value of the other by "leq#"
eq k((leq#(V, Var'), V') -> (RelOp(Var,Var') -> ( <= -> K))) = k(bool(true) -> K) .
eq k((gt#(V, Var'), V') -> (RelOp(Var,Var') -> ( <= -> K))) = k(bool(false) -> K) .
eq k((V, gt#(V', Var)) -> (leqN(Var,Var') -> ( <= -> K))) = k(bool(true) -> K) .
rl k((V, leq#(V', Var)) -> (RelOp(Var,Var') -> ( <= -> K))) =>  k(bool(true) -> K) .
rl k((V, leq#(V', Var)) -> (RelOp(Var,Var') -> ( <= -> K))) =>  k(bool(false) -> K) .
```

Figure 5.9: Continuation-based equations for Java less-or-equal operator on integers.

the same abstract domain for the second variable, since the value of this variable can change dynamically. To see this consider, for instance, a variant of Example 5 where the loop therein contains the assignment `n- = 1`, and thus variable `n` changes at each iteration.

The adequate versions of the less-or-equal operator for this new abstract domain is shown in Figures 5.8 and 5.9. Note that here, we also introduced non-determinism by using rules when the comparison operator <= can return `true` or `false` indifferently. Note also that, for the less-or-equal operator (and for all relational operators), we have to handle as a special case the relation between two integer variables: we have to consider the possibility that the involved variables would be (abstractly) related by the "`leq#`" or "`gt#`" abstractions.

The appropriate specification of the Java abstract increment (++) integer operator is shown in Figure 5.10. The corresponding rewriting logic specifications of the Java post-increment (`Var++`) and pre-increment (`++Var`) operators are shown in Figures 5.11, 5.12, and 5.13.

Note that the post-increment and pre-increment integer operators may change the relation `leq#` of `leq#(Val,Var)` values, depending on the relation between the value `Val` and the value of variable `Var` "modulo 2" or "modulo 4". For instance, if the value `Val` is `#I`, the value of `Var` is `int(J)`, and `I == J` *modulo* 4 (see Figure 5.10), then the abstract value `leq#(Val,Var)` may correspond to the concrete value `int(J)`, and hence the incremented

++	*result*	*condition*
leq#(#(I), Var)	leq#(mod4(I + 1), Var)	true
leq#(#(I), Var)	gt#(mod4(I + 1), Var)	if mod4(Var) = #(I)
gt#(#(I), Var)	gt#(mod4(I + 1), Var)	true

Figure 5.10: Specification of Java post- and pre-increment operator on integers.

```
--- Keep ++' operator, location and the old value in the continuation,
--- and use the ++ operator to increment the leq# value
eq t(k(leq#(Val,Var) -> ++'(L) -> K)
  = t(k(leq#(Val,Var) -> ++ -> (++'(nullv,L,leq#(Val,Var)) -> K)) .
--- The value of Var in memory has to be obtained before incrementing.
--- We have two cases:
--- 1) if the value of Var is not equal to Val,
---    keep the leq# contructor with a null value part and its variable
---    part in the continuation, and increment the value part
ceq t(k(leq#(Val,Var) -> ++ -> K) env(E [Var,L2]) Tc) store(St [L2,Val',N'])
    = t(k(Val -> ++ -> (leq#(nullv,Var)-> K)) env(E [Var,L2]) Tc)
      store(St [L2,Val',N']) if (Val =/= Val')  .
```

Figure 5.11: Continuation-based equations for Java post- and pre-increment operator for `leq#` values: Case 1) if the value of `Var` is not equal to `Val`.

value might now not be less or equal than the value of variable `Var` (i.e. it can become greater than the value of variable `Var`).

In other words, the post-increment and pre-increment integer operators have non-deterministic results when applied to `leq#(Val,Var)` values, thus they have to be specified by using rules instead of equations, as shown in Figures 5.10, 5.11 and 5.12. The post-increment operator is defined by using the pre-increment operator. The relation `gt#` of the `gt#(Val,Var)` values is invariant regarding the post-increment and pre-increment integer operators, as shown in Figure 5.13.

Example 9. Let us reconsider now Example 8. The code of function `inAbsDomain` for Example 5 is as follows, denoting that variables `i` and `n` have domains `mod4`, variable `sum` has domain `mod2` and that the relation $i \leq n$ is also represented in the abstract domain:

```
op inAbsDomain : Qid Value -> Value .
eq inAbsDomain('n,int(I)) = mod4(int(I)) .
eq inAbsDomain('i,int(I)) = leq#(mod4(int(I),'n) .
eq inAbsDomain('sum,int(I)) = mod2(int(I)) .
eq inAbsDomain(Var,V) = V [owise] .
```

When we search for solutions to the Java function `main` using the following command

```
--- The second case:
--- 2) if the value of Var is equal to Val, either
--- 2.1) the relation "leq#" may not change:
crl t(k(leq#(Val,Var) -> ++ -> K) env(E [Var,L2]) Tc)
    store(St [L2,Val',N'])
  => t(k(Val -> ++ -> (leq#(nullv,Var)-> K)) env(E [Var,L2]) Tc)
     store(St [L2,Val',N']) if (Val == Val')  .
--- or 2.2) the relation "leq#" may change:
crl t(k(leq#(Val,Var) -> ++ -> K) env(E [Var,L2]) Tc) store(St [L2,Val',N'])
  => t(k(Val -> ++ -> (gt#(nullv,Var)-> K)) env(E [Var,L2]) Tc)
     store(St [L2,Val',N']) if (Val == Val')  .
--- fill the null value part of the leq# and gt# contructors
eq k(Val -> leq#(nullv,Var) -> K) = k(leq#(Val,Var) -> K) .
eq k(Val -> gt#(nullv,Var) -> K) = k(gt#(Val,Var) -> K) .
--- store the new value and yield the old value
eq k(Val -> ++'(nullv,L,Val') -> K) = k([Val -> L] -> (Val' -> K)) .
```

Figure 5.12: Continuation-based equations for Java post- and pre-increment operator for `leq#` values: Case 2) if the value of `Var` is equal to `Val`.

```
--- Keep ++ operator, gt# constructor with a null value and the variable Var
--- and use the ++ operator to increment the value Val
eq k(gt#(Val,Var) -> ++(L) -> K) = k(Val -> ++ -> (++(gt#(nullv,Var),L) -> K)) .
--- put the incremented value part within the constructor "gt#"
eq k(Val -> ++(gt#(nullv,Var),L) -> K) = k([gt#(Val,Var) -> L] -> (gt#(Val,Var) -> K)) .
--- Keep the old value, gt# constructor with a null value and the variable Var
--- and increment the value part
eq k(gt#(Val,Var) -> ++'(L) -> K) =
   k(Val -> ++ -> (++'(gt#(nullv,Var),L,gt#(Val,Var)) -> K)) .
--- put the incremented value part within the constructor gt#
eq k(Val -> ++'(gt#(nullv,Var),L,Val') -> K) = k([gt#(Val,Var) -> L] -> (Val' -> K)) .
```

Figure 5.13: Continuation-based equations for Java pre- and post-increment operator for `gt#` values.

```
search in PGM-SEMANTICS-ABSTR : java((preprocess
   (default class 'Safe1Even1 extends Object implements none {
       (default static) int 'summation(int d('n))throws(noType) {
            (((int d('sum) ;)
              (int d('i) = i(0) ;))
        17 @ (while 'i <= 'n 17 @ {
                  (15 @ ('sum += 'i ;))
                   16 @ ('i ++ ;)}))
        18 @ return 'sum ;}
      (public static) void 'main(t('String)[] d('args))throws(noType) {
          7 @ ('System . 'out . 'println < 'summation < i(0) > > ;)}})
   t('Safe1Even1) . 'main < new string [i(0)] > noVal))
   =>! X:Output .
```

we get the following unique output, meaning that exactly one abstract Java execution trace is proven, which returns the abstract value `even` as a result of the Java instruction "`System.out.println(summation(0))`":

```
Solution 1 (state 2)
X:Output --> pl(even)
```

This certifies that every possible Java execution starting with an integer n such that $n \ mod \ 4 = 0$ does always return an even value. Indeed, we can verify that the initial calls "`System.out.println(summation(0))`" and "`System.out.println(summation(3))`" do return the abstract value `even` whereas "`System.out.println(summation(1))`" and "`System.out.println(summation(2))`" return the abstract value `odd`.

Regarding Example 5, the user of the Bandera tool can choose *modulo*-`2` abstraction for the variable `sum` and *modulo*-`4` for the variables `i` and `n`. The assignment `sum+ = i` does not cause an abstract type conflict because each *modulo*-`4` value has one corresponding *modulo*-`2` value. However, the problem here is the relational operation `i <= n`, i.e. the comparison between *modulo*-`4` and *modulo*-`2` values, that yields the boolean values {*true*,*false* }, the only correct abstraction in this case. This problem is similar to the problem of our non-extended abstract semantics, as shown in Example 8 on page 86. Alternatively, the Bandera's user may choose to abstract the variables `i` and `n` to their concrete type in order to maximize the precision of the abstraction but in this case the state space becomes infinite. This means that the Bandera tool does not handle relational abstraction [Cousot and Halbwachs, 1978] and then it cannot verify Example 5.

The JML-based theorem proving tools JML4, LOOP, Jack, Krakatoa, KeY, and Jive can also be used to verify Example 5 (and ESC/Java2 with the `-loopSafe` switch) as well as the KIV tool, but they require users to supply loop invariants. Alternatively, KIV and KeY users may choose to prove loop post-condition by induction. Recall that these tools cannot produce independent checkable proofs. These tools generate proofs that need the full prover in order to be validated.

The type system of Appel et al. work [Wu et al., 2003] (see FPCC Section 1.1.1 on page 10) deals with even/odd properties only, and it does not considers programs with loops as Example 5.

Besson et al. [Besson et al., 2005; Besson et al., 2006; Besson et al., 2007] (see Section 1.1.2 on page 20) introduced a certified abstract interpretation approach to the verification of relational linear properties among program integer variables. These relational, abstraction-based techniques can handle more complex relations among integer variables than the considered in Example 5,

i.e. linear relations involving more than two variables. However, it does not simultaneously consider properties of integer results of Java methods, like the parity properties.

The abstract domains `EvenOdd` and `Mod4` could be specified in rewriting logic by using equational abstractions [Meseguer et al., 2003]. For instance, regarding the `EvenOdd` domain, the even integers can be reduced to `int(0)` while the odd ones can be reduced to `int(1)`. In this case, we have to introduce conditional equations to reduce stored values of program variables depending on the corresponding JML-style annotation, if any. The specification of the integer arithmetic operators does not need any change. However, the specification of the relational operators need to be modified in order to recover correctness while keeping equations confluent by introducing rules to cope with non-determinism. For instance the expression "`int(0)` $\leq$ `int(1)`" should be evaluated to `bool(true)` and to `bool(false)`. The difference between concrete and abstract operations become unclear. This way, equational abstractions can be applied to analyse Examples 2 and 4, yet avoiding the specification of some abstract operators, but with the additional cost of conditional evaluation during term reduction. The kind of relational abstraction based on inequations or linear constraints that is required for the analysis of Example 5 can also be specified by using slightly more complicated conditional equational abstractions, but all abstract operations need to be specified because of the involved non-determinism that implies the "`leq#`" and "`gt#`" abstractions.

Examples with more complicated but linear relations among integer variable values would require a more general approach to linear constraint solving like polyhedra [Cousot and Halbwachs, 1978; Besson et al., 2005; Besson et al., 2006; Besson et al., 2007].

However, we cannot handle recursive functions because their abstraction may introduce an infinite sequence of recursive invocations, as in the case of Example 10.

Example 10. This is the recursive version of the `summation` method in Example 5.

```
/*@    requires AbsValue(n) == #0 || AbsValue(n) == #3;
  @    ensures AbsValue(\result)  == #even ;              @*/

static int summation(int n) {
       aux_summation(n);
}
```

Code example	Source Size (bytes)	Full Cert. Size (Kbytes)	Red. Rule Cert. Size (Kbytes)	F/R	Full Cert. Gen. Time (ms)	Red. Rule Cert. Gen. Time (ms)	Full Cert. Val. Time (ms)
even16	562	117	0.93	126	~0	~0	~0
even16*	767	401	3.58	112	6	4	1
evenOdd	671	312	1.08	288	~0	~0	~0
summation	870	1551	39.03	40	2294	146	344

Table 5.1: Sizes of source code and certificates, and certificate generation and validation times.

```
static int aux_summation(int n) {
       if (n>0)
            return n + aux_summation(n-1);
       else
            return 0;
}
```

The non–deterministic execution of the `if then else` statement means that, in every invocation of the method `aux_summation`, both the then part and the else part are executed. Maude cannot detect this cycle because given the Java semantics, every invocation to the `aux_summation` method creates a new local variable so that the program state always changes.

5.3 Experimental Evaluation

In Table 5.1, we study two key points for the practicality of our proposal: the size of the reduced certificate versus the full certificate and the relative efficiency of checking[1] certificates w.r.t. their generation. The experiments have been performed on a MacBook with 2 Gb RAM. Programs even16, evenOdd, and summation are the Java programs of Examples 2, 4, and 5, respectively. Program even16* performs more involved arithmetic computations than even16, including subtraction and multiplication, while returning the same result (see Example 24 in the Appendix B). The first column contains the size (in bytes) of the source code for each benchmark program. The three columns for Full Certificates show the size in Kbytes, the generation time, and the validation time, respectively, for the full certificates. Similarly the two columns for Reduced Rule Certificate show the size in Kbytes and the generation time, respectively.

[1]The checking time is estimated (see Section 4.4 on page 66).

The validation time of the reduced-rule certificates is similar to but greater than the validation time of the full certificates; this is because the reduced-rule certificate includes the state space (i.e. there is no need to generate the state space). On the other hand, the reduced-label certificate does not include the state space but only the labels of the used rules; then the validation time of the reduced-label certificate is similar to but less than the generation time of the full certificates.

Running times are given in milliseconds and were averaged over a sufficient number of iterations. Our figures demonstrate that the reduction in size of the certificate is very significant in all cases, ranging the quotient F/R (Full Cert. Size/Red. Rule Cert. Size) from 288 in even16* to 40 for summation. When we compare the time employed to generate the (full and reduced-rule) certificates w.r.t. the corresponding validation time, we have that if the certificate size is reduced by a factor of 40 the validation time only grows by a factor of 3. Thus we conclude that, by minimizing the number of equations in the certificate, we achieve a simpler and indeed superior certificate that can be verified efficiently.

CHAPTER 6

Analyzing Confidentiality of Java Programs

6.1 Introduction

Confidentiality is a property by which information that is related to an entity or party is not made available or disclosed to unauthorized individuals, entities, or processes. One way to protect confidential data is by establishing an *access control policy* [Bishop, 2004] that restricts the access to objects depending on the identity or the role performed by the user, meaning that some privilege is required to access confidential data.

A user might establish an access control policy by stipulating that no data that is visible to other users be affected by confidential data. Such a policy allows programs to manipulate and modify confidential data as long as the observable data generated by those programs do not improperly reveal information about the confidential data. A security policy of this sort is called a *non–interference policy* [Denning and Denning, 1977] because confidential data should not interfere with publicly observable data. Thus, ensuring that a program adheres to a non–interference policy means analyzing how information flows within the program.

The mechanism for transfering information through a computing system is called a *channel*. Variable updating, parameter passing, value return, file reading and writing, and network communication are channels. Channels that use a mechanism that is not designed for information communication are called *covert channels* [Sabelfeld and Myers, 2003]. There are covert channels such as the control structure of a program, termination, timing, exceptions, and resource exhaustion channels.

The information flow that occurs through channels is called *explicit flow* [Denning and Denning, 1977] because it does not depend on the specific information that flows. The information flow that occurs through the control

structure of a program (conditionals, loops, breaks, and exceptions) is called an *implicit flow* [Denning and Denning, 1977] because it depends on the value of the condition that guards the control structure.

In this chapter, we are interested in both explicit and implicit flows for non–interference analysis of deterministic Java programs. However, we do not consider covert channels such as termination, timing, exceptions, and resource exhaustion channels, i.e., releasing information through termination or non termination of a computation, through the time at which an action occurs, or by the exhaustion of a finite shared resource such as memory.

This chapter provides a comprehensive and full-fledged formulation of our abstract non–interference certification methodology, namely: (i) the characterization of non–interference as a safety property on extended Java computations; (ii) the conditions required by Java programs in order to ensure the correctness of our methodology; (iii) the observational capabilities of an attacker; and (iv) the soundness of our abstract non–interference analysis technique. This non–interference certification technique was originally introduced at the local level, i.e. for Java methods, and at the global level, i.e. for complete Java programs, in [Alba-Castro et al., 2009a] and [Alba-Castro et al., 2010a], respectively.

6.2 Non–interference policies

A non–interference policy establishes a confidentiality level for each source program variable of primitive datatypes. It guarantees that actual values of variables with a higher confidentiality level do not influence the output of a variable with a lower confidentiality level during program execution [Denning and Denning, 1977; Goguen and Meseguer, 1982; Sabelfeld and Myers, 2003; Barthe and Rezk, 2005; Warnier, 2005; Dufay et al., 2005]. It is implicitly assumed that constants that appear in a program always have the lowest confidentiality level as in [Denning and Denning, 1977] (i.e., the considered program is authorized to access secret data, but it does not contain secret data in its code).

A non–interference policy can be represented by a *partially ordered set* $\langle Labels, \leq \rangle$ and a labeling function $Labeling : Var \rightarrow Labels$, where $Labels$ is the finite set of confidentiality levels, $\leq$ is a partial order between confidentiality levels, and Var is the set of source program variables [Volpano et al.,

1996; Barbuti et al., 2002; Hunt and Sands, 2006].

There are usually two confidentiality levels given by: $\mathit{Labels} = \{\texttt{Low}, \texttt{High}\}$. These represent public non-secret data (low confidentiality) and secret data (high confidentiality), respectively. $\langle \mathit{Labels}, \leq \rangle$ forms a lattice where `Low` is the greatest lower bound or *bottom* element ($\bot$), `High` is the least upper bound or *top* element ($\top$), and $\texttt{Low} < \texttt{High}$. The *join* operator ($\sqcup$) is defined as $\texttt{Low} \sqcup \texttt{Low} = \texttt{Low}$; otherwise, $X \sqcup Y = \texttt{High}$. Enforcing non–interference means that the values of `High`-labeled source variables cannot flow to `Low`-labeled source variables, whereas the values of `Low`-labeled source variables can flow to `High`-labeled source variables.

The attacker model for global non–interference that we formalize below assumes that the attacker is passive and can only see the `Low`-labeled source variables of the Java program at the initial and final execution states but not at the intermediate states. Our methodology can certify programs that have temporal breaches and are still non–interferent.

The initial confidentiality level of a variable in a Java program is written in JML-like syntax with the word `setLabel` (e.g. `setLabel(var, High)`). The confidentiality label of program variables is `Low` if nothing is specified (i.e., program variables are public by default). We do not need to specify the label of either the formal parameters or local variables because they can be inferred from the confidentiality labels of other program variables if they are properly initialized. These JML-like annotations, together with the default assumption, define the labelling function of the non–interference policy.

Example 11. Consider the following Java program borrowed from [Darvas et al., 2005] that models a bank account and the initial state given by the execution of the function `main`:

```
class System {
     static Account a = new Account();
     public static void main(String[] args) {
          int initbalance; //@ setLabel(initbalance, High);
          initbalance = Integer.parseInt(args[0]);
          a.writeBalance(initbalance);
          System.out.println(a.readExtra());
     }
}

public class Account {
     int balance; //@ setLabel(balance, High);
     public boolean extraService;
     public Account() {
          balance = 0;
```

```
        extraService = false;
    }
    public void writeBalance(int amount) {
        balance = amount;
        if (balance>=10000)
            extraService=true;
            else   extraService=false;
    }
    private int readBalance() {
        return balance;}
    public boolean readExtra() {
        return extraService;}
}
```

This non–interference policy specifies that the object field `balance` of the global object `a` and the initialization parameter `initbalance` (i.e., `args[0]`) hold secret data. This program is insecure w.r.t. this policy since an observer with low access rights can obtain partial information about the variable `balance` via an observation of the non–secret variable `extraService`.

We assume a fixed Java program P_{Java}. $Vars(P_{\text{Java}})$ denotes the set of `static` program source variables that may be initialized by the `main` function call. We denote the set of *Low* program variables as $Low(P_{\text{Java}}) = \{var \in Vars(P_{\text{Java}}) \mid Labeling(var) = \texttt{Low}\}$. A program state St is a set of value assignments to program variables. Given $var \in Vars(P_{\text{Java}})$ and a state St, $St[var]$ denotes the value of variable var in St. We model a *Java program* P_{Java} as a state transition system between pairs $\langle P, St\rangle$, where P is the current, still-to-be-executed part of the Java program P_{Java} and St represents the current program state. $\langle P_{\text{Java}}, St_0\rangle$ denotes the initial *configuration* of standard program execution and $\langle\checkmark, St\rangle$ denotes a final *configuration*, where $\checkmark$ stands for the empty program. Note that we assume that every Java program properly terminates for each set of input data (i.e., we do not consider non-terminating programs, deadlocks, or runtime errors). We also assume deterministic Java programs, without threads or exceptions. $\mapsto_{\text{Java}}$ is the transition relation that describes any possible one-step transition between any two Java program states. An *execution* (or trace) of P_{Java} is a sequence $\langle P_{\text{Java}}, St_0\rangle \mapsto_{\text{Java}} \cdots \langle P_i, St_i\rangle \mapsto_{\text{Java}} \cdots \mapsto_{\text{Java}} \langle\checkmark, St_n\rangle$, which is simply denoted by $\langle P_{\text{Java}}, St_0\rangle \mapsto^*_{\text{Java}} \langle\checkmark, S_n\rangle$ if the intermediate states are irrelevant. We can also abbreviate $\langle\checkmark, S_n\rangle$ by $\langle S_n\rangle$.

We define program non–interference by using an equivalence $=_{Low}$ relationship between states [Sabelfeld and Myers, 2003; Volpano et al., 1996; Barbuti et al., 2002; Matos and Boudol, 2005]. Roughly speaking, non–

interference establishes that any two terminating runs of a program that start from indistinguishable initial states produce indistinguishable final states.

Definition 3 (State equality [Sabelfeld and Myers, 2003])**.** Given a Java program P_{Java}, two states St_1 and St_2 for P_{Java} are *indistinguishable* at the confidentiality level `Low`, written $St_1 =_{\texttt{Low}} St_2$, if for all $var \in Low(P_{\text{Java}})$, $St_1[var] = St_2[var]$.

What the attacker can see from a final state is determined by a relation $\approx_{\texttt{Low}}$. Two executions of a program P_{Java} are related by $\approx_{\texttt{Low}}$ if they are indistinguishable to the attacker [Sabelfeld and Myers, 2003]. The notion of non–interference is therefore parametric on $\approx_{\texttt{Low}}$. A program is non–interferent if, whenever different initial program states are indistinguishable at level `Low`, this implies that the corresponding final states are also indistinguishable at level `Low`.

Definition 4 (Non–interference [Sabelfeld and Myers, 2003])**.** A Java program P_{Java} is *non–interferent* if for every pair of different program initial states St_1 and St_2, and for their corresponding final program states St_1', St_2' such that $\langle P_{\text{Java}}, St_1\rangle \mapsto^*_{\text{Java}} \langle St_1'\rangle$ and $\langle P_{\text{Java}}, St_2\rangle \mapsto^*_{\text{Java}} \langle St_2'\rangle$, we have that $St_1 =_{Low} St_2$ implies $St_1' \approx_{Low} St_2'$.

In our formulation, we follow the standard approach in the literature that considers $St \approx_{Low} St'$ iff $St =_{Low} St'$. Then, the non–interference condition of Definition 4 is understood as the lack of any *strong dependence* [Sabelfeld and Myers, 2003] of `Low`-labeled variables on any of the `High`-labeled variables.

The following example illustrates two executions of a given program using the Java semantics of Chapter 3.

Example 12. Consider again the Java program of Example 11 and two program executions, respectively fed with values 5000 and 10000 for the initialization parameter `initbalance`. Note that the corresponding initial states are indistinguishable at the `Low` confidentiality level (e.g. the only `Low`-labeled variable, `extraService`, is set to `false` in both of them). The Maude command `search` provides built–in breadth-first search. We ask for the final Java program state of each execution trace (actually, in order to visualize the results, we show the output of `println` Java instructions). The Maude terms

`EX1-MAUDE` and `EX2-MAUDE` stand for the Java program with the corresponding initial call (for input values 5000 and 10000, respectively), which are compiled into a Maude expression.

```
search in PGM-SEMANTICS :
java((preprocess(EX1-MAUDE) noType . 'main < new string [i(0)] > noVal))
=>! JO:Output .
Solution 1 JO:Output --> pl(bool(false))
No more solutions.

search in PGM-SEMANTICS :
java((preprocess(EX2-MAUDE) noType . 'main < new string [i(0)] > noVal))
=>! JO:Output .
Solution 1 JO:Output --> pl(bool(true))
No more solutions.
```

If the attacker observes these two final states, she will appreciate the two different values for the variable `extraService`.

The standard JML [Leavens et al., 2006] property specification language has no constructs for expressing non–interference, hence existing Java verification tools that use standard JML do not support non–interference verification and certification. Nevertheless, the confidentiality aspect of non–interference is expressible using the JML specification pattern suggested in [Jacobs et al., 2005; Warnier, 2005] as an instrument for program verification using the theorem prover PVS. Unfortunately, this proposal abuses notation by identifying the confidentiality levels with the values of program variables, and it does not consider important Java features such as method calls and interruptions (`break`, `return` or `continue` statements) within conditional instructions and iterations. Moreover, a specification pattern for confidentiality cannot be created in all cases, as mentioned in [Warnier, 2005].

Although non–interference has not been considered in current PCC implementations, there are some proposals that are based on type systems for a subset of Java [Barthe et al., 2006], Java bytecode [Rose, 2003; Barthe and Rezk, 2005; Barthe et al., 2007b], and simple imperative languages [Volpano et al., 1996; Hunt and Sands, 2006; Beringer and Hofmann, 2007]. However, to the best of our knowledge none of these have yet been implemented.

In [Barthe et al., 2006], a type system is proposed as a basis for deriving a certifying compiler for a subset of Java source code with objects, inheritance, methods and simplified exceptions. In [Barthe et al., 2007b] is defined the first information flow type system for a sequential JVM-like language with

classes, objects, arrays, exceptions and method calls that guarantees non–interference in type checked programs. The soundness was proven by using the theorem prover Coq, and a certified lightweight bytecode verifier for information flow was extracted from the proof. The system follows the principles of standard bytecode verification and can be used in a standard Java security architecture within the JVM. The certified verifier could also be used as a PCC proof checker at the consumer's side, as it was already mentioned in the FPCC Subsection 1.1.3 on page 21.

Volpano et al. [Volpano et al., 1996] developed the first sound information-flow type system that can be used to check non–interference of programs written in a generic deterministic sequential imperative language.

In [Hunt and Sands, 2006], Hunt and Sands proposed a flow sensitive, dynamic type system that has not yet been implemented. It tracks syntactical dependences between program variables in a simple imperative language without objects or function calls.

To verify non–interferent source Java programs, there are other type-based proposals that also do not use JML to specify information flow policies, namely the Java extensions JFlow [Myers, 1999] and Jif [Myers et al., 2001]. JFlow and Jif are security-typed programming languages with support for enforcing information-flow and access control with dynamic label policies, at both compile time and run time. These compilers produce secure Java source code for verified programs. Dynamic labels are introduced in order to deal with program variables whose confidentiality labels are only known at run time. However, JFlow does not have a soundness proof [Myers, 1999], and the dynamic labels of Jif have not yet been demonstrate to enforce secure information flow [Zheng and Myers, 2007].

The work of Banerjee and Naumann [Banerjee and Naumann, 2002] proposed a type system with security levels as annotations for a Java-like language, to deal with non–interference with object aliasing, a problem that the aforementioned works have not considered. They introduce a new security level to constraint object field updating and evaluation. The type system assume that all classes and methods are public, and that all object fields are secret. This work considers sequential terminating programs with dynamic object creation, references (pointers), and mutable fields. However, it does not consider loops, threads and exceptions, and it is not implemented yet when this dissertation is written. By now, our proposal does not consider dynamic objects in order to deal with object confidentiality levels and object

aliasing in a proper way.

In [Banerjee and Naumann, 2005] Banerjee and Naumann extended the type system of its previous work [Banerjee and Naumann, 2002] with permissions, in order to consider an access policy with a stack-based access control mechanism. The classes have permissions. The permissions can be enabled and checked at run time. Methods have types that depend on the permissions authorized by the caller. As far as we know this proposal has not been implemented yet.

A flow-sensitive and termination-insensitive analysis for object-oriented programs based on Hoare logic and separation logic is proposed in Amtoft et al. [Amtoft et al., 2006]. This analysis considers pointer aliasing that can leak confidential information. The non–interference property is specified by using independence assertions that are written in JML. In order to compute post-conditions, the analysis uses an algorithm that is sound and complete given some assumptions, but it does not generate a program security proof.

Wasserrab et al. presented in [Wasserrab et al., 2009] the first machine-checked correctness proof for information-flow control that is based on program dependence graphs using static intraprocedural slicing. The proof is formalized in Isabelle/HOL. The analysis applies to deterministic terminating programs and is flow-sensitive, object-sensitive and context-sensitive. The machine-checked proof was instantiated for a simple imperative language with loops and for a subset of Jinja (a sequential definition of Java bytecode) [Klein and Nipkow, 2006], which must be manually annotated with security labels. The Jinja language has boolean and integer values (including null values) as well as references (including null references). Jinja is an expression-oriented language and it has object creation, casting, literal, binary operation, and method call expressions, as well as variable and field access, blocks, sequential composition, conditionals, while loops, and exception throwing and catching. Jinja does not include `break` neither `continue` statements. Also, method calls are not considered in the proposed validation methodology.

Bavera and Bonelli presented in [Bavera and Bonelli, 2008] a flow-sensitive type system for verifying non–interference of bytecode, where class fields may have different confidentality labels for different instance objects. This methodology does not consider method calls and it does not generate checkable proofs. Moreover, as is usually the case in type-based analysis, once the object fields and the variable labels are determined, they remain fixed throughout the analysis.

A proposal that deals with dynamic information-flow policies is in [Shroff et al., 2007]. This technique is based on runtime tracking of indirect dependencies between program points. While our confidentiality label tracking is also dynamic, our approach is based on static analysis rather than runtime monitoring, similarly to [Hunt and Sands, 2006; Jacobs et al., 2005].

As an important difference with respect to the related literature, our work considers global non–interference of complete Java classes. We do not need to explicitly state the confidentiality level for all program variables. Moreover, we provide a general and language-independent characterization as well as a formal and rigorous relation between the approximate properties and the security model.

Our global policies are very flexible since the security levels of object variables, local variables, and method parameters may change temporarily as in [Hunt and Sands, 2006; Jacobs et al., 2005; Barbuti et al., 2002; Francesco and Martini, 2007; Barthe et al., 2004].

6.3 The extended Rewriting Logic semantics of Java for non–interference

Goguen and Meseguer [Goguen and Meseguer, 1982] formalized non–interference of deterministic and terminating systems as a security property that is defined for pairs of system output traces that are indistinguisable for an observer. Non–interference is usually understood to be a security property and is therefore defined as a *hyperproperty* [Clarkson and Schneider, 2008] (i.e., a property defined on a set of sets of traces, see Figure 6.1). For instance, in Example 12, the verification process for non–interference should check the (possibly infinite) set of (possibly infinite) sets of program traces with indistinguishable initial and final configurations regarding the `Low` confidentiality program variables. Note that checking the final states issued from `EX1-MAUDE` and `EX2-MAUDE` is just one of the combinations to be analyzed. In this case, the sets of sets of traces are defined by the indistinguishability relationship among initial states and among final states at the `Low` confidentiality level (Definition 4).

In contrast, the verification process for a safety property should simply check the traces issuing from the (possibly infinite) set of initial configurations, which is much simpler (see Figure 6.2).

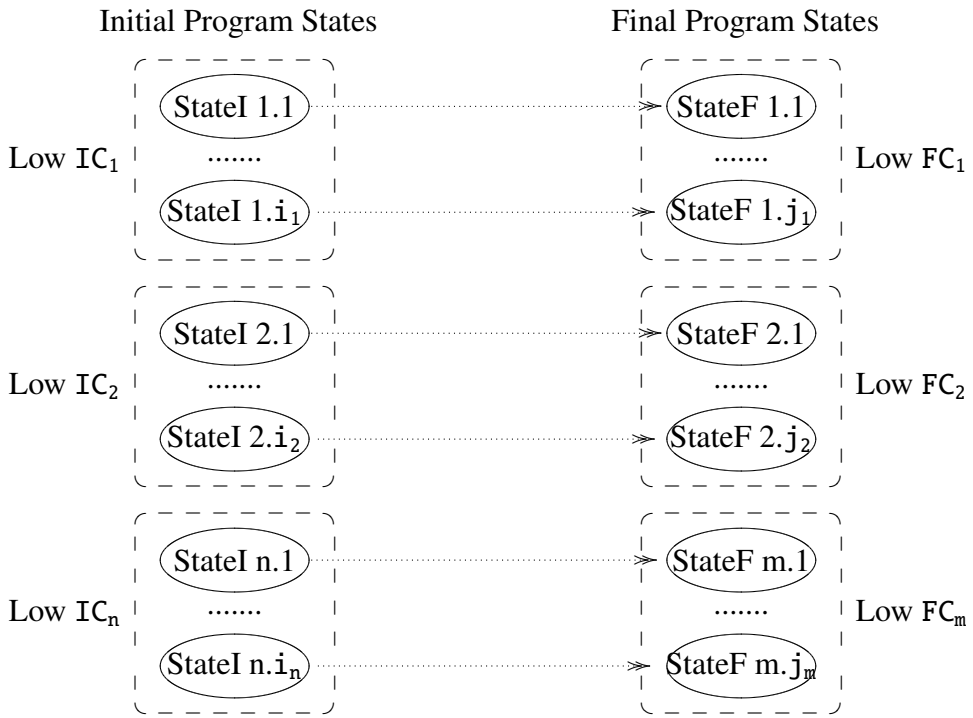

Figure 6.1: Sets of sets of Java program traces.

In this chapter, we prove non–interference as a safety property by instrumenting the Java semantics in order to dynamically keep track of the change of the confidentiality levels of program variables. Intuitively, the semantic instrumentation is defined as follows:

1. Attach a confidentiality level label to each memory location; this allows us to observe their confidentiality level at the final execution state.

2. Attach a confidentiality level label to the evaluation of program expressions; this allows us to know whether the evaluation of an expression involves high confidentiality data.

3. Associate a confidentiality level label to the evaluation of program statements, particularly those involving conditional expressions or guards; this allows us to determine whether the control flow at a given execution point depends on the actual value of high confidential variables. However, this label is not attached to each program statement; it is kept

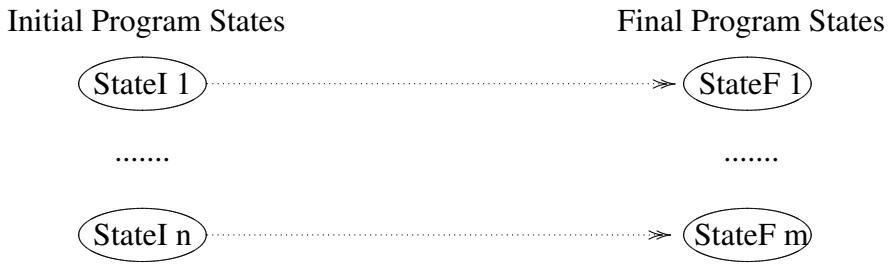

Figure 6.2: Sets of Java program traces.

as an extra attribute of a state in the extended Java semantics. This corresponds to the notion of a *context level* being updated after each evaluation step in order to take into account implicit information flows as in [Denning and Denning, 1977; Volpano et al., 1996; Barbuti et al., 2002; Sabelfeld and Myers, 2003; Jacobs et al., 2005; Hunt and Sands, 2006], which is introduced in the following example.

Example 13. Consider the following Java program `LeakClass` that is borrowed from [Warnier, 2005]. We endow it with the attached non–interference policy:

```
public class LeakClass {
     static int low=0, high; //@ setLabel(high, High);
     public static void main(String[] args) {
          high = Integer.parseInt(args[0]);
          while (high > 0) {
               high--;
               low++;}
     }
}
```

Here there is an illicit and implicit information flow from the `High`-labeled source variable `high` to the `Low`-labeled source variable `low`. For instance, when the variable `high` contains the value `0` or `1`, the variable `low` is assigned the value `0` and `1`, respectively. This implicit flow would be detected using the context label, which is set to `High` after evaluating the expression `high>0`, and which forces variable `low` to be set to `High` independently of the confidentiality level of the expression `low++`.

In contrast to the previous chapter where we studied local method properties, here we consider a global program property (i.e., we are able to ensure a non–interference policy at the final state of the whole Java program execution, which contains several methods, classes, and function calls). This global

```
eq k(i(I) -> K) lenv(CL) = k(<int(I),CL> -> K) lenv(CL) .
eq k(b(B) -> K) lenv(CL) = k(<bool(B),CL> -> K) lenv(CL) .
```

Figure 6.3: Extended equations for constant evaluation.

non–interference analysis requires to address the difficult (or costly) process of tracing the current confidentiality label of a memory location back to the point where this location was created.

1. We introduce an additional confidentiality label (`Low` $\gg$ `High`), which allows us to represent not only the current confidentiality level label of a memory location but also to keep track, at a global level, of hazardous transitions from an initial confidentiality level `Low` to `High`. Similarly, we introduce the confidentiality label (`High` $\gg$ `Low`), in order to avoid false positives where a `High`–labeled variable is updated with the value of a `Low`–labeled expression and then updated again with the value of a `High`–labeled expression.

2. In this chapter, we use the context level label during expression evaluation, as in [Barbuti et al., 2002], instead of using it when updating the value of a variable in memory, as in [Jacobs et al., 2005; Warnier, 2005; Hunt and Sands, 2006; Francesco and Martini, 2007], or when returning values as in [Francesco and Martini, 2007].

We describe the information-flow extended version of the rewriting logic semantics of Java by the rewrite theory $\mathcal{R}_{\text{Java}^{\text{E}}} = (\Sigma_{\text{Java}^{\text{E}}}, E_{\text{Java}^{\text{E}}}, R_{\text{Java}^{\text{E}}})$, $E_{\text{Java}^{\text{E}}} = \Delta_{\text{Java}^{\text{E}}} \uplus B_{\text{Java}^{\text{E}}}$ and its corresponding $\rightarrow_{\text{Java}^{\text{E}}}$ rewriting relation. In the new semantics, program data not only consist of standard concrete values but each value is actually decorated with its corresponding confidentiality label. Formally, we consider the label change $\mathit{LabelChange} = \{\texttt{Low} \gg \texttt{High}, \texttt{High} \gg \texttt{Low}\}$ so that the domain of program variables in the extended semantics is $\mathit{Value} \times (\mathit{Labels} \cup \mathit{LabelChange})$. We write `<Value,LValue>` for a pair consisting of a concrete value and its corresponding confidentiality label in $\mathit{Labels} \cup \mathit{LabelChange}$.

Thanks to the modularity of the rewriting logic approach to formalizing program semantics [Farzan et al., 2007], our changes to the semantics of Section 3 are incremental and minimal.

As Figures 6.3 and Figure 6.4 show, the evaluation of constants and variables uses the context label and the `join` operator. The `join` operator specification is shown in Figure 6.5. This means that the actual label of the constant

```
---First obtain location in store from variable name
eq k(Var -> K) env([Var, Loc] Env) = ... .
---Then obtain value stored in this location
eq k(#(Loc) -> K) store([Loc,<Val,LVal>] Store) lenv(CL)
 = k(<Val,LVal join CL> -> K) store([Loc,<Val,LVal>] Store) lenv(CL) .
```

Figure 6.4: Extended equations for variable content retrieval.

$\sqcup$	Low	High
Low	Low	High
High	High	High

$\sqcup$	Low $\gg$ High	High $\gg$ Low
Low $\gg$ High	Low $\gg$ High	High
High $\gg$ Low	High	High $\gg$ Low

Figure 6.5: Specification of the `join` operator.

```
eq   t(k(buildEnv(((T d(X)), Pl), (V, Vl)) -> K) env(Env) id(I) lenv(Lab) TC)
     store(store) nextLoc(I')
 =   t(k(buildEnv(Pl, Vl) -> K) env([X, l(I' + 1)] Env) id(I) lenv(Lab) TC)
     store([l(I' + 1), setTid(setAbsValue(X,V,Lab), I), I] store)
     nextLoc(I' + 1) [owise] .
```

Figure 6.6: Continuation-based equations for building the extended environment.

```
op setLabel : Var Value Label -> Value .
--- equation generated from JML-like annotation setValue(h, High)
eq setLabel('h, < Val, Label >, EnvLabel ) = < Val, High > .
--- Default case:
eq setLabel(Var, < Val, Label >, EnvLabel ) = < Val, Label join EnvLabel > [owise] .
```

Figure 6.7: Continuation-based equations for setting the initial variable confidentiality level.

and the actual label of the variable, both depend, not only on their own labels but also, on the context label at the program point where they are evaluated.

Figure 6.6 shows the extended equation that builds the environment for a new variable using the `setLabel` function that sets up the initial confidentiality level of the variable with the label given within the corresponding `setLabel` JML-like annotation, if any. Figure 6.7 shows the specification of the `setLabel` function. Figure 6.8 shows the specification of the Java + binary operator that also uses the `join` operator. The extended specification of the Java <= operator is shown in Figure 6.9. Note that the specification of binary operators are very similar. The label of the resulting value is the `join` of the operand's labels. Figure 6.10 shows the specification of the Java post-increment unary operator. Note that this unary operator updates the involved variable but it does not change its confidentiality label.

```
eq k((<int(I),Lab1>, <int(I'),Lab2>) -> (+ -> K))
 = k(<int(I + I'),Lab1 join Lab2> -> K) .
```

Figure 6.8: Equations of extended Java + operator.

```
eq k((< int(I), Lab1  >,< int(I'), Lab2 >) -> (<= -> K))
 = k(< bool(I <= I'), Lab1 join Lab2 > -> K)  .
```

Figure 6.9: Equations of extended Java <= operator.

```
eq k(< int(I), Lab > -> ++'(L) -> K)
  = k([< int(I + 1),Lab > -> L] -> (< int(I), Lab > -> K)) .
```

Figure 6.10: Equations of extended Java ++ post-increment operator.

```
---Obtain variable location and evaluate expression
eq k(Var = E -> K) env([Var, Loc] Env) = ... .
---Once the expression is computed, assign to location
eq k(<Val,LVal> -> =(L) -> K) = k([<Val,LVal> -> L] -> (<Val,LVal> -> K )) .
---General procedure to update the memory
eq k([<Val,LVal> -> Loc] -> K) store([Loc,<Val',LVal'>] ST)
 = k(K) store([Loc,<Val, LVal' >>> LVal >] ST) .
```

Figure 6.11: Extended equations for the Java assignment operator.

LVal'	LVal	LVal' $\ggg$ LVal
L	L	L
Low	High	Low $\gg$ High
High	Low	High $\gg$ Low
$L_1 \gg L_2$	L_1	L_1
$L_1 \gg L_2$	L_2	$L_1 \gg L_2$
L	$L_1 \gg L_2$	L $\ggg L_2$
$L_1 \gg L_2$	$L_3 \gg L_4$	$(L_1 \gg L_2) \ggg L_4$

$L, L_1, L_2, L_3, L_4 \in \{\text{Low}, \text{High}\}, L_1 \neq L_2, L_3 \neq L_4$

Figure 6.12: Updating memory locations.

As Figure 6.11 shows, the assignment computes the new confidentiality label in terms of the previous label at the memory location, namely `NewVal = LVal' >>> LVal`. The new operator $\ggg$ is defined in Figure 6.12 following [Alba-Castro et al., 2010a].

The context label can only change due to conditional control flow statements. According to [Denning and Denning, 1977; Barbuti et al., 2002; Sabelfeld and Myers, 2003; Jacobs et al., 2005; Hunt and Sands, 2006], the evaluation of its boolean guards returns a confidentiality level that is associ-

ated to the resulting `true` or `false` value, and it may also return a modified context label. The semantic equations for the `if-then-else` operator of Figure 3.9 need some slight revision, which is motivated by the following example.

Example 14. Consider the following Java class, where the value computed for the variable `low` does not actually depend on the value of the high confidentiality variable `high` (which only affects the temporal variable `aux`). This program does fulfill the non–interference policy at the final state, which can be proved by using our non–interference verification methodology.

```
class NoLeakclass {
     static int low=0, high; //@ setLabel(high, High);
     public static void main(String[] args) {
         high = Integer.parseInt(args[0]);
         int aux=0;
         if (high > 2) aux = 1;
             else aux = 0;
         low = 0;
     }
}
```

In order to avoid false positives during the evaluation of conditional statements, we dynamically restore the previous context label after its execution. The extended semantic equations for the `if-then-else` are shown in Figure 6.13, where a new continuation symbol `restoreLEnv` is used to restore the previous confidentiality label. This allows us to verify safe programs that have temporary breaches, i.e. the implicit flow from variable `high` to variable `aux` in Example 14, as the dynamic typing approaches [Jacobs et al., 2005; Warnier, 2005; Hunt and Sands, 2006] and the self-composition type system approach of Barthe et al. [Barthe et al., 2004]. This example cannot be verified by the type system in [Volpano et al., 1996]

However, restoring the previous context label has to be carefully considered in the presence of `break` or `continue` statements within a loop, since they can abruptly change the information flow as shown in the following example.

Example 15. Consider a variation of Example 13 where the while loop has a bogus guard together with a `break` statement to exit the loop:

```
--- Evaluates boolean expression keeping the then and else statements
ceq k((if E S else S') -> K) lenv(CL)
  = k(E -> (if(S, S') -> restoreLEnv(CL) -> K)) lenv(CL)
 if not break-or-continue(S) and not break-or-continue(S') .
ceq k((if E S else S') -> K) lenv(CL) = k(E -> (if(S, S') -> K)) lenv(CL)
 if break-or-continue(S) or break-or-continue(S') .
eq k(<bool(true),LVal> -> (if(S, S') -> K)) lenv(CL) = k(S -> K) lenv(CL join LVal) .
eq k(<bool(false),LVal> -> (if(S, S') -> K)) lenv(CL) = k(S' -> K) lenv(CL join LVal) .
--- New equation to restore previous context label
eq k(restoreLEnv(CL) -> K) lenv(CL') = k(K) lenv(CL) .
```

Figure 6.13: Extended equations for the if-then-else statement.

```
--- Stack loop and transform while expression into while continuation
eq k((while E S) -> K) lstack(Lstack) lenv(CL)
 = k(while(E,S) -> restoreLEnv(CL) -> popLStack -> K) lstack(while(E,S) -> K, Lstack)
   lenv(CL) .
```

Figure 6.14: Extended equations for the while statement.

```
public class Leakclass {
     static int low=0, high; //@ setLabel(high, High);
     public static void main(String[] args) {
          high = Integer.parseInt(args[0]);
          int aux=0;
          while (true) {
               high--;
               low++;
               if (high == 0) break;
          }
     }
}
```

As in Example 13, when the while loop ends, the variable `low` has the initial value of the variable `high`. Whenever `high` $\neq 0$, the `break` statement is not executed. In this case, the conditional guard uses `High`-labeled data, and the conditional statement should not restore the previous context label. In other words, the critical component here is not the `break` statement but rather the else branch that does not contain the `break`.

In order to solve this problem, the equations in Figure 6.13 check whether each of the two branches of a conditional statement contains a `break` or `continue` statement and no other conditional statement or `while` loop in between. If there is such a statement, `restoreLEnv` is not used. This case was not considered in [Jacobs et al., 2005; Warnier, 2005] nor in [Hunt and Sands, 2006], which do not consider neither `break` nor `continue` statements.

Since while statements were expressed in terms of if-then-else statements, they need a slight extension in order to introduce the `restorelEnv` continuation (shown in Figure 6.14).

The semantic specification of the `break` statement stays the same as shown in Figure 3.11: the context label `lenv(CL)` is not modified and the `restoreLEnv` expression introduced by the while statement is removed.

Method invocation propagates the context label without changes as proposed in [Jacobs et al., 2005], where it was not implemented. Thus, the standard specifications of instance method invocation (Figure 3.12) and return statements (Figure 3.13) can be used here with no modification. Hunt and Sands work [Hunt and Sands, 2006] does not considers objects, nor method invocations.

We do not consider exceptions, nor Hunt and Sands work [Hunt and Sands, 2006], neither Jacobs et al. and work [Jacobs et al., 2005]. However, exceptions could by handled in a similar way as `break` and `continue` statements.

6.3.1 Proving non–interference as a safety property

Now, we are ready to formulate a novel characterization of non–interference that allows us to check it as a property that is verified for each possible execution trace instead of being verified for each set of indistinguishable execution traces. We state non–interference as a safety property in Definition 5.

Darvas et al. [Darvas et al., 2005] and Barthe et al. [Barthe et al., 2004] approaches used self-composition of programs for proving non–interference as a safety property. Barthe et al. [Barthe et al., 2004] developed a methodology for proving non–interference of deterministic terminating programs in an imperative language with loops, conditionals, and mutable data structures (i.e. objects). Their methodology relies on using Hoare logic and separation logic, and handles non–interference as a safety property by using program self–composition with variable renaming (i.e., they compose a program with a copy of itself without sharing memory positions). This proposal is complete and sound, but the criterion is undecidable.

Some sophisticated non–interference policies can be expressed by using the JML extensions of the Krakatoa Java verification tool [Dufay et al., 2005]. These JML extensions were developed for Hoare-style assertions regarding program self-composition [Barthe et al., 2004]. This means duplicating the code of the program and makes it necessary to distinguish the same program variables in its two runs. The JML extensions are used to express non–interference pre– and post–conditions, but they do not handle confidentiality

labels of program variables explicitly. The method assumes that all the variables annotated with the extended JML assertions called "`ni1`" and "`ni2`", are labeled `Low`. This means that the confidentiality level of all `Low` variables have to be explicitly stated, whereas in our framework it isn't because of the default policy.

Hunt and Sands work also proved non–interference as a safety property, but by using flow sensitive security types [Hunt and Sands, 2006].

Definition 5 (Strong Non–Interference)**.** A Java program P_{Java} is *strongly non–interferent* for a given labeling function if, for every extended initial state St_1^E and for its corresponding final program state St_2^E given by $\langle P_{\text{Java}}, St_1^E\rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St_2^E\rangle$, we have that for each $var \in Low(P_{\text{Java}})$, $St_2^E[var] = \langle Val, \texttt{Low}\rangle$ for some value Val.

Since in our model, a public variable can only have the label `Low` or the label `Low` $\gg$ `High`, this means that in the extended execution of a program that is not strongly non–interferent, the label of at least one program variable is `Low` $\gg$ `High`. Given an initial state St and a given labeling function, we denote the corresponding extended state by St^E.

Theorem 3 (Strong non–interference soundness)**.** Consider a Java program P_{Java} and two initial states St_1 and St_2 such that $St_1 =_{Low} St_2$. Consider the two corresponding final program states St_1' and St_2' given by $\langle P_{\text{Java}}, St_1\rangle \mapsto^*_{\text{Java}} \langle St_1'\rangle$, $\langle P_{\text{Java}}, St_2\rangle \mapsto^*_{\text{Java}} \langle St_2'\rangle$. If there exists $var \in Low(P_{\text{Java}})$ such that $St_1'[var] \neq St_2'[var]$, then $\langle P_{\text{Java}}, St_1^E\rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St_1'^E\rangle$ and $St_1'^E[var] = \langle Val, \texttt{Low} \gg \texttt{High}\rangle$ for a value Val.

Proof. Consider the two traces $\mathcal{D}_1 : \langle P_{\text{Java}}, St_1\rangle \mapsto^*_{\text{Java}} \langle St_1'\rangle$ and $\mathcal{D}_2 : \langle P_{\text{Java}}, St_2\rangle \mapsto^*_{\text{Java}} \langle St_2'\rangle$. Let $\{var_1, \ldots, var_k\} \subseteq Low(P_{\text{Java}})$ be those variables such that $St_1'[var_i] \neq St_2'[var_i]$ for all $1 \leq i \leq k$. Since we assume $k > 0$, then there is at least one of those variables (say var_1) and an assignment statement $var_1 = E_1$ that is executed at least once in one of the two traces (say $\mathcal{D}_1$). Let n be the total number of assignments in $\mathcal{D}_1$ to variables $\{var_1, \ldots, var_k\}$. Note that n is finite since execution traces are finite because of the termination assumption, and that $k \leq n$, i.e. all k variables were updated at least once in one of the traces. The result is reformulated as Proposition 1, where $PI(n, k)$ is the predicate that relates n and k. The proof proceeds by induction on n.

Proposition 1. $PI(n, k)$:

$\mathcal{D}_1$ has n assignments to the k variables $\{var_1, \ldots, var_k\}$ [1], and it holds that for all $j \in \{1, \ldots, k\}$, $St'^{E}_{1}[var_j] = \langle Val_j, \texttt{Low} \gg \texttt{High} \rangle$ for a value Val_j.

1. ($n = 1$ and $k = 1$) Let us consider the last execution step in $\mathcal{D}_1$ where the assignment $var_1 = E_1$ is executed. Then, it may happen that the assignment $var_1 = E_1$ is also executed in $\mathcal{D}_2$, or not. We consider these two cases separately.

   (a) If $var_1 = E_1$ is also executed in $\mathcal{D}_2$, then $St'_1[var_1] \neq St'_2[var_1]$ implies that the values for E_1 are different in the two traces. Thus, expression E_1 must contain at least one variable var' such that the actual values of var' are different in the two traces when the considered assignments to var_1 are executed. Since $St'_1[var'] \neq St'_2[var']$ and $n = 1$, then $var' \notin Low(P_{\text{Java}})$. Therefore var' is a `High` confidentiality variable, hence it has a `High` label in our extended semantics. This means that the label $\texttt{Low} \gg \texttt{High}$ is assigned to variable var_i (according to Figure 6.12) in $\mathcal{D}_1$, then $St'^{E}_{1}[var_1] = \langle Val_1, \texttt{Low} \gg \texttt{High} \rangle$ for a value Val_1, and $PI(1, 1)$ follows.

   (b) If $var_1 = E_1$ is not executed in $\mathcal{D}_2$, then $St'_1[var_1] \neq St'_2[var_1]$ implies that the execution of this last assignment statement $var_1 = E_1$ in $\mathcal{D}_1$ is conditioned to the result of a boolean expression containing `High` confidentiality variables that guard a conditional (or `while` loop) statement so that the assignment is executed in $\mathcal{D}_1$ and not in $\mathcal{D}_2$. Then, the assignment statement $var_1 = E_1$ in $\mathcal{D}_1$ was executed either (i) within the then or else branch of an `if-then-else` Java statement (recall that while loops are expressed as `if-then-else` statements), (ii) within the then branch of an `if-then` Java statement, or (iii) after evaluating a conditional expression within a while loop that includes a `break` expression. Note that no other case can generate an interference condition. In all three cases, our extended semantics assigns a `High` label to the boolean guard expression of such a conditional expression, and the context label is set to `High` (according to Figures 6.13 and 6.14) before the expression E_1 is evaluated in the

[1] These k variables $\{var_1, \ldots, var_k\}$ satisfy the *interference* condition, i.e. $St'_1[var_i] \neq St'_2[var_i]$ for all $1 \leq i \leq k$.

statement $var_1 = E_1$. Note that in case (iii), the conditional expression propagates the `High` context label outside itself (according to Figure 3.11), i.e. the conditional does not restore the previous context label precisely to record that even if sequence $\mathcal{D}_1$ does not execute the `break` statement, another possible trace (e.g. $\mathcal{D}_2$) can do it. Finally, in all three cases, the expression E_1 is evaluated within a `High`–labeled context and then the label `Low` $\gg$ `High` is assigned to variable var_1, independently of whether expression E_1 manipulates `High` confidential data or not. This means that $St_1'^{E}[var_1] = \langle Val_1, \texttt{Low} \gg \texttt{High}\rangle$ for a value Val_1; then, $PI(1,1)$ holds.

2. ($n > 1$) Let us consider the *last* execution step in $\mathcal{D}_1$ where the assignment $var_i = E_i$ is executed, with $1 \leq i \leq k$. We split into two cases.

   (a) If $var_i = E_i$ is also the last assignment of variables $\{var_1, \ldots, var_k\}$ executed in $\mathcal{D}_2$, then $St_1'[var_i] \neq St_2'[var_i]$ implies that the values for E_i are different in the two traces. Thus, expression E_i must contain at least one variable var' such that the actual values of var' are different in the two traces when the considered assignments to var_i are executed. Then, let us consider whether $var' \in \{var_1, \ldots, var_k\}$ or not. If it is, consider the replacement of the last assignment $var_i = E_i$ by an empty statement in both traces $\mathcal{D}_1$ and $\mathcal{D}_2$, both traces will have $n-1$ assignments to the k variables $\{var_1, \ldots, var_k\}$, and the program will be still interferent. There are two cases:

      i. var_i and var' are the same variable. By induction hypothesis, it can be assumed that $PI(n-1, k)$ holds for the modified program with $n-1$ assignments to the k variables (Proposition 1). If the empty statements are replaced in both traces by the assignment $var_i = E_i$, then the original program traces $\mathcal{D}_1$ and $\mathcal{D}_2$ with n assignments are obtained, and the variable var_i is updated by the introduced assignment statement, but its label `Low` $\gg$ `High` is not modified. This is because the label of expression E_i is either `High` (if E_i contains a secret variable) or `Low` $\gg$ `High` (if E_i contains variable var_i).

ii. var_i and var' are different variables. By induction hypothesis, it can be assumed that the Proposition 1 holds for the modified program with $n-1$ assignments to k' variables, $PI(n-1,k')$, where

A. $k' = k-1$, when the deleted assignment $var_i = E_i$ is the unique assignment that updates variable var_i. In this case, the set of variables that satisfies the interference condition does change, because variable var_i does not satisfy it in the program without the last assignment over it. The k' variables that satisfies the interference condition, are such that $\{var_1,\ldots,var_{k'}\} = \{var_1,\ldots,var_k\} - \{var_i\}$.

B. $k' = k$, otherwise. The set of variables that satisfies the interference condition does not change.

If the empty statements are replaced in both traces by the assignment $var_i = E_i$, then the original program traces $\mathcal{D}_1$ and $\mathcal{D}_2$ with n assignments are obtained, and the variable var_i is updated as before in both cases 2(a)iiA and 2(a)iiB. In case 2(a)iiA, the label of variable var_i is also updated from `Low` to `Low` $\gg$ `High`. In case 2(a)iiB, the label `Low` $\gg$ `High` of variable var_i is not modified (similarly to case 2(a)i).

If $var' \notin \{var_1,\ldots,var_k\}$, then var' is a `High` confidentiality variable and it has a `High` label in our extended semantics.

In all cases, the label `Low` $\gg$ `High` is assigned to variable var_i (according to Figure 6.12). This means that in all cases $St_1'^{E}[var_i] = \langle Val_i, \texttt{Low} \gg \texttt{High}\rangle$ for a value Val_i, and thus $PI(n,k)$ holds for the original program, and the result follows.

(b) If $var_i = E_i$ is not the last assignment of variables $\{var_1,\ldots,var_k\}$ executed in $\mathcal{D}_2$, then either there is no such an assignment in $\mathcal{D}_2$ to variables $\{var_1,\ldots,var_k\}$, or the last assignment in $\mathcal{D}_2$ has the form $var_i = E'$, with E' different from E_i, or it affects a variable var'' that is different from var_i. All three cases imply that the execution of the last assignment statement $var_i = E_i$ in $\mathcal{D}_1$ is conditioned to the result of a boolean expression containing `High` confidentiality variables that guard a conditional (or `while` loop) statement so that such assignment is executed in $\mathcal{D}_1$ and not in $\mathcal{D}_2$. Then this case is perfectly similar to case (1)(b) above, so that in

all cases $St_1'^{E}[var_i] = \langle Val_i, \texttt{Low} \gg \texttt{High} \rangle$ for a value Val_i, and thus $PI(n,k)$ holds for the original program, and the conclusion follows.

□

From Theorem 3 we derive that strong non–interference implies non–interference, as given by the following result.

Theorem 4 (Strong Non–Interference Soundness)**.** Given a Java program P_{Java}, if P_{Java} is strongly non–interferent (Definition 5), then P_{Java} is non–interferent (Definition 4).

Proof. (By contradiction) Assume that program P_{Java} is strongly non–interferent and that P_{Java} is interferent. Since P_{Java} is strongly non–interferent, for every extended initial state St^{E} and for its corresponding final program state $St^{E'}$ given by $\langle P_{\text{Java}}, St^{E} \rangle \mapsto^{*}_{\text{Java}^{\text{E}}} \langle St^{E'} \rangle$, we have that for all $var \in Low(P_{\text{Java}})$, $St^{E'}[var] = \langle Val, \texttt{Low} \rangle$ for a value Val. By Lemma 3 and the assumption that P_{Java} is interferent we have that $St^{E'}[var] = \langle Val, \texttt{Low} \gg \texttt{High} \rangle$ for a value Val, hence P_{Java} is not strongly non–interferent, contradicting the hypothesis.
□

Foccardi and Gorrieri [Focardi et al., 1994] defined a security–based notion of non–interference stronger than the notion of Goguen and Meseguer [Goguen and Meseguer, 1982], that also considers pairs of system input/output traces, but with both secret inputs and outputs, instead of secret outputs only. In contrast to [Focardi et al., 1994], our safety-based notion of strong non–interference only considers secret outputs, similarly to [Goguen and Meseguer, 1982].

The following example illustrates the mechanization of our verification methodology.

Example 16. Consider again the Java program of Example 11. Now, we compute the final state in the extended Java program execution for `EX1-MAUDE` (for simplicity we show only the value of variable `extraBalance`).

```
search in PGM-SEMANTICS-EXTENDED :
java((preprocess(EX1-MAUDE) noType .  'main < new string [i(0)] > noVal)) =>! M:Store .
Solution 1 M:Store --> store([l(6),<bool(false),Low >> High>] ...)
No more solutions.
```

The execution for `EX2-MAUDE` will also contain the label `Low` $\gg$ `High` for variable `extraBalance`.

In other words, we transform non–interference into a stronger property which can be effectively checked in the extended semantics. Although we consider only two security levels, our methodology can easily been extended to the multilevels of confidentiality of [Hunt and Sands, 2006; Barthe et al., 2007b]. Moreover, we have shown that our analysis can achieve more precision than traditional, type-based approaches, thanks to the combination of static analysis and dynamic labeling.

Obviously, we are not able to certify the security of all the programs that are secure, as shown in Example 17.

Example 17. Consider the following Java program borrowed from [Warnier, 2005].

```
class NoLeakclass {
     static int low=0, high; //@ setLabel(high, High);
     public static void main(String[] args) {
          high = Integer.parseInt(args[0]);
          low = high;
          low = low - high;
     }
}
```

Apparently, there is an explicit flow from variable `high` to variable `low` through the two assignment statements. However for any execution, when program execution ends, the value of variable `low` is always `0` so that the variable `low` does not depend on the variable `high`. According to Definition 4, the program is non–interferent. However, we give a false positive by using our notion of strong non–interference since the assignment "`low = high`" assigns to the variable `low` a high confidentiality label `Low` $\gg$ `High` and the last statement "`low = low − high`" does not revert the label back to `low`.

The program of Example 17 cannot be verified by traditional type inference approaches [Zanotti, 2002; Avvenuti et al., 2003] either, since they fail to verify (type check) any program with temporary breaches, e.g. Examples 14 and 17 above, whereas Example 14 is effectively verified by using our methodology.

The self-composition method in [Barthe et al., 2004] can verify non–interference of secure programs with temporary breaches such as "`low=high; low=2`", Example 14 and Example 17 above, whereas imprecise conservative type systems like the proposed in [Volpano et al., 1996] cannot.

6.4 The extended abstract Rewriting Logic semantics of Java

The extended, instrumented Java semantics defined so far allows us to develop a technique for proving non–interference. However, this technique is still not feasible in general because there are too many possible initial states to consider for the safety property to be checked. In the following, we develop an abstract, rewriting logic Java semantics that allows us to statically analyze global non–interference. Similar to Chapter 5, the purpose of the abstract semantics is to correctly approximate the extended computations in a finite way. But, differently than Chapter 5, the abstract values used in this chapter do not depend on the concrete values given by the standard semantics of Chapter 3.

Given the extended Java semantics of Section 6.3, where there are concrete labeled values, we simply get rid of the values in the abstract semantics, and use their confidentiality labels as the abstract values instead.

In the following, we develop an abstract version of the extended rewriting logic semantics of Java developed in Section 6.3, which we describe by the rewrite theory $\mathcal{R}_{\text{Java}^\#} = (\Sigma_{\text{Java}^\#}, E_{\text{Java}^\#}, R_{\text{Java}^\#})$, $E_{\text{Java}^\#} = \Delta_{\text{Java}^\#} \uplus B_{\text{Java}^\#}$ and its corresponding $\rightarrow_{\text{Java}^\#}$ rewriting relation. As in Section 6.3, our approach for the abstract Java semantics consists of modifying the original theory $\mathcal{R}_{\text{Java}^\text{E}}$ (taking advantage of its modularity) by abstracting the domain to $\mathit{Labels} \cup \mathit{LabelChange}$ and introducing approximate versions of the Java constructions and operators tailored to this domain.

Our abstraction function $\alpha : \wp(\mathsf{State}^E) \rightarrow \wp(\mathsf{State}^E)$ is a simple homomorphic extension to sets of states of the function $\alpha : \wp(\mathsf{Value}^E) \rightarrow \wp(\mathsf{Value}^E)$, where $\mathsf{Value}^E = \mathsf{Value} \times (\mathit{Labels} \cup \mathit{LabelChange})$, and given $S\,Value^E = S\,Value \times \{\mathtt{SecLab}\}$, with $S\,Value \subseteq \mathsf{Value}$, $\alpha(S\,Value^E) = \mathsf{Value} \times \{\mathtt{SecLab}\}$; for instance, $\alpha(\{\mathtt{int(2)}\} \times \{\mathtt{SecLab}\}) = \mathsf{Value} \times \{\mathtt{SecLab}\}$. This means that we abstract the actual values $S\,Value, \mathtt{int(2)}$ of data, and only keep their sorts $(\mathsf{Value}, \mathsf{Int})$; this is actually equivalent to discard the actual value of data by using the projection function $2nd : \mathsf{Value} \times (\mathit{Labels} \cup \mathit{LabelChange}) \rightarrow (\mathit{Labels} \cup \mathit{LabelChange})$.

Figures 6.15 and 6.16 show the abstract equations for constant evaluation and variable content retrieval, that correspond to the concrete extended equations of Figures 6.3 and 6.4, respectively. The specification of the abstract addition and less-or-equal operators, corresponding to the extended ones of

```
eq k(i(I) -> K) lenv(CL) = k( CL -> K) lenv(CL) .
eq k(b(B) -> K) lenv(CL) = k( CL  -> K) lenv(CL) .
```

Figure 6.15: Extended abstract equations for constant evaluation.

```
---First obtain location in store from variable name
eq k(Var -> K) env([Var, Loc] Env) = ... .
---Then obtain value stored in this location
eq k(#(Loc) -> K) store([Loc, LVal ] Store) lenv(CL)
 = k( (LVal join CL) -> K) store([Loc, LVal ] Store) lenv(CL) .
```

Figure 6.16: Extended abstract equations for variable content retrieval.

```
eq k((Lab1 , Lab2 ) -> (+ -> K)) = k( ( Lab1 join Lab2 ) -> K) .
```

Figure 6.17: Abstract equations of extended Java + operator.

```
eq k(( Lab1 , Lab2 ) -> (<= -> K)) = k( ( Lab1 join Lab2 ) -> K)  .
```

Figure 6.18: Abstract equations of extended Java <= operator.

```
eq k( Lab  -> ++'(L) -> K) = k([ Lab  -> L] -> ( Lab  -> K)) .
```

Figure 6.19: Abstract equations of extended Java ++ post-increment operator.

Figures 6.8 and 6.9 are shown in Figures 6.17 and 6.18, respectively. Figure 6.19 shows the specification of the abstract unary post-increment operator whose concrete extended specification was given in Figure 6.10. The extended specification for building the environment given in Figure 6.6 can be used here with no changes, but the specification of the `setLabel` function regarding abstraction need to be modified as shown in Figure 6.20. The variable assignment abstract specification corresponding to the extended equations of Figure 6.11 is shown in Figure 6.21.

Because of abstraction, some extended concrete equations are such that their corresponding abstract versions are not confluent: the equations have left-hand sides that are equal but their right-hand sides are not. This means that the program can behave concurrently when abstractly interpreted. In this case, as is Section 5.2, we use concurrency at the Maude level, using rules instead of equations. Despite the fact that our extended Java semantics contains only equations and no rules, the abstract Java semantics does contain rules in $R_{\text{Java}^{\#}}$ to reflect the different possible evolutions of the system.

The abstract rules associated to two of the equations of the extended semantics of the `if-then-else` statement are shown in Figure 6.22.

```
op setLabel : Var Value Label -> Value .
--- equation generated from JML-like annotation setValue(h, High)
eq setLabel('h, Label , EnvLabel ) =  High .
--- Default case:
eq setLabel(Var,  Label , EnvLabel  ) =  Label join EnvLabel   [owise] .
```

Figure 6.20: Continuation-based equations for setting initial variable confidentiality level.

```
---Obtain variable location and evaluate expression
eq k(Var = E -> K) env([Var, Loc] Env) = ... .
---Once the expression is computed, assign to location
eq k( LVal -> =(L) -> K)= k([ LVal -> L] -> ( LVal -> K )) .
---General procedure to update the memory
eq k([ LVal  -> Loc] -> K) store([Loc, LVal'] ST) = k(K) store([Loc,LVal' >>> LVal] ST) .
```

Figure 6.21: Abstract equations for the Java assignment operator.

```
rl k(LVal -> (if(S,S') -> K)) lenv(CL) => k(S -> K) lenv(CL join LVal) .
rl k(LVal -> (if(S,S') -> K)) lenv(CL) => k(S' -> K) lenv(CL join LVal) .
```

Figure 6.22: Abstract rules for the if-then-else statement.

Now, we are ready to formalize the abstract rewriting relation $\rightarrow_{\text{Java}^\#}$, which intuitively develops the idea of applying only one rule or equation from the concrete Java semantics to an abstract Java state while exploring the different alternatives in a non-deterministic way. By abusing of notation, we denote the abstraction of a rule $\alpha(\{l\}) \rightarrow \alpha(\{r\})$ by $\alpha(\{l\} \rightarrow \{r\})$. $\mathcal{P}_{\text{Java}}$ denotes the sort of Java programs P_{Java} (i.e. $P_{\text{Java}} \in \mathcal{P}_{\text{Java}}$).

Definition 6 (Abstract rewriting)**.** We define the abstract rewriting relation $\rightarrow_{\text{Java}^\#} \subseteq (\mathcal{P}_{\text{Java}} \times \wp(\mathsf{State}^E)) \times (\mathcal{P}_{\text{Java}} \times \wp(\mathsf{State}^E))$ by $\langle P_{\text{Java}_1}, S\,S\,t_1\rangle \rightarrow_{\text{Java}^\#} \langle P_{\text{Java}_2}, S\,S\,t_2\rangle$ if $\exists u \in S\,S\,t_1, \exists v \in S\,S\,t_2$ s.t. $\langle P_{\text{Java}_1}, u\rangle \rightarrow_{\text{Java}^{\text{E}}} \langle P_{\text{Java}_2}, v\rangle$.

Note that in particular this applies to $S\,S\,t_1 = \alpha(\{u\})$ and $S\,S\,t_2 = \alpha(\{v\})$, given $u \in \alpha(\{u\})$ and $u \in \alpha(\{u\})$. We denote by $\rightarrow^*_{\text{Java}^\#}$ the extension of $\rightarrow_{\text{Java}^\#}$ to multiple rewrite steps.

Our abstraction consists of transforming equations into rules and getting rid of the value component of states. In order to guarantee that the abstract semantics correctly (over-)approximates the extended semantics, we need to prove that:

1. The abstraction is correct regarding the relation between the concrete and abstract domains of program states, based on the abstraction function α. This is done by proving that this abstraction function α and

a corresponding concretization function γ, both constitute a Galois insertion [Cousot and Cousot, 1979; Cousot and Cousot, 2002; Cousot, 2004]. This statement is presented in Theorem 5 below, with the corresponding proof.

2. All extended program traces have corresponding abstract program traces, such that no extended program trace is disregarded. The transformation of a set of equations (which are confluent and terminating modulo axioms) into rules preserves the execution traces. The removal of the value component of states does not eliminate execution traces neither. This formal statement is presented in Theorem 6 (on page 123). This theorem and its proof are very similar to Theorem 2 in Chapter 5.

Recall the abstraction function $\alpha : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$, that is a simple homomorphic extension to sets of states of the function $\alpha : \wp(\mathsf{Value} \times SecLabels) \to \wp(\mathsf{Value} \times SecLabels)$, where $SecLabels$ denotes the sort $Labels \cup LabelChange$. Given $SValue^E \in \wp(\mathsf{Value} \times SecLabels)$, i.e. $SValue^E = \{\langle Value, \mathtt{SecLab}\rangle\}$ with $\mathtt{SecLab} \in \{\mathtt{Low}, \mathtt{High}, \mathtt{Low} \gg \mathtt{High}, \mathtt{High} \gg \mathtt{Low}\}$, $\alpha(SValue^E) = \mathsf{Value} \times \{\mathtt{SecLab}\}$; note that all tuples of set $SValue^E$ have the same and unique confidentiality level, w.r.t. the confidentiality level of the corresponding variable; we denote the set $\mathsf{Value} \times \{\mathtt{SecLab}\} \in \wp(\mathsf{Value} \times SecLabels)$ by $\#\mathtt{SecLab}$, such that $\alpha(\{\langle \mathtt{Val}, \mathtt{SecLab}\rangle\}) = \#\mathtt{SecLab}$; this means, for instance, that $\alpha(\{\langle \mathtt{Val}, \mathtt{High}\rangle\}) = \#\mathtt{High}$.

The corresponding concretization function $\gamma : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$, is the homomorphic extension to sets of states of the function $\gamma : \wp(\mathsf{Value} \times SecLabels) \to \wp(\mathsf{Value} \times SecLabels)$, given $SSecLabel \in \wp(\mathsf{Value} \times SecLabels)$ such that $SSecLabel = \mathsf{Value} \times \{\mathtt{SecLab}\}$, $\gamma(SSecLabel) = SSecLabel$; $\gamma(SSecLabel) = \mathsf{Value} \times \{\mathtt{SecLab}\}$, e.g. $\gamma(\#\mathtt{SecLab}) = \#\mathtt{SecLab}$, for instance, $\gamma(\#\mathtt{High}) = \mathsf{Value} \times \{\mathtt{High}\}$.

Theorem 5. The abstraction function $\alpha : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$ (or $\alpha : \wp(\mathsf{State}^E) \to \mathsf{State}^{E\#}$) and the corresponding concretization function $\gamma : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$ (or $\gamma : \mathsf{State}^{E\#} \to \wp(\mathsf{State}^E)$) satisfy that, $\langle \alpha, \gamma \rangle$ is a Galois insertion: for all $S \in \wp(\mathsf{State}^E)$ and $S^\# \in \mathsf{State}^{E\#}$, it holds that $\alpha(S) \sqsubseteq S^\#$ if and only if $S \subseteq \gamma(S^\#)$.

Proof. In order to prove that $\langle \alpha, \gamma \rangle$ is a Galois insertion, it is enough to prove that α and γ satisfy: i) *monotonicity*, ii) the *deflationary* property, and iii) the

non-information loss property [Cousot and Cousot, 1979; Cousot and Cousot, 2002; Cousot, 2004], at the variable level.

1. *monotonicity* property:

   (a) monotonicity of $\gamma : \wp(\mathsf{Value}\times SecLabels) \rightarrow \wp(\mathsf{Value}\times SecLabels)$: Given $SValue_1^E, SValue_2^E \in \wp(\mathsf{Value} \times SecLabels)$, we have $SValue_1^E = \mathsf{Value}\times\{\mathsf{SecLab}_1\}$, and $SValue_2^E = \mathsf{Value}\times\{\mathsf{SecLab}_2\}$; by definition, $\gamma(SValue_1^E) = SValue_1^E$ and $\gamma(SValue_2^E) = SValue_2^E$. given $SValue_1^E \sqsubseteq SValue_2^E$, then we have $\gamma(SValue_1^E) \subseteq \gamma(SValue_2^E)$.
   Consequently γ satisfies monotonicity.

   (b) monotonicity of $\alpha : \wp(\mathsf{Value}\times SecLabels) \rightarrow \wp(\mathsf{Value}\times SecLabels)$: $SValue_1^E, SValue_2^E \in \wp(\mathsf{Value} \times SecLabels)$, with $SValue_1^E = SValue_1 \times \{\mathsf{SecLab}_1\}$, $SValue_2^E = SValue_2 \times \{\mathsf{SecLab}_2\}$, and $SValue_1, SValue_1 \in \wp(\mathsf{Value})$.

      i. given $SValue_1^E \subseteq SValue_2^E$, we have $SValue_1 \times \{\mathsf{SecLab}_1\} \subseteq SValue_2 \times \{\mathsf{SecLab}_2\}$; then $\mathsf{SecLab}_1 = \mathsf{SecLab}_2$;

      ii. By definition, $\alpha(SValue_1^E) = \mathsf{Value} \times \{\mathsf{SecLab}_1\}$, and $\alpha(SValue_2^E) = \mathsf{Value}\times\{\mathsf{SecLab}_2\}$, so given 1(b)i, $\alpha(SValue_1^E) = \alpha(SValue_2^E)$, therefore we have $\alpha(SValue_1^E) \sqsubseteq \alpha(SValue_2^E)$.

2. *deflationary* property:

   (a) $SValue^E \in \wp(\mathsf{Value} \times SecLabels)$, with $SValue^E = SValue \times \{\mathsf{SecLab}\}$, $SValue \subseteq \mathsf{Value}$ and $\mathsf{SecLab} \in SecLabels$.

   (b) By definition, $\alpha(SValue^E) = \mathsf{Value} \times \{\mathsf{SecLab}\}$.

   (c) By definition, $\gamma(\mathsf{Value} \times \{\mathsf{SecLab}\}) = \mathsf{Value} \times \{\mathsf{SecLab}\}$, then $\gamma(\alpha(SValue^E)) = \mathsf{Value} \times \{\mathsf{SecLab}\}$.

   (d) Given 2a we have $SValue \times \{\mathsf{SecLab}\} \subseteq \mathsf{Value} \times \{\mathsf{SecLab}\}$.
   Consequently, given 2b we have $SValue^E \subseteq \gamma(\alpha(SValue^E))$.

3. *non-information loss* property:

   (a) Given $SValue^E \in \wp(\mathsf{Value} \times SecLabels)$, we have $SValue^E = \mathsf{Value}\times\{\mathsf{SecLab}\}$; by definition, $\gamma(SValue^E) = SValue^E = \mathsf{Value}\times\{\mathsf{SecLab}\}$.

(b) By definition $\alpha(\mathsf{Value} \times \{\mathtt{SecLab}\}) = \mathsf{Value} \times \{\mathtt{SecLab}\}$, so we have $\alpha(S\,Value^E) = S\,Value^E$.

(c) Given 3a and 3b we have $S\,Value^E = \alpha(\gamma_{Var}(S\,Value^E))$, consequently $S\,Value^E \sqsubseteq \alpha(\gamma(S\,Value^E))$.

□

This result can be homomorphically extended to $\alpha : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$, so that the abstraction function α and the concretization function γ constitute a Galois insertion.

As it is well known, an alternative proof technique for demonstrating the correctness of the approximation would consist in proving that the abstraction function α is an upper closure operator with monotonicity, idempotency, and extensitivity.

Theorem 6 (Correctness)**.** If $\langle P_{\text{Java}}, S\,t_1^E\rangle \to^*_{\text{Java}^{\text{E}}} \langle S\,t_2^E\rangle$, then there exists $S\,S\,t_3 \in \wp(\mathsf{State}^E)$ s.t. $\langle P_{\text{Java}}, \alpha(\{S\,t_1^E\})\rangle \to^*_{\text{Java}^{\#}} \langle S\,S\,t_3\rangle$ and $S\,t_2^E \in S\,S\,t_3$.

Proof. The proof is by induction on the length n of the extended program trace or rewriting sequence denoted by $\to^*_{\text{Java}^{\text{E}}}$ (n is also the length of the corresponding abstract program trace $\to^*_{\text{Java}^{\#}}$).

1. ($n = 1$). There is only one rewriting step $\langle P_{\text{Java}}, S\,t_1^E\rangle \to_{\text{Java}^{\text{E}}} \langle S\,t_2^E\rangle$. Given $S\,t_1^E \in \alpha(\{S\,t_1^E\})$ and $S\,t_2^E \in \alpha(\{S\,t_2^E\})$, if $S\,S\,t_1 = \alpha(\{S\,t_1^E\})$ and $S\,S\,t_2 = \alpha(\{S\,t_2^E\})$, then by Definition 6 it holds that $\langle P_{\text{Java}}, \alpha(\{S\,t_1^E\})\rangle \to_{\text{Java}^{\#}} \langle\alpha(\{S\,t_2^E\})\rangle$, then it holds for $n = 1$.

2. ($n > 1$). The program trace of length n, $\langle P_{\text{Java}}, S\,t_1^E\rangle \to^*_{\text{Java}^{\text{E}}} \langle S\,t_2^E\rangle$, can be split as formed by two sub–traces of length $n-1$ and 1 respectively:

$$\underbrace{\langle P_{\text{Java}}, S\,t_1^E\rangle \to^*_{\text{Java}^{\text{E}}} \langle P_{\text{Java}^{\text{int}}}, S\,t_1^{intE}\rangle}_{\text{length } n-1} \to_{\text{Java}^{\text{E}}} \langle S\,t_2^E\rangle$$

By the induction hypothesis, the extended sub–trace of length $n-1$ has the following corresponding abstract sub–trace (of equal length):

$\langle P_{\text{Java}}, \alpha(S\,t_1^E)\rangle \to^*_{\text{Java}^{\#}} \langle P_{\text{Java}^{\text{int}}}, \alpha(S\,t_1^{intE})\rangle$.

Since $S\,t_1^E \in \alpha(\{S\,t_1^E\})$ and $S\,t_1^{intE} \in \alpha(\{S\,t_1^{intE}\})$, by Definition 6, the rewriting step $\langle P_{\text{Java}^{\text{int}}}, S\,t_1^{intE}\rangle \to_{\text{Java}^{\text{E}}} \langle S\,t_2^E\rangle$, has a corresponding abstract rewriting step $\langle P_{\text{Java}^{\text{int}}}, \alpha(S\,t_1^{intE})\rangle \to_{\text{Java}^{\#}} \alpha(\langle S\,t_2^E\rangle)$, then it also holds for $n > 1$.

Thus the conclusion follows. □

A program is non–interferent for a given labeling function if the abstract values (the confidentiality labels) of the *Low* variables in the final state of an abstract program execution do not have the label `Low` ≫ `High`.

Theorem 7 (Abstract Non–Interference Soundness)**.** Given a Java program P_{Java}, P_{Java} is non–interferent (Definition 4) if for all $SSt_1 \in \wp(\mathsf{State}^E)$ s.t. $\langle P_{\text{Java}}, SSt_1\rangle \mapsto^*_{\text{Java}^\#} \langle SSt_2\rangle$, for all $St \in SSt_2$, and for all variables $var \in Low(P_{\text{Java}})$, $St[var] = \langle Val, \texttt{Low}\rangle$ for a value Val.

Proof. By contradiction. Let us assume that P_{Java} is not non–interferent, i.e., there exists St_1^E with $\langle P_{\text{Java}}, St_1^E\rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St_2^E\rangle$ and $var \in Low(P_{\text{Java}})$ s.t. $St_2^E[var] = \langle Val, L\rangle$ for a value Val and $L \neq \texttt{Low}$. Since $\langle P_{\text{Java}}, St_1^E\rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St_2^E\rangle$, by Lemma 6, there exists $SSt_3 \in \wp(\mathsf{State}^E)$ s.t. $\langle P_{\text{Java}}, \alpha(\{St_1^E\})\rangle \rightarrow^*_{\text{Java}^\#} \langle SSt_3\rangle$ and $St_2^E \in SSt_3$. This contradicts the assumption that for all $St \in SSt_3$, and for all variables $var \in Low(P_{\text{Java}})$, $St[var] = \langle Val', \texttt{Low}\rangle$ for a value Val'. □

The following example illustrates the mechanization of the Java non–interference analysis.

Example 18. Consider again the Java program of Example 11. By virtue of the abstraction, we consider just one abstract initial state that safely approximates any extended initial state and compute the corresponding abstract final states.

```
search in PGM-SEMANTICS-ABSTRACT :
java((preprocess(EX1-MAUDE) noType . 'main < new string [i(0)] > noVal)) =>! M:Store .
Solution 1 M:Store --> store([l(6),Low >> High] ...)
No more solutions.
```

Due to the transformation of some equations into rules in the abstract semantics, there may be several execution paths but all lead to the same abstract final state.

Some proposals also exist for non–interference verification that are based on abstract interpretation [Barbuti et al., 2002; Zanotti, 2002; Giacobazzi and Mastroeni, 2004; Francesco and Martini, 2007; Zanardini, 2007]. However, these proposals do not generate a certificate as an outcome of the verification process, and they do not use JML to express non–interference policies.

The idea of first enriching the original semantics of the language by pairing each data value to its security level, and then approximating it by only considering the security level was also proposed in [Barbuti et al., 2002; Zanotti, 2002]. A similar idea is pursued in [Francesco and Martini, 2007], where an abstract information–flow sensitive collecting semantics, which is called instruction–level security typing, for programs with dynamic structures is proposed; here input and output channels are given security levels, but the variables have no associated security levels. On the one side, these approaches [Barbuti et al., 2002; Francesco and Martini, 2007] can verify non–interference of secure programs with temporary breaches such as Example 14, as well as program "`low = high; low = 2`". On the other side, these proposals fail to verify Example 17.

A different notion of abstract non–interference is proposed in [Giacobazzi and Mastroeni, 2004] that approximates the standard notion of non–interference by making it parametric relative to input/output abstractions. In abstract non–interference, the abstract domains encode the allowed flows that characterize the degree of precision of the knowledge of a potential attacker observing the data. By using classes and class hierarchies as abstract domains, Zanardini adopts a different perspective of abstract non–interference for classes in [Zanardini, 2007], where the abstract value of a concrete object is its class. Two objects (values) are indistinguishable at an abstraction level (class) if the objects belong to the given class or if the given class is a superclass of object classes. An algorithm for checking abstract non–interference of Java classes is proposed that relies on class–based dependencies.

Our global policies are very flexible since the security levels of object variables, local variables, and method parameters may change temporarily as in [Hunt and Sands, 2006; Jacobs et al., 2005; Barbuti et al., 2002; Francesco and Martini, 2007; Barthe et al., 2004]. This is illustrated in the Example 14 as well as in program "`low = high; low = 2`".

Equational abstractions [Meseguer et al., 2003] can be used to specify the abstract extended Java semantics, i.e. all extended integer values $\langle$`int(i), SecLab`$\rangle$ can be reduced to $\langle$`int(1), SecLab`$\rangle$. In this case, the multiplicative operations need no change. The extended integer arithmetic additive operations have to be modified so that the value `int(1)` become the unique possible result. The abstract extended relational integer operations need to be modified by using rules in order to maintain correctness and confluent equations while becoming non-deterministic. In this case, its is enough to intro-

duce equations without conditions in order to reduce stored values of program variables, because the abstraction of a given variable value does not depend on which program variable is or which is the corresponding actual concrete value.

However, all computations over integer numbers are useless and can be avoided as we do.

6.5 Experimental evaluation

The abstract certification methodology described here has been implemented in Maude. The tool is provided with a novel Web interface written in Java and is publicly available at `http://zenon.dsic.upv.es:8080/certificateX/controler`. The prototype system offers a rewriting-based program certification service, which is able to analyze global confidentiality properties related to program non–interference. Our certification tool can generate three types of certificates: (i) the full certificates consist of complete rewriting sequences including all rewrite steps; (ii) the reduced rules certificates only contain the rewrite steps that use rules; and (iii) the reduced labels certificates only record the labels of the used rules. A detailed description of the tool is given in Chapter 8.

Code Examples → Experiment Measures ↓	1	2	3	4	5
Code size in LOC	27	31	48	80	117
Code size in bytes	869	924	1981	3305	3504
Code cyclomatic complexity	1	1	4	16	192
Full Cert. size (Kb)	1134	1251	4223	10619	24176
Red. Rules Cert. size (Kb)	6.1	6.3	21.1	47.1	21.3
Red. Labels Cert. size (Kb)	1.8	1.8	2.6	3.7	5.2
Full Cert. Gen. Time (ms)	10408	23574	29482	45709	84331
Red. Rules Cert. Gen. Time (ms)	7057	7030	7527	8215	9547
Red. Labels Cert. Gen. Time (ms)	7030	6700	7190	8198	9537

Table 6.1: Code measures, certificate sizes, and generation times.

In Table 6.1, we analyze three key points for the practicality of our approach: the size and complexity of the program code, the size of the three types of certificates, and the certificate generation times. The running times

are given in milliseconds and were averaged over a sufficient number of iterations. We considered three code measures, the code size in LOC (lines of source code), the code size in bytes, and the cyclomatic complexity, which counts the execution paths of a program. The experiments were performed on a laptop with a Pentium M 1.40 GHz processor and 0.5 Gb RAM.

Program `1` mainly consists of a simple non–interferent code example borrowed from [Warnier, 2005; Jacobs et al., 2005] (Example 25 in Appendix C). The program has been structured into two classes. The first class has one secret variable and one public variable, a constructor method, two get methods, and a method that contains the non–interferent piece of code of [Warnier, 2005; Jacobs et al., 2005]. The second class is the main class with four method invocations. Similarly, program `2` is a simple non–interferent example borrowed from [Hunt and Sands, 2006]. It is structured into two classes (Example 26 in the Appendix). Program `3` includes three simple methods in two classes: the non–interferent method included in program `1`, an interferent method borrowed from [Warnier, 2005; Jacobs et al., 2005], and another non–interferent method borrowed from [Sabelfeld and Sands, 2009]. The main method has a sequence of method invocations such that the last invocation calls a non–interferent method, and thus the entire program is non–interferent (Example 27 in the Appendix). Program `4` includes six simple methods, the three methods included in program `3` and three other interferent methods also borrowed from [Warnier, 2005; Jacobs et al., 2005], including a method with a `while` loop and a method that calls another method. In this case, the last invoked method as well as the whole example program are non-interferent (Example 28 in the Appendix). Similarly, program `5` includes nine simple methods, the six examples included in program `4` plus three other interferent methods: two interferent variations of the loop example of Example 13 and an interferent method with a return statement within a conditional statement (Example 29 in the Appendix). The source code of all our benchmarks is also provided within the distribution package.

The experiments are very encouraging since they show that the reduction in size of the certificate is very significant in all cases, with the quotient "Red. Rules Cert. Size/Full Cert. Size" ranging from 0.54% in program `2` to 0.09% in program `5`. Note that the biggest reduction occurs for the largest program. When the time employed to generate the full and reduced rules certificates are compared, the reduced certificate generation time vs. the full certificate generation time range from 11.32% to 67.80%. The reduction for the biggest

example (program 5) was the largest one (11.32%). Note that the generation time for the reduced labels certificate were not significantly lower than the reduced rules certificate. These results confirm that the technique scales up much better when reduced certificates are considered.

CHAPTER 7

Analyzing Erasure with or without Non–Interference of Java Programs

7.1 Introduction

The non–interference policies considered in Chapter 6 are not restrictive enough for computer systems that are required to remove (or *erase*) the secret data after its intended use. *Erasure* is a way of strengthening confidentiality by upgrading data confidentiality levels, up to the extreme of demanding the removal of secret data from the system after it was used [Chong and Myers, 2005; Chong and Myers, 2008].

Let us illustrate the erasure policies by means of the following medical information example adapted from [Chong and Myers, 2008].

Example 19. Consider an on–line diagnosis web system implemented in Java which, once the patient has entered information about her symptoms, returns information about possible corresponding diseases back to the user. The website confidentiality policy states that the client symptoms and diagnostics are private, and that no record of them will be stored after the user has finished the application session.

```
class MedicalDiagnosis {
   boolean malaise, fever, influenza, userReqExit;
   //@ setLabel(fever, High);
   //@ setLabel(malaise, High);
   public void getSymptoms(){/* */}
   public void getUserReq(){/* */}
   public void exit(){/* */}
   public void diagnosis(){
       if (malaise && fever) then influenza = true; }
   public void appEnd(){
       malaise=false; fever=false; influenza=false;}
   public void medicalDiagnosis(){
```

```
        while (!userReqExit){
            getSymptons();
            diagnosis();
            getUserReq();};
        appEnd();
        exit();}
}
```

When the user requests to exit the application, the condition `userReqExit` becomes true, the method `appEnd` is executed, and the three sensitive variables `malaise`, `fever`, and `influenza` are erased. That is, after the execution of method `diagnosis`, the variables `malaise`, `fever`, and `influenza` have a high confidentiality level but no record of their values is kept after execution.

There are many situations where organizations or individuals have to make available to other organizations or systems some sensitive confidential data for specific purposes. These organizations and individuals assume that the receptors of their confidential information will erase this information once the purpose was achieved. For instance, when a customer pays a purchase using a credit card, the customer assumes that the merchant -and the payment system- keeps secure the customer and credit card details until the transaction ends, when the merchant -and the payment system- must erase them. The payment system is allowed to transmit credit card details to the bank system in order to obtain the bank authorization, but not to other users. All authentication systems must receive sensitive user data, i.e. login and password, biometric data such as finger-prints or images of the iris, but they have to erase them once authentication was done [Hunt and Sands, 2008].

7.2 Erasure policies

An erasure policy is a confidentiality policy that specifies that the confidentiality level of a given variable is upgraded to the extent that the system should not keep its value [Chong and Myers, 2005; Chong and Myers, 2008; Hunt and Sands, 2008]. Erasure does not imply non–interference (a program that satisfies erasure may not satisfy non–interference), nor vice versa. The erasure of a variable *var* is expressed as $var : L_1 \nearrow L_2$ where $L_1 \in \{\texttt{Low}, \texttt{High}\}$, $L_2 \in \{\texttt{High}, \top\}$, and $L_1 < L_2$. This means that the (value of the) variable *var* should be explicitly erased within the program code in every execution, as

well as any other variable that *depends on* it. The partial order $\leq$ is extended so that $\texttt{Low} < \texttt{High} < \top$. The commutative $\sqcup$ operator is also extended such that $\texttt{Low} \sqcup \texttt{Low} = \texttt{Low}$, $\texttt{Low} \sqcup \texttt{High} = \texttt{High}$, and $X \sqcup \top = \top$.

If $var : L_1 \nearrow \top$, the erasure policy means complete erasure [Hunt and Sands, 2008], i.e. initial states that differ only in the value of variable *var* produce the same final state. This means that, even if an observer can see `High`–labeled variables, it should not distinguish the erased values. Erasure policies with complete erasure mean that attackers can see `Low`–labeled variables (as the `Low` clearance of the non–interference attacker) and also `High`–labeled variables. If $var : L_1 \nearrow \texttt{High}$, this means partial erasure, i.e. initial states that differ only in the value of variable *var* produce the same final state, except for `High` variables. In summary, given an erasure policy $var : L_1 \nearrow L_2$, the erasure attacker has a clearance level as high as the highest level L such that $L < L_2$.

The erasure policy will be enforced from the program point where the JML-like annotation is located until the end of the entire program execution. This means that we should place the `erase` annotation at the program point just *after* the point where the variable is updated with the confidential data that must be erased. This can be seen as an instrumentation in our setting of the (conditioned) erasure of [Chong and Myers, 2005; Chong and Myers, 2008].

In our notation for erasure policy specification, an erasure policy $var : L_1 \nearrow L_2$ is written as a JML-like annotation with the word `erase`, e.g. $\texttt{erase}(\texttt{influenza}, \texttt{High}, \texttt{Top})$ represents $\texttt{influenza} : \texttt{High} \nearrow \top$.

Example 20. Consider the following example program adapted from [Hunt and Sands, 2008] and the initial state given by the execution of the function `main`:

```
class Eraserclass {
 int xh, yh, zl; //@ setLabel(xh, High); setLabel(yh, High);
 public void Testclass() {  }
 public void setxh(int xp) {
      /*@ setLabel(xp, High); @*/
      xh = xp; }
 public void setyh(int yp) {
      /*@ setLabel(yp, High); @*/
      yh = yp; }
 public void setzl(int zp) {
      zl = zp; /*@ erase(zl, Low, High); *@/}
 public void mE3(){
      xh = xh + yh + zl;
      yh = yh + 2;
```

```
      zl = 0;}  }
class Erasure3 {
      static  Eraserclass t = new Eraserclass();
      static int initzl;
      public static void main(String[] args) {
           initzl = Integer.parseInt(args[0]);
           t.setzl(initzl);
           t.mE3();}}
```

In the `Eraserclass`, the fields `xh` and `yh` are labeled `High`, and the field `zl` is labeled `Low` by default. The program `Erasure3` obeys the erasure policy `zl` : `Low`↗`High` because it erases the value of the `zl` variable that is set by the method `setzl` before program ends execution. In fact, it erases this value in the last assignment statement of the method `mE3`. The other variable `yh` that is labeled `High` does not depend on `zl`, whereas the variable `xh` depends on `zl` but it remains labeled as `High`. It must be noted that if we modify the erasure policy of `zl` from `zl` : `Low`↗`High` to `zl` : `Low`↗$\top$, the program is no longer secure, since variable `xh` should also be erased. Finally, if we replace the statement "`zl = 0;`" by "`zl = yh;`", the program satisfies the erasure policy but not the non–interference policy.

We define erasure of variables following the end-to-end erasure of non-interactive programs [Hunt and Sands, 2008], but we generalize it to support the erasure of n variables. A variable var_i is erased to some confidentiality level L_i if varying the initial value of var_i does not change the final state to all observers except for those at level L_i or above. Given a subset of program variables $V \subseteq Vars(P_{\text{Java}})$, we define an equivalence relationship $=_{\overline{V}}$ between states as $St_1 =_{\overline{V}} St_2$ if for all $var \in Vars(P_{\text{Java}}) - V$, it holds that $St_1[var] = St_2[var]$. We extend the notation $St_1 =_{Low} St_2$ of Definition 3 as follows: $St_1 =_L St_2$ with $L \in Labels = \{\texttt{Low}, \texttt{High}, \top\}$, if for all $var \in Vars(P_{\text{Java}})$ such that $Labeling(var) \leq L$ it holds that $St_1[var] = St_2[var]$. Note that $St_1 =_\top St_2$ implies $St_1 =_{High} St_2$, which implies $St_1 =_{Low} St_2$.

Definition 7 (Erasure)**.** Given a program P_{Java} and the program variables $V = \{var_1, \ldots, var_n\}$, P_{Java} complies with the erasure policy $var_i : L_i \nearrow L'_i$, with $i = 1 \ldots n$, iff for every pair of different program initial states St_1 and St_2, and for their corresponding final program states St'_1, St'_2 such that $\langle P_{\text{Java}}, St_1\rangle \mapsto^*_{\text{Java}} \langle St'_1\rangle$ and $\langle P_{\text{Java}}, St_2\rangle \mapsto^*_{\text{Java}} \langle St'_2\rangle$, we have that $St_1 =_{\overline{V}} St_2$ implies $St'_1 =_L St'_2$ for all $L < \lceil L \rceil$, where $\lceil L \rceil = L'_1 \sqcup \cdots \sqcup L'_n$.

This means that if each variable var_i is erased to confidentiality level L'_i, then varying the values of var_i does not change the final state to all observers

except for those at level $\lceil L \rceil$ or above, where $\lceil L \rceil$ is the lowest upper bound (*join*) of the L'_i confidentiality levels. Note that the attacker clearance depends on that *lub* level $\lceil L \rceil$. The following example considers the erasure of two variables to different confidentiality levels.

Example 21. Consider the following example program with two `High`–labeled variables `xh` and `yh`, and two `Low`–labeled variables `ul` and `zl`, together with a `main` function and a constructor method similar to Example 20, but with a different erasure policy.

```
class NoEraserclass {
      int xh; int yh; int zl; int ul;
      //@ setLabel(xh, High); setLabel(yh, High);
      public void setxh(int xp) {
           /*@ setLabel(xp, High); @*/
           xh = xp; }
      public void setyh(int yp) {
           /*@ setLabel(yp, High); @*/
           yh = yp; }
      public void setul(int up) {
           ul = up; /*@ erase(ul, Low, Top); *@/ }
      public void setzl(int zp) {
           zl = zp; /*@ erase(zl, Low, High); *@/ }
      public void mE4() {
           xh = xh + zl;
           yh = yh + ul ;
           zl = 0;
           ul = 0;} }
     class Erasure4 {
          static  NoEraserclass t = new NoEraserclass();
          static int initul, initzl;
          public static void main(String[] args) {
               initul = Integer.parseInt(args[0]);
               t.setul(initul);
               initzl = Integer.parseInt(args[1]);
               t.setzl(initzl);
               t.mE4(); } }
```

This program does comply with the single erasure policy $\texttt{zl} : \texttt{Low} \nearrow \texttt{High}$ because it satisfies the condition $St'_1 =_{\texttt{Low}} St'_2$. However, it does not comply with the single erasure policy $\texttt{ul} : \texttt{Low} \nearrow \top$ even if it satisfies the condition $St'_1 =_{\texttt{Low}} St'_2$, because the condition $St'_1 =_{\texttt{High}} St'_2$ (i.e. the variable `ul` is erased, but the variable `yh` is not) is not satisfied. When we consider the erasure of two variables $\texttt{zl} : \texttt{Low} \nearrow \texttt{High}$ and $\texttt{ul} : \texttt{Low} \nearrow \top$ altogether, it is enough to require the conditions corresponding to the later erasure, e.g. $St'_1 =_{\texttt{High}} St'_2$ and $St'_1 =_{\texttt{Low}} St'_2$, because this implies the condition required by the former erasure (i.e. $St'_1 =_{\texttt{Low}} St'_2$) . Thus, the erasure policy of the two variables is fullfilled, whenever $St'_1 =_{\texttt{L}} St'_2$, for all $L < \texttt{High} \sqcup \top$.

7.2.1 Erasure and non–interference

In practice, it is useful to enforce erasure together with non–interference [Hunt and Sands, 2008; Chong and Myers, 2008]. We state that a program P_{Java} complies with a combined erasure with non–interference policy if it simply satisfies the conditions of both, Definition 4 and Definition 7, independently. If we wish to enforce an erasure policy together with a non–interference policy, regarding both the variables directly affected by the erasure policy and the variables that depend on variables directly affected by an erasure policy, the erasure of a given variable *var* should be done by using an expression whose value has a confidentiality label as high as *Labeling*(*var*).

7.3 The extended Rewriting Logic semantics of Java for erasure

In order to consider both erasure and non–interference, the semantic instrumentation for non–interference of Chapter 6 is extended all together in this Section as follows:

1. Attach two extra labels to each memory location: a confidentiality label from the set $\{\texttt{Low}, \texttt{High}, \texttt{Low} \gg \texttt{High}, \texttt{Low} \gg \top, \texttt{High} \gg \texttt{Low}, \texttt{High} \gg \top\}$ and an erasure label from the set $\{\emptyset, \texttt{Low} \nearrow \texttt{High}, \texttt{Low} \nearrow \top, \texttt{High} \nearrow \top\}$. When program ends execution, a label $\texttt{L} \gg \texttt{L}'$ with $\texttt{L} \neq \texttt{L}'$ indicates a change of the confidentiality level label of the associated memory location, from the initial level L to the final level L′. This way we can detect hazardous confidentiality level changes of these locations, e.g. from level Low to level High. In other words, we can not only observe the confidentiality levels of the memory locations at the final execution state but also detect whether a location is directly affected by an erasure policy (or if it depends on variables that must obey an erasure policy).

2. Attach a confidentiality label and an erasure label to the evaluation of program expressions; this allows us to track whether the evaluation of the expression involves high confidentiality data and data that are affected by an erasure policy.

3. Associate a confidentiality label and an erasure label to the evaluation of program statements, especially conditional statements, to control im-

plicit flows from high confidential variables and from variables that should be erased. However, these confidentiality and erasure labels are not attached to each program statement. Rather they are kept as extra attributes of states in the extended Java semantics.

The confidentiality level labels for non–interference are extended here for erasure policies with the new confidentiality level label associated with complete erasure, i.e. $\top$, and the corresponding new confidentiality level changes, i.e. $\texttt{Low} \gg \top$, and $\texttt{High} \gg \top$. The new erasure label is introduced in order to propagate the erasure policies as information flows between the variables of the program, either explicitly by assignment statements or implicitly by control flow statements.

Here we extend further, in a modular way, the extended Java semantics for non–interference of Section 6.3 to deal with the JML-like annotations of the form $\texttt{erase}(\texttt{Var}, \texttt{L}_1, \texttt{L}_2)$, which correspond to the erasure policies. We describe the information flow extended version of the rewriting logic semantics of Java by the rewrite theory $\mathcal{R}_{\text{Java}^{\text{E}}} = (\Sigma_{\text{Java}^{\text{E}}}, E_{\text{Java}^{\text{E}}}, R_{\text{Java}^{\text{E}}})$, $E_{\text{Java}^{\text{E}}} = \Delta_{\text{Java}^{\text{E}}} \uplus B_{\text{Java}^{\text{E}}}$ and its corresponding $\rightarrow_{\text{Java}^{\text{E}}}$ rewriting relation [Alba-Castro et al., 2010b].

In the new semantics, program data do not only consist of standard concrete values but each value is decorated with its corresponding confidentiality and erasure labels. We introduce the new confidentiality label $\top$. Thus, we now have $\mathit{Labels} = \{\texttt{Low}, \texttt{High}, \top\}$ and $\mathit{ConfLabChange} = \{\texttt{Low} \gg \texttt{High}, \texttt{Low} \gg \top, \texttt{High} \gg \texttt{Low}, \texttt{High} \gg \top\}$. We extend the join operator consistently in order to consider the new $\top$ label, as well as the new composed labels $\{\texttt{Low} \gg \top, \texttt{High} \gg \top\}$. Also, $\texttt{L join} \top = \top$ for any $\texttt{L} \in \mathit{Labels}$. Regarding variable updating, we also extend the $\ggg$ operator (that computes the new confidentiality label in terms of the previous label at the memory location) as shown in Figure 7.1.

In order to record erasure, we introduce the sort $\mathit{EraLabels} = \{\varnothing, \texttt{Low} \nearrow \texttt{High}, \texttt{Low} \nearrow \top, \texttt{High} \nearrow \top\}$. The domain of program variables in the extended semantics for erasure and non–interference is now $\mathit{Value} \times (\mathit{Labels} \cup \mathit{ConfLabChange}) \times \mathit{EraLabels}$. The empty label ($\varnothing$) of domain $\mathit{EraLabels}$ means no erasure policy. If a variable *var* has the labeled value $\langle \mathit{val}, L, L' \nearrow L'' \rangle$, L is the confidentiality level of the variable, with $L \in \mathit{Labels} \cup \mathit{ConfLabChange}$, and $L' \nearrow L''$ is the erasure label of the variable. It holds that either $L = L''$ or $L = L''' \gg L''$. The erasure label $L' \nearrow L''$ means that the variable *var* has to be erased, either because this erasure label corresponds

LVal	LVal'	LVal $\ggg$ LVal'
L_1	L_1	L_1
L_1	L_2	$L_1 \gg L_2$
$L_1 \gg L_2$	L_1	L_1
$L_1 \gg L_2$	L_2	$L_1 \gg L_2$
$L_1 \gg L_2$	L_3	$L_1 \gg L_3$
L	$L_1 \gg L_2$	$L \ggg L_2$
$L_1 \gg L_2$	$L \gg L_3$	$(L_1 \gg L_2) \ggg L_3$

$L_1, L \in \{\texttt{Low}, \texttt{High}\}, L_2, L_3 \in \{\texttt{Low}, \texttt{High}, \top\}$
$L_1 \neq L_2, L \neq L_3, L_1 \neq L_3, L_2 \neq L_3$

Figure 7.1: Updating memory locations for Erasure.

to an erasure policy for *var* (i.e. $var : L' \nearrow L''$) or because, the variable *var* is (recursively) affected by an erasure policy for other variables on which *var* depends on. With these labels, we can check whether the variables are involved in erasure or non–interference policies. If they are involved in an erasure policy, we can also check if they must be erased or not when program ends execution, using their confidentiality labels.

We also extend the labels of expressions and statements in order to propagate the erasure label of a variable to the variables that depend on it, explicitly or implicitly. The context label has now two labels, the confidentiality label and the erasure label. We introduce a new commutative $\uplus$ (*join*) operator for erasure labels in order to propagate the strongest policies on erased values and its dependences, as shown in Figure 7.2.

Erasure Label	$\uplus$	Erasure Label	=	Resulting Erasure Label
$l_1 \nearrow l_2$	$\uplus$	$\emptyset$	=	$l_1 \nearrow l_2$
$l_1 \nearrow l_2$	$\uplus$	$l_3 \nearrow l_4$	=	$(l_1 \sqcup l_3) \nearrow (l_2 \sqcup l_4)$

$l_1, l_2, l_3, l_4 \in Labels \cup ConfLabChange, l_1 \nearrow l_2, l_3 \nearrow l_4 \in EraLabels$

Figure 7.2: Joining over erasure labels.

The $\uplus$ operator is used during the evaluation of expressions with operators to *join* the erasure labels of the operands, as shown in Figure 7.3 for an `op` binary operator. The $\sqcup$ operator introduced in Figure 6.5 for non–interference is extended here to *join* the confidentiality labels $Labels = \{\texttt{Low}, \texttt{High}, \top\}$ and the confidentiality level changes $ConfLabChange = \{\texttt{Low} \gg \texttt{High}, \texttt{Low} \gg \top, \texttt{High} \gg \texttt{Low}, \texttt{High} \gg \top\}$ during binary expression evaluation, as shown in Figure 7.3.

$$\frac{\text{Labeled Value op Labeled Value} = \text{Resulting Labeled Value}}{\langle v_1, l_1, p_1\rangle \quad \texttt{op} \quad \langle v_2, l_2, p_2\rangle \quad = \langle v_1 \ \texttt{op} \ v_2, l_1 \sqcup l_2, p_1 \uplus p_2\rangle}$$
$$v_1, v_2 \in Value,\ l_1, l_2 \in Labels \cup ConfLabChange,\ p_1, p_2 \in EraLabels$$

Figure 7.3: Binary expression evaluation.

The new $\uplus$ operator is used during variable evaluation to *join* the erasure label of the variable with the erasure label of the program context, as shown in Figure 7.5. The extended $\sqcup$ operator is also used during variable evaluation, to *join* the confidentiality label of the variable with the confidentiality label of the context as specified in Figure 6.4 for non–interference, but here it works over the extended domain $Labels \cup ConfLabChange$.

```
eq k(i(I) -> K) lenv(CL,CEraL) = k(< int(I),CL,  CEraL > -> K) lenv(CL,CEraL) .
eq k(b(B) -> K) lenv(CL,CEraL) = k(< bool(B), CL,  CEraL > -> K) lenv(CL,CEraL) .
```

Figure 7.4: Equations of extended constant evaluation with erasure labels.

```
---First obtain location in store from variable name
eq k(Var -> K) env([Var, Loc] Env) = ... .
---Then obtain value stored in this location
eq k(#(Loc) -> K)   store([Loc, < Val, ValLab, ValEraLab > ] Store) lenv(CL,CEraL)
 = k(< Val, ValLab join CL, ValEraLab join CEraL > -> K)
   store([Loc, < Val, ValLab, ValEraLab > ] Store) lenv(CL,CEraL) .
```

Figure 7.5: Equations of extended variable content retrieval with erasure labels.

The extended equations that build the environment for non–interference analysis (Figure 6.6) can be used here with no modification. However, here we need a new specification of the `setLabel` function to take into account the introduced erasure label, as shown in Figure 7.6. Note the use of the extended $\sqcup$ operator to join the confidentiality level of the program variable with the confidentiality level of the program context, and the use of the $\uplus$ operator to join the erasure label of the variable with the erasure label of the program context.

In Figure 7.7 we show the specification of the Java + binary operator expression that also uses the `join` operators: $\sqcup$ to `join` confidentiality labels and $\uplus$ to `join` erasure labels. The extended specification of the Java <= operator is shown in Figure 7.8. Note that the specifications of binary operators are very similar. The confidentiality label and erasure label of the resulting

```
 op setLabel : Var Value SecLabel ErasureLabel  -> Value .
--- equation generated from JML-like annotation setLabel(h, High)
eq setLabel('h, < Val, ConfLab, EraLab >, EnvConfLab, EnvEraLab ) = < Val, High, nopol > .
--- Default case:
eq setLabel(Var, < Val, ConfLab, EraLab >, EnvLab, Pole  )
   = < Val, ConfLab join EnvLab  , EraLab join EnvEraLab  > .
```

Figure 7.6: Continuation-based equations for setting the initial variable confidentiality level.

```
eq k((< int(I), SecLab1, EraLab1 >, < int(I'), SecLab2, EraLab2 >) -> (+ -> K))
 = k(< int(I + I'), SecLab1 join SecLab2, EraLab1 join EraLab2 > -> K)  .
```

Figure 7.7: Equations of extended Java + operator.

```
eq k((< int(I), SecLab1, EraLab1   >,< int(I'), SecLab2, EraLab2 >) -> (<= -> K))
 = k(< bool(I <= I'), SecLab1 join SecLab2, EraLab1 join EraLab2 > -> K)  .
```

Figure 7.8: Equations of extended Java <= operator.

```
eq k(< int(I), SecLab, EraLab > -> ++'(L) -> K)
  = k([< int(I + 1),SecLab, EraLab > -> L] -> (< int(I), SecLab, EraLab > -> K)) .
```

Figure 7.9: Equations of extended Java ++ post-increment operator.

```
---Obtain variable location and evaluate expression
eq k(Var = E -> K) env([Var, Loc] Env) = ... .
---Once the expression is computed, assign to location
eq k(<Val,LVal, ELab> -> =(L) -> K)
 = k([<Val,LVal, ELab> -> L] -> (<Val,LVal, ELab> -> K )) .
---General procedure to update the memory
eq k([<Val,LVal, ELab> -> Loc] -> K) store([Loc,<Val',LVal', ELab'>] ST)
 = k(K) store([Loc,<Val, LVal' >>> LVal, ELab >] ST) .
```

Figure 7.10: Equations of extended Java assignment operator.

value is the `join` of the operand's confidentiality labels and erasure labels, respectively. Figure 7.9 shows the specification of the Java ++ post-increment unary operator. Note that this unary operator updates the involved variable but it does not change its confidentiality label, nor its erasure label.

The specification of the assignment operator extended for erasure is shown in Figure 7.10. The specifications uses the $\ggg$ operator extended for erasure in Figure 7.1.

When a boolean expression guards a conditional statement, the $\uplus$ operator is also used to update the erasure label of the program context in order to consider implicit flows regarding erasure policies, as shown in Figure 7.11.

```
--- Evaluates boolean expression keeping the then and else stmnts
ceq k((if E S else S') -> K) lenv(CL, EL) = k(E -> (if(S, S') ->
       restoreLEnv(CL,EL) -> K)) lenv(CL,EL)
    if not break-or-continue(S) and not break-or-continue(S') .
ceq k((if E S else S') -> K) = k(E -> (if(S, S') -> K))
    if break-or-continue(S) or break-or-continue(S') .
eq k(<bool(true),LVal, EVal> -> (if(S, S') -> K)) lenv(CL,EL)
   = k(S -> K) lenv(CL join LVal,EL join EVal) .
eq k(<bool(false),LVal, EVal> -> (if(S, S') -> K)) lenv(CL,EL)
   = k(S' -> K) lenv(CL join LVal,EL join EVal) .
--- New equation to restore previous context labels
eq k(restoreLEnv(CL,EL) -> K) lenv(CL',EL') = k(K) lenv(CL,EL) .
```

Figure 7.11: Equations of extended if-then-else with erasure labels.

On the one side, the `setLabel` annotations that specify in the JML-style syntax the confidentiality level of some program variables are translated to the Maude `setLabel` operator (see Figure 7.6) invocations that are used to set the confidentiality label and the erasure label of the variables when their environments are built. The default confidentiality label is `Low` and the initial erasure label is always $\emptyset$. On the other side, the `erase` JML-like annotations corresponding to the erasure policies are translated into calls to the special Java operators `eraseT` and `eraseH`, depending on the specified complete or partial erasure policy, in order to enforce it from the program execution point where the annotation is located and executed until the program ends its execution.

The automatic translation of the `erase`(`Var`, $\mathtt{L}_1$, $\mathtt{L}_2$) annotations is specified as shown below, where `X` $\in$ {`Low`, `High`}.

JML-like erasure annotations:	Java generated code:
//@ `erase`(`Var`, `X`, `Top`);	`eraseT`(`Var`);
//@ `erase`(`Var`, `Low`, `High`);	`eraseH`(`Var`);

These `eraseT` and `eraseH` Java operators have no semantics in Java (i.e., they behave as the identity function), but our technique interprets them in the proper way as shown in Figure 7.12.

The Java `eraseH` operator upgrades the `Low` confidentiality label of the variable *var* up to `High`. The definitions and equations of the Java and Maude `eraseH` operator are similar to the `eraseT` case described above and are thus omitted.

```
--- First obtain and keep the variable location and obtain its value
eq k((eraseT < Var'>) -> K) obj(o(OA)) env([Var', Loc'] E')
= k(#(Loc') ->(eraseT(Var',Loc') ->K)) obj(o(OA)) env([Var',Loc'] E') .
--- Then upgrade label and store at var location
eq k(Val -> ( eraseT(Var,Loc) -> K)) = k([eraseT(Val) -> Loc] -> K) .
op eraseT : Value -> Value .
eq eraseT(< Val, L1, EL >) = < Val , Top, L1 -> Top > .
eq eraseT(< Val, L1 >> L2, EL>) = < Val , Top, L1 -> Top > .
```

Figure 7.12: Java and Maude `eraseT` operator equations.

7.3.1 Proving erasure as a safety property

Erasure is also considered a security property, i.e., a property defined on sets of sets of traces. For instance, sets of traces whose initial and final states are indistinguishable at a given confidentiality level (Definition 7). Erasure is therefore defined as a *hyperproperty* [Clarkson and Schneider, 2008]. In order to simplify the verification of the erasure policies we approximate information erasure to a safety property, i.e a property based on set of traces.

Let us introduce our notion of erasure that is stated as a safety property.

Definition 8 (Strong Erasure)**.** For a given labeling function, a Java program P_{Java} strongly complies with the erasure policy var_i: $L_i \nearrow L_i'$, for $i = 1 \ldots n$, if for every extended initial state St_1^E and for its corresponding final program state St_2^E given by $\langle P_{\text{Java}}, St_1^E \rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St_2^E \rangle$, we have that for all $var \in Vars(P_{\text{Java}})$, either $St_2^E[var] = \langle Val, Lab, \emptyset \rangle$ or $St_2^E[var] = \langle Val, \texttt{High}, \texttt{Low} \nearrow \texttt{High} \rangle$, for a value Val and a label Lab.

If a public variable has a non-empty erasure label, this means that the variable must be erased. However, if a secret variable has a non-empty erasure label, this does not necessarily means that the variable must be erased. A secret variable with the erasure label "`Low`$\nearrow$`High`" must not be erased, because it has the confidentiality label "`High`" that corresponds to its `High`-confidentiality.

For a program that does not comply with a strong erasure policy, this means that in the final state of, at least, one extended execution there is one variable that, either, i) it is a public variable that has a non-empty erasure label, or ii) it is a secret variable that has a non-empty erasure label that indicates complete erasure (i.e. $\texttt{L} \gg \top$ for $\texttt{L} \in \{\texttt{Low}, \texttt{High}\}$).

Note that the erasure policies are satisfied by programs with final states such that the program variables have all *empty* erasure labels, whenever or not they have confidentiality level changes that indicate transitions from the

Low confidentiality level to the High level ("Low $\gg$ High"). In other words erasure policies (Definition 8) do not mean non–interference policies (Definition 5).

Theorem 8. Consider a Java program P_{Java} together with the erasure policy var_i: $L_i \nearrow L'_i$, for $i = 1 \ldots n$, and two initial states St_1 and St_2 such that $St_1 =_{\overline{V}} St_2$, where $V = \{var_1, \ldots, var_n\}$. Consider the two corresponding final program states St'_1 and St'_2 given by $\langle P_{\text{Java}}, St_1\rangle \mapsto^*_{\text{Java}} \langle St'_1\rangle$ and $\langle P_{\text{Java}}, St_2\rangle \mapsto^*_{\text{Java}} \langle St'_2\rangle$. If there exists $var \in Vars(P_{\text{Java}})$, such that $Labeling(var) < \lceil L \rceil$, where $\lceil L \rceil = \sqcup L'_i$ for $i = 1 \ldots n$, and $St'_1[var] \neq St'_2[var]$, then the extended state St'^E_1 satisfies $\langle P_{\text{Java}}, St^E_1\rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St'^E_1\rangle$ and it holds that for a value Val, either, $St'^E_1[var] = \langle Val, \texttt{Low} \gg \texttt{High}, \texttt{Low} \nearrow \texttt{High}\rangle$ or $St'^E_1[var] = \langle Val, \texttt{L}_1 \gg \top, \texttt{L}_2 \nearrow \top\rangle$, with $L_1, L_2 \in \{\texttt{Low}, \texttt{High}\}$.

Proof. The proof is similar to the proof of Theorem 3 in Section 6.3.1, by considering eventual flows from variables that have to be erased (V), to any program variable ($Vars(P_{\text{Java}})$). Additionally, we have to consider the case when the variables directly affected by the erasure policy are not erased as they should be.

Consider the two traces $\mathcal{D}_1 : \langle P_{\text{Java}}, St_1\rangle \mapsto^*_{\text{Java}} \langle St'_1\rangle$ and $\mathcal{D}_2 : \langle P_{\text{Java}}, St_2\rangle \mapsto^*_{\text{Java}} \langle St'_2\rangle$. Let $\{var_1, \ldots, var_k\} \subseteq Vars(P_{\text{Java}})$ be those variables such that $Labeling(var_j) < \lceil L \rceil$, where $\lceil L \rceil = \sqcup L'_i$ for $i = 1 \ldots n$, and $St'_1[var_j] \neq St'_2[var_j]$ for all $1 \leq j \leq k$. These k variables can be or not variables directly affected by the erasure policy. We assume $k > 0$, then there is at least one of those variables (say var_1).

Let n be the total number of assignments in $\mathcal{D}_1$ to variables $\{var_1, \ldots, var_k\}$, after their corresponding erase annotations (if any). Note that n is finite since execution traces are finite because of the termination assumption. Since each variable var_j could have or not an erasure policy, this number of assignments could be equal to zero, i.e. $0 \leq n$.

The result is reformulated as Proposition 2, where $PE(n, k)$ is a predicate that relates n with k.

Proposition 2. $PE(n, k)$:

$\mathcal{D}_1$ has n assignments to the k variables $\{var_1, \ldots, var_k\}$ [1], and it holds that for all var_j in $\{var_1, \ldots, var_k\}$ and a value Val_j, either, $St'^E_1[var_j] =$

[1] These k variables $\{var_1, \ldots, var_k\}$ satisfy the *non-erasure* condition, i.e. $St'_1[var_j] \neq St'_2[var_j]$ for all $1 \leq j \leq k$.

$\langle Val_j, \texttt{Low} \gg \texttt{High}, \texttt{Low} \nearrow \texttt{High}\rangle$, or $St_1'^{E}[var_j] = \langle Val_j, \mathtt{L}_1 \gg \top, \mathtt{L}_2 \nearrow \top\rangle$, for $\mathtt{L}_1, \mathtt{L}_2 \in \{\texttt{Low}, \texttt{High}\}$.

Now, we prove the result by induction on n.

1. ($n = 0$) This case means that all k variables var_j are directly affected by the erasure policy (i.e. $\forall var_j : L_j \nearrow L'_j$), and it also means that they have not been erased after executing the erasure annotations. Thus, the final values of all variables var_j in the corresponding extended execution are the values assigned by the erase operator (See Figure 7.12), i.e. $St_1'^{E}[var_j] = \langle Val, L_j \gg L'_j, L_j \nearrow L'_j\rangle$ and the conclusion $PE(0, k)$ follows.

2. ($n = 1$) Let us consider the execution step in $\mathcal{D}_1$ where the assignment $var_1 = E_1$ is executed. Then, it may happen that the assignment $var_1 = E_1$ is also executed in $\mathcal{D}_2$, or not. We consider these two cases separately.

   (a) If $var_1 = E_1$ is also executed in $\mathcal{D}_2$, then $St'_1[var_1] \neq St'_2[var_1]$ implies that the values for E_1 are different in the two traces. Thus, expression E_1 must contain at least one variable var_e such that the actual values of var_e are different in the two traces when the considered assignments to var_1 are executed. Since $n = 1$ and $St'_1[var_e] \neq St'_2[var_e]$, then $var_e \notin Vars(P_{\text{Java}}) - V$, i.e. $var_e \in V$. Therefore var_e is variable with an erasure policy, e.g. $var_e : L_e \nearrow L'_e$, but, it is not erased. Regarding the confidentiality level label of variables var_1 and var_e we have four possible cases that we analyse as follows, using the erasure confidentiality label changes of Figure 7.1. First, we consider the case $Labeling(var_1) = \texttt{Low}$, and $Labeling(var_e) = \texttt{Low}$. This means that the erasure policy over var_e is $var_e : \texttt{Low} \nearrow L'_e$, with $L'_e \in \{\texttt{High}, \top\}$. In this case, the confidentality label $\texttt{Low} \gg L'_e$ and the erasure label $\texttt{Low} \nearrow L'_e$ are assigned to variable var_1. Next, we consider the case $Labeling(var_1) = \texttt{Low}$, and $Labeling(var_e) = \texttt{High}$. This means that the erasure policy over var_e is $var_e : \texttt{High} \nearrow \top$. In this case, the confidentality label $\texttt{Low} \gg \top$ and the erasure label $\texttt{Low} \nearrow \top$ are assigned to variable var_1. Then, we consider the case where $Labeling(var_1) = \texttt{High}$, and $Labeling(var_e) = \texttt{Low}$.

This means that the erasure policy over var_e is var_e : Low $\nearrow \top$. In this case, the confidentality label High $\gg \top$ and the erasure label Low $\nearrow \top$ are assigned to variable var_1. Finally, we consider the case where $Labeling(var_1)$ = High, and $Labeling(var_e)$ = High. This means that the erasure policy over var_e is var_e : High $\nearrow \top$. In this case, the confidentality label High $\gg \top$ and the erasure label High $\nearrow \top$ are assigned to variable var_1. In the four cases, the variable var_1 is such that, its final confidentiality label is a label change, and its final erasure label is a non-empty erasure label. Thus, the conclusion $PE(1, k)$ follows.

(b) If $var_1 = E_1$ is not executed in $\mathcal{D}_2$, then $St'_1[var_1] \neq St'_2[var_1]$ implies that the execution of this last assignment statement $var_1 = E_1$ in $\mathcal{D}_1$ is conditioned to the result of a boolean expression containing non erased variables with erasure policies over them, that guards a conditional (or `while` loop) statement so that the assignment is executed in $\mathcal{D}_1$ and not in $\mathcal{D}_2$. Then, the assignment statement $var_1 = E_1$ in $\mathcal{D}_1$ was executed either (i) within the *then* or *else* branch of an if-then-else Java statement (recall that while loops are expressed as `if-then-else` statements), (ii) within the *then* branch of an `if-then` Java statement, or (iii) after evaluating a conditional expression within a while loop that includes a `break` expression. Note that no other case can generate a non erasure condition. In all three cases, our extended semantics assigns a confidentiality label change and a non-empty erasure label to the boolean guard expression of such a conditional expression as shown in Figures 7.3 and 7.2, and the context confidentiality and erasure labels are set (according to Figure 7.11) before the expression E_1 is evaluated in the statement $var_1 = E_1$. Note that in case (iii), the conditional expression propagates the context confidentiality and erasure labels outside itself (according to Figure 7.11), i.e. the conditional does not restore the previous context confidentiality and erasure labels precisely to record that even if sequence $\mathcal{D}_1$ does not execute the `break` statement, another possible trace (e.g. $\mathcal{D}_2$) can do it. Finally, in all three cases, the expression E_1 is evaluated within a context labeled with a confidentiality label change and a non-empty erasure label, and then, a confidentiality label change $\mathtt{L}'' \gg \mathtt{L}'$ and a non-empty erasure label $\mathtt{L} \nearrow \mathtt{L}'$ are

assigned to variable var_1, independently of whether expression E_1 manipulates erased confidential data or not. Then, the conclusion $PE(1, k)$ follows.

3. ($n > 1$) Let us consider the last execution step in $\mathcal{D}_1$ where the assignment $var_i = E_i$ is executed, with $1 \leq i \leq k$. We split into two cases.

   (a) If $var_i = E_i$ is also the last assignment of variables $\{var_1, \ldots, var_k\}$ executed in $\mathcal{D}_2$, then $St'_1[var_i] \neq St'_2[var_i]$ implies that the values for E_i are different in the two traces. Thus, expression E_i must contain at least one variable var_e such that the actual values of var_e are different in the two traces when the considered assignments to var_i are executed. Then, let us consider whether $var_e \in \{var_1, \ldots, var_k\}$ or not. If it does, then by induction hypothesis, we can assume that variable var_e is labeled with a confidentiality label change ($\mathtt{L}'' \gg \mathtt{L}'$) and a non empty erasure label ($\mathtt{L} \nearrow \mathtt{L}'$). This means that, variable var_e is either a non-erased variable with an erasure policy over it, or it is a variable that, directly or indirectly, depends on variables with erasure policies over them. In both cases, a confidentiality label change and a non-empty erasure label are assigned to variable var_i and the conclusion follows. Since $var_i = E_i$ is the last assignment of variables $\{var_1, \ldots, var_k\}$ in both traces, and var_e has different values in both traces, then the case $var_e \notin \{var_1, \ldots, var_k\}$ cannot happen.

   (b) If $var_i = E_i$ is not the last assignment of variables $\{var_1, \ldots, var_k\}$ executed in $\mathcal{D}_2$, then either there is no such an assignment in $\mathcal{D}_2$ to variables $\{var_1, \ldots, var_k\}$, or the last assignment in $\mathcal{D}_2$ has the form $var_i = E'$, with E' different from E_i, or it affects a variable var'' that is different from var_i. All three cases imply that the execution of the last assignment statement $var_i = E_i$ in $\mathcal{D}_1$ is conditioned to the result of a boolean expression containing either, non erased variables with erasure policies over them, or variables that depend on variables with erasure policies over them. This boolean expression guards a conditional (or `while` loop) statement so that such assignment is executed in $\mathcal{D}_1$ and not in $\mathcal{D}_2$. Then this case is perfectly similar to case (2)(b) above, and the result follows.

□

We conclude that strong erasure implies erasure as given by the following result.

Theorem 9 (Strong Erasure Soundness)**.** Given a Java program P_{Java} with the erasure policy var_i: $L_i \nearrow L_i'$, if P_{Java} strongly complies with this erasure policy (Definition 8), then P_{Java} complies with this erasure policy (Definition 7).

Proof. (By contradiction) Assume that program P_{Java} complies with the strong erasure policy and also that P_{Java} does not comply with the erasure policy. Since P_{Java} complies with the strong erasure policy, for every extended initial state St^E and for its corresponding final program state $St^{E'}$ given by $\langle P_{\text{Java}}, St^E\rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St^{E'}\rangle$, we have that, for all $var \in Vars(P_{\text{Java}})$, either $St^{E'}[var] = \langle Val, Lab, \emptyset\rangle$, or $St^{E'}[var] = \langle Val, \texttt{High}, \texttt{Low} \nearrow \texttt{High}\rangle$, for a value Val and $Lab \in \{\texttt{Low}, \texttt{High}, \texttt{Low} \gg \texttt{High}, \texttt{High} \gg \texttt{Low}\}$. By Lemma 8 and the assumption that P_{Java} does not comply with the erasure policy we have that either, $St^{E'}[var] = \langle Val, \texttt{Low} \gg \texttt{High}, \texttt{Low} \nearrow \texttt{High}\rangle$, or $St^{E'}[var] = \langle Val, \texttt{Low} \gg \top, \texttt{Low} \nearrow \top\rangle$, or $St^{E'}[var] = \langle Val, \texttt{Low} \gg \top, \texttt{High} \nearrow \top\rangle$, or $St^{E'}[var] = \langle Val, \texttt{High} \gg \top, \texttt{High} \nearrow \top\rangle$ for a value Val, hence P_{Java} does not comply with the strong erasure policy, contradicting the hypothesis. □

In other words, the above result allows us to transform erasure into a stronger, safety property in the extended semantics. Obviously, we are not able to certify the security of all of the programs that are secure with erasure. For instance, the code of method `secure` in Example 22 cannot be verified by our technique neither by the type system of [Hunt and Sands, 2008].

Example 22. Consider the following Java program adapted from [Hunt and Sands, 2008] and semantically equivalent to Example 20, but with only one `High`–labeled variable.

```
class Testclass {
  int xh; int yl; int zl;
  //@ setLabel(xh, High);
  public void secure(){//@ erase(zl,Low,High);
     xh=xh+yl+zl;
     yl=yl+2+zl;
     yl=yl-zl;
     zl=0;}
}
```

Apparently, there is a direct flow from variable `zl` to variable `yl` in two assignment statements of the method `secure`. However for any execution, when method `secure` ends, the value of variable `yl` coincides with its own

value before the execution of method plus `2` so that the variable `yl` does not depend on the variable `zl`. According to Definition 7, the program satisfies the erasure policy. However, we cannot prove it by using our notion of strong erasure since the assignments and "`yl = yl + 2 + zl;`" and "`yl = yl − zl;`" assigns the label $\langle \texttt{Low} \gg \texttt{High}, \texttt{Low} \nearrow \texttt{High} \rangle$ to the variable `yl`.

Note that this is not a limitation of our method but a simple consequence of the undecidability of erasure, and affects any erasure analysis technique including those based on type inference [Hunt and Sands, 2008].

If we want to analyse non–interference with erasure, we have to use our stronger notion of non–interference as a safety property of Section 6.3.1.

In our methodology, a program P_{Java} complies with both, non–interference and erasure policies if it complies with the two policies separately. The following result is inmediate.

Corollary 1 (Strong Erasure with Non–interference Soundness)**.** A program P_{Java} is non–interferent (Definition 4) and secure with end-to-end erasure (Definition 7), if it satisfies conditions of Definitions 5 and 8.

7.4 The extended abstract Rewriting Logic semantics of Java for erasure

Finally, we develop an abstract version of the extended rewriting logic semantics of Java developed in Section 7.3, which we describe by the rewrite theory $\mathcal{R}_{\text{Java}^\#} = (\Sigma_{\text{Java}^\#}, E_{\text{Java}^\#}, R_{\text{Java}^\#})$, $E_{\text{Java}^\#} = \Delta_{\text{Java}^\#} \uplus B_{\text{Java}^\#}$ and its corresponding $\rightarrow_{\text{Java}^\#}$ rewriting relation. As in Section 6.4, our approach for the abstract Java semantics consists in modifying the original theory $\mathcal{R}_{\text{Java}^\text{E}}$ (taking advantage of its modularity) by abstracting the domain $(\mathit{Labels} \cup \mathit{ConfLabChange}) \times \mathit{EraLabels}$, and introducing approximate versions of the Java constructions and operators tailored to this domain. Intuitively, this means that we simply get rid of the values in the abstract semantics, and use only their confidentiality and erasure labels as the abstract values instead [Alba-Castro et al., 2010b]. The abstract values are pairs of labels, the confidentiality level label and the erasure label, while in the abstract semantics for non–interference the abstract values are single labels: the confidentiality level labels.

The abstract semantics is mainly a straightforward extension of the extended semantics. The only difference is that any set of equations that be-

```
rl k( (LabVal, EraVal) -> (if(S,S') -> K)) lenv(CL,CEraLab)
   => k(S -> K) lenv(CL join LabVal, CEraL join EraVal) .
rl k( (LabVal, EraVal) -> (if(S,S') -> K)) lenv(CL,CEraL)
   => k(S' -> K) lenv(CL join LabVal, CEraL join EraVal) .
```

Figure 7.13: Abstract rules for the if-then-else statement.

come non confluent (because they have the same left-hand side in the abstract semantics) is transformed into rules, as we did in Section 6.4 for abstractly analyse non–interference. The abstract rules associated to the equations of the extended semantics of the `if-then-else` statement are shown in Figure 7.13. Note that these rules are very similar to the rules of the `if-then-else` statement for abstractly analyse non–interference, shown in Figure 6.22 in Section 6.4. However, here the rules set both, the confidentiality level and the erasure label of the context.

By abuse, we denote the abstraction of a rule $\alpha(\{l\}) \rightarrow \alpha(\{r\})$ by $\alpha(\{l\} \rightarrow \{r\})$. $\mathcal{P}_{\text{Java}}$ denotes the sort of Java programs P_{Java} (i.e. $P_{\text{Java}} \in \mathcal{P}_{\text{Java}}$).

Definition 9 (Abstract rewriting)**.** We define the abstract rewriting relation for erasure $\rightarrow_{\text{Java}^{\#}} \subseteq (\mathcal{P}_{\text{Java}} \times \wp(\mathsf{State}^{E})) \times (\mathcal{P}_{\text{Java}} \times \wp(\mathsf{State}^{E}))$ by $\langle P_{\text{Java}_1}, S\,S\,t_1 \rangle \rightarrow_{\text{Java}^{\#}} \langle P_{\text{Java}_2}, S\,S\,t_2 \rangle$ if $\exists u \in S\,S\,t_1, \exists v \in S\,S\,t_2$ s.t. $\langle P_{\text{Java}_1}, u \rangle \rightarrow_{\text{Java}^{\text{E}}} \langle P_{\text{Java}_2}, v \rangle$.

Note that this applies to $S\,S\,t_1 = \alpha(\{u\})$ and $S\,S\,t_2 = \alpha(\{v\})$, given $u \in \alpha(\{u\})$ and $u \in \alpha(\{u\})$. We denote by $\rightarrow^{*}_{\text{Java}^{\#}}$ the extension of $\rightarrow_{\text{Java}^{\#}}$ to multiple rewrite steps.

Our abstraction consists of transforming equations into rules and getting rid of the value component of states as we did in Section 6.4. In order to guarantee that the abstract semantics correctly (over-)approximates the extended semantics, we need to prove that:

1. The abstraction is correct regarding the relation between the concrete and abstract domains of program states, based on the abstraction function α. This is done by proving that this abstraction function α and the corresponding concretization function γ constitute a Galois insertion [Cousot and Cousot, 1979; Cousot and Cousot, 2002; Cousot, 2004]. This result is formalized in Theorem 10 below.

2. All extended program traces have corresponding abstract program traces, such that no extended program trace is disregarded. The transformation of a set of equations (which are confluent and terminating modulo

axioms) into rules preserves the execution traces. The removal of the value component of the states does not eliminate execution traces either. This result is formalized in Theorem 11 [Alba-Castro et al., 2010b] (on page 149). This theorem (and its proof) are very similar to Theorem 6 in Chapter 6.

The abstraction function $\alpha : \wp(\mathsf{State}^E) \rightarrow \wp(\mathsf{State}^E)$ is a simple homomorphic extension to sets of states of the function $\alpha : \mathsf{Value} \times SecLabels \times EraLabels \rightarrow \mathsf{Value} \times SecLabels \times EraLabels$, $\alpha(\{\langle Val, \mathtt{SecLab}, \mathtt{EraLab}\rangle\}) = \langle \mathsf{Value}, \mathtt{SecLab}, \mathtt{EraLab}\rangle$, where $SecLabels$ denotes the sort $Labels \cup LabelChange$. Given $SValue^E \in \wp(\mathsf{Value} \times SecLabels \times EraLabels)$, i.e. $SValue^E = \{\langle Value, \mathtt{SecLab}, \mathtt{EraLab}\rangle\}$ with $\mathtt{SecLab} \in \{\mathtt{Low}, \mathtt{High}, \mathtt{Low} \gg \mathtt{High}, \mathtt{High} \gg \mathtt{Low}\}$, $\mathtt{EraLab} \in \{\emptyset, \mathtt{Low} \nearrow \mathtt{High}, \mathtt{Low} \nearrow \top, \mathtt{High} \nearrow \top\}$, $\alpha(SValue^E) = \mathsf{Value} \times \{\mathtt{SecLab}\} \times \{\mathtt{EraLab}\}$; note that all tuples of set $SValue^E$ have the same and unique confidentiality level, and erasure policy label w.r.t. the confidentiality level and erasure policy label of the corresponding variable; we denote the set $\mathsf{Value} \times \{\mathtt{SecLab}\} \times \{\mathtt{EraLab}\} \in \wp(\mathsf{Value} \times SecLabels \times EraLabels)$ by $\#\langle \mathtt{SecLab}, \mathtt{EraLab}\rangle$, such that $\alpha(\{\langle \mathtt{Val}, \mathtt{SecLab}, \mathtt{EraLab}\rangle\}) = \#\langle \mathtt{SecLab}, \mathtt{EraLab}\rangle$; this means, for instance, that $\#\langle \mathtt{High}, \mathtt{Low} \nearrow \mathtt{High}\rangle$ denotes $\alpha(\{\langle \mathtt{Val}, \mathtt{High}, \mathtt{Low} \nearrow \mathtt{High}\rangle\})$.

The corresponding concretization function $\gamma : \wp(\mathsf{State}^E) \rightarrow \wp(\mathsf{State}^E)$, is the homomorphic extension to sets of states of the function $\gamma : \wp(\mathsf{Value} \times SecLabels \times EraLabels) \rightarrow \wp(\mathsf{Value} \times SecLabels \times EraLabels)$, given $SSecLabel \in \wp(\mathsf{Value} \times SecLabels \times EraLabels)$ such that $SSecLabel = \mathsf{Value} \times \{\mathtt{SecLab}\} \times \{\mathtt{EraLab}\}$, $\gamma(SSecLabel) = SSecLabel$; $\gamma(SSecLabel) = \mathsf{Value} \times \{\mathtt{SecLab}\} \times \{\mathtt{EraLab}\}$, e.g. $\gamma(\#\langle \mathtt{SecLab}, \mathtt{EraLab}\rangle) = \#\langle \mathtt{SecLab}, \mathtt{EraLab}\rangle$, for instance, $\gamma(\#\langle \mathtt{High}, \mathtt{Low} \nearrow \mathtt{High}\rangle) = \mathsf{Value} \times \{\mathtt{High}\} \times \{\mathtt{Low} \nearrow \mathtt{High}\}$.

Theorem 10. The abstraction function $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ (or $\alpha : \wp(\mathsf{State}) \rightarrow \mathsf{State}^\#$) and the corresponding concretization function $\gamma : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ (or $\gamma : \mathsf{State}^\# \rightarrow \wp(\mathsf{State})$) satisfy that, for all $S \in \wp(\mathsf{State})$ and $S^\# \in \mathsf{State}^\#$, it holds $\alpha(S) \sqsubseteq S^\#$ if and only if $S \subseteq \gamma(S^\#)$,

Proof. In order to prove that the functions α and γ are such that $\langle \alpha, \gamma \rangle$ is a Galois insertion, we can prove that the α and its corresponding γ functions satisfy the *monotonicity*, *deflationary*, and *non-information loss* properties [Cousot and Cousot, 1979; Cousot and Cousot, 2002; Cousot, 2004], at the variable level.

The proof is perfectly analogous to the proof of Theorem 5. That proof is extended here in order to consider the new erasure policy labels.

□

This result can be extended homomorphically to $\alpha : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$, so that the pair of functions $\langle \alpha, \gamma \rangle$ constitutes a Galois insertion.

Theorem 11. If $\langle P_{\text{Java}}, St_1^E \rangle \to^*_{\text{Java}^{\text{E}}} \langle St_2^E \rangle$, then there exists $SSt_3 \in \wp(\mathsf{State}^E)$ s.t. $\langle P_{\text{Java}}, \alpha(\{St_1^E\}) \rangle \to^*_{\text{Java}^{\#}} \langle SSt_3 \rangle$ and $St_2^E \in SSt_3$.

Proof. The proof is done by induction on the length n of the extended program trace or rewriting sequence denoted by $\to^*_{\text{Java}^{\text{E}}}$, and is very similar to the proofs of Theorems 2 and 6.

□

Given that the abstract semantics correctly (over-)approximates the extended semantics, we have to prove that our abstract erasure certification technique is sound, i.e. that abstract erasure implies extended concrete erasure. This is proven in Theorem 12, as follows [Alba-Castro et al., 2010b].

Theorem 12 (Abstract Strong Erasure Soundness)**.** Given a Java program P_{Java}, P_{Java} complies with the erasure policy (Definition 7) if for all $SSt_1 \in \wp(\mathsf{State}^E)$ s.t. $\langle P_{\text{Java}}, SSt_1 \rangle \mapsto^*_{\text{Java}^{\#}} \langle SSt_2 \rangle$, for all $St \in SSt_2$, and for all variables $var \in Vars(P_{\text{Java}})$, either $St[var] = \langle Val, \mathtt{Lab}, \emptyset \rangle$, or $St[var] = \langle Val, \mathtt{High}, \mathtt{Low} \nearrow \mathtt{High} \rangle$, for a value Val and a label $Lab \in \{\mathtt{Low}, \mathtt{High}, \mathtt{Low} \gg \mathtt{High}, \mathtt{High} \gg \mathtt{Low}\}$.

Proof. By contradiction. Let us assume that P_{Java} does not comply with the erasure policy i.e., there exists St_1^E with $\langle P_{\text{Java}}, St_1^E \rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St_2^E \rangle$ and $var \in Vars(P_{\text{Java}})$ s.t. either $St_2^E[var] = \langle Val, \mathtt{Low} \gg \mathtt{High}, \mathtt{Low} \nearrow \mathtt{High} \rangle$, or $St_2^E[var] = \langle Val, \mathtt{Low} \gg \top, \mathtt{Low} \nearrow \top \rangle$, or $St_2^E[var] = \langle Val, \mathtt{Low} \gg \top, \mathtt{High} \nearrow \top \rangle$, or $St_2^E[var] = \langle Val, \mathtt{High} \gg \top, \mathtt{High} \nearrow \top \rangle$ for a value Val. Since $\langle P_{\text{Java}}, St_1^E \rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St_2^E \rangle$, by Theorem 11, there exists $SSt_3 \in \wp(\mathsf{State}^E)$ s.t. $\langle P_{\text{Java}}, \alpha(\{St_1^E\}) \rangle \to^*_{\text{Java}^{\#}} \langle SSt_3 \rangle$ and $St_2^E \in SSt_3$. This contradicts the assumption that for all $St \in SSt_3$, and for all variables $var \in Vars(P_{\text{Java}})$, either $St[var] = \langle Val, \mathtt{Low}, \emptyset \rangle$, or $St[var] = \langle Val, \mathtt{Low} \gg \mathtt{High}, \emptyset \rangle$, or $St[var] = \langle Val, \mathtt{High}, \emptyset \rangle$, or $St[var] = \langle Val, \mathtt{High} \gg \mathtt{Low}, \emptyset \rangle$, or $St[var] = \langle Val, \mathtt{High}, \mathtt{Low} \nearrow \mathtt{High} \rangle$ for a value Val .

□

Given that the abstract semantics correctly (over-)approximates the extended semantics, we have to demonstrate that the abstract erasure with non–interference certification approach is sound, i.e. that abstract erasure with non–interference implies extended concrete erasure with non–interference. This statement is proven in Theorem 13, as follows [Alba-Castro et al., 2010b].

Theorem 13 (Abstract Strong Erasure with Non–Interference Soundness)**.** Given a Java program P_{Java}, P_{Java} in non–interferent (Definition 4) and complies with the erasure policy (Definition 7) if for all $SSt_1 \in \wp(\mathsf{State}^E)$ s.t. $\langle P_{\text{Java}}, SSt_1\rangle \mapsto^*_{\text{Java}^\#} \langle SSt_2\rangle$, for all $St \in SSt_2$, and for all variables $var \in Vars(P_{\text{Java}})$, either $St[var] = \langle Val, \mathtt{Lab}, \varnothing\rangle$, or $St[var] = \langle Val, \mathtt{High}, \mathtt{Low} \nearrow \mathtt{High}\rangle$ for a value Val and a label $Lab \in \{\mathtt{Low}, \mathtt{High}, \mathtt{High} \gg \mathtt{Low}\}$.

Proof. By contradiction. Let us assume that P_{Java} does not comply with non–interference with the erasure policy i.e., there exists St_1^E with $\langle P_{\text{Java}}, St_1^E\rangle \mapsto^*_{\text{Java}^E} \langle St_2^E\rangle$ and $var \in Vars(P_{\text{Java}})$ s.t. either $St_2^E[var] = \langle Val, \mathtt{Low} \gg \mathtt{High}, \varnothing\rangle$, or $St_2^E[var] = \langle Val, \mathtt{Low} \gg \mathtt{High}, \mathtt{Low} \nearrow \mathtt{High}\rangle$, or $St_2^E[var] = \langle Val, \mathtt{Low} \gg \top, \mathtt{Low} \nearrow \top\rangle$, or $St_2^E[var] = \langle Val, \mathtt{Low} \gg \top, \mathtt{High} \nearrow \top\rangle$, or $St_2^E[var] = \langle Val, \mathtt{High} \gg \top, \mathtt{High} \nearrow \top\rangle$ for a value Val. Since $\langle P_{\text{Java}}, St_1^E\rangle \mapsto^*_{\text{Java}^E} \langle St_2^E\rangle$, by Theorem 11, there exists $SSt_3 \in \wp(\mathsf{State}^E)$ s.t. $\langle P_{\text{Java}}, \alpha(\{St_1^E\})\rangle \rightarrow^*_{\text{Java}^\#} \langle SSt_3\rangle$ and $St_2^E \in SSt_3$. This contradicts the assumption that for all $St \in SSt_3$, and for all variables $var \in Vars(P_{\text{Java}})$, either $St[var] = \langle Val, \mathtt{Low}, \varnothing\rangle$, or $St[var] = \langle Val, \mathtt{High}, \varnothing\rangle$, or $St[var] = \langle Val, \mathtt{High} \gg \mathtt{Low}, \varnothing\rangle$, or $St[var] = \langle Val, \mathtt{High}, \mathtt{Low} \nearrow \mathtt{High}\rangle$ for a value Val .

□

The following example illustrates the mechanization of the Java erasure with or without non–interference abstract analysis [Alba-Castro et al., 2010b].

Example 23. Consider Example 20 together with the constructor method:

```
public Eraserclass(int xp,int yp,int zp) {
     /*@ setLabel(xp,High); setLabel(yp,High);@*/
     xh =xp; yh =yp; zl =zp; }
```

and the main class:

```
class Erasure3 {
     static  Eraserclass t = new Eraserclass(3, -3, 0);
     public static void main(String[] args) {
          t.setzl( 6);
          t.mE3(); } }
```

In the search command below, we ask for all possible abstract final program states.

```
search in PGM-SEMANTICS-ABSTR :
java((preprocess(EX5-MAUDE) noType . 'main <new string [i(0)]> noVal)) =>! JS:JavaState .
Solution 1 (state 0)
store(...[l(2),o(f([t(t('Eraserclass)),f(['xh,l(6)] ['yh,l(7)]
  ['zl,l(8)])])...),0]..[l(6),High,0] [l(7),High,0] [l(8),Low,0])..)
No more solutions.
```

After the execution of the program "`Erasure3.java`", the search command returns that only one extended final state of the Java program is possible, which has the values `High`, `High`, and `Low`, in the memory locations `l(6)`, `l(7)`, and `l(8)` which correspond to the variables `xh`, `yh`, and `zl`, respectively. These final variable values show that the erasure policy is fulfilled.

The first work that addressed information erasure from an information flow perspective was [Chong and Myers, 2005], where erasure is combined with downgrading (declassification) policies. Erasure policies were defined in [Chong and Myers, 2005; Chong and Myers, 2008] with conditions (i.e. and *var*: $L_1 \overset{c}{\nearrow} L_2$), and are enforced only when condition c is satisfied during program execution. In [Chong and Myers, 2008], the Jif programming language that extends Java source was further extended in order to handle erasure and declassification. In this thesis, we provide a less precise but simpler and convenient alternative to explicit conditions. Both papers [Chong and Myers, 2005; Chong and Myers, 2008] analyzed erasure with non–interference whereas we can analyze erasure with or without non–interference. Another advantage of our proposal is that it can be applied to real Java programs by just inserting the code annotations that express the required policies. Moreover, we do not need runtime policy enforcement since our hybrid policy verification methodology is based on static as well as dynamic mechanisms. Finally, we provide an effective implementation that supports program certification.

A static analysis approach to enforce erasure was proposed in [Hunt and Sands, 2008], which extends the flow sensitive type system for non–interference of [Hunt and Sands, 2006] to enforce erasure with non–interference. This work introduces erasure with and without non–interference for terminating deterministic programs with input and output channels. It also introduces a block structured input command "`input x: a ↗ b in C`", where `x` is a variable, `a` is an input channel with confidentiality level `a`, `b` is a higher confidentiality level, and `C` is a command. This local end-to-end erasure condition means that command `C` should erase the input `x` obtained from

channel `a` to the level `b`. Local erasure with non–interference is formulated as a reclassified non–interference condition for deterministic and terminating programs without input and output channels Then, this local condition is extended to achieve global erasure with non–interference. However, this proposal does not enforce erasure without non–interference, nor includes procedure neither function invocations, and it does not consider high guarded loops, and for the best of our knowledge, it has not been implemented yet.

In [Hansen and Probst, 2006], the notion of non–interference was extended to consider a simple erasure policy to be applied to the Carmel Core language, an abstract version of the Java Card bytecode (a subset of the Java bytecode). The simple erasure policy `Low`↗`High` indicates that the erasure of a whole level, instead of some program variables, should be done before the program ends its execution. They analyse erasure as an extension of non–interference but do not consider the erasure of `High`–labeled variables. Moreover, they do not consider how to verify nor enforce this erasure policy.

The work [L. Jiang and Pan, 2007] studied erasure in multi-threaded programs written in the prototypical MWL language. The considered erasure policies state the erasure of program variables as an upgrading of low–labeled variables into high–labeled ones. Erasure is observed as an extension of non–interference, namely as a reclassification of some low-labeled variables but the erasure of `High`–labeled variables is not considered. As far as we know, this proposal is not yet implemented.

As suggested in Chapter 6 in case of non–interference policies, equational abstractions [Meseguer et al., 2003] can be used to specify the abstract extended Java semantics for erasure policies with the same advantages and disadvantages mentioned there.

7.5 Experimental evaluation

The certification methodology presented here has been implemented in Maude: the certification system JavaPCC (8) can verify non–interference alone and erasure with or without non–interference. The system is a new, totally redesigned implementation of the technique of Chapter 6 in order to (i) deal with erasure, (ii) improve both functionality and performance, and (iii) make

it easier to use and extend. Recall that JavaPCC is publicly available on-line[2]. The prototype system offers a rewriting-based program certification service, which is able to analyze confidentiality global program properties related to non–interference and erasure.

The Java operational semantics in rewriting logic that we have used is modular and has 2635 lines of code in 4 files [Farzan et al., 2007]. We have modified less than 25 of the 1527 lines of code in the main file of the original Java semantics. The abstract operational Java semantics was developed as a source–to–source transformation in rewriting logic and consists of 419 lines of extra code with 123 equations and 8 rules.

Programs	Source Size LOC	Source Cicl. Comp.	Full Cert. Size (Kb)	Red. Cert. Size (Kb)	Size Rel. (Red. / Source)	Full Cert. Gen. Time (ms)	Red. Cert. Gen. Time (ms)
19	90	3	2991.911	14.371	0.006	1765	88
11	34	2	1377.158	6.253	0.04	1156	313
20	51	1	1017.592	6,253	0.004	407	22
30	46	2	1127.606	10.373	0.007	581	53
31	88	3	2820,719	18,86	0.008	1680	97
32	47	1	1020.961	6,253	0.004	437	35
29	117	192	20009.28	389.62	0.111	30409	1778

Table 7.1: Certificate sizes and certification times.

We have benchmarked our implementation including the examples used in this Chapter. In Table 7.1, we study three key points for the practicality of our approach: the size of the reduced certificate versus the Java source code, the size of the reduced certificate versus the size of the full certificate and the relative efficiency of producing certificates. The experiments have been performed on a MacBook with 2 Gb RAM.

The columns "Source Size LOC", and "Source Cicl. Comp.", show two source metrics, namely the lines of source code and the cyclomatic complexity (number of paths), respectively. The two columns for "Full Cert." show the size in Kbytes (similarly for the two columns of "Red. Cert.") and the generation time, respectively, for the full certificates. Running times are given in milliseconds and were averaged over a sufficient number of iterations.

Programs 19 and 20 are respectively the ones of Examples 19 and 20 previously introduced in this Chapter. Program 11 is the interferent Example 11

[2] The JavaPCC system is publicly available on-line at `http://zenon.dsic.upv.es:8080/certificateX/`.

of Chapter 6. Program 30 (given in the Appendix) is adapted from [Hunt and Sands, 2008] and does not comply with the required erasure policy. Program 31 (given in the Appendix) is a version of Example 19 that does not erase the required variables either. Program 32 (given in the Appendix) is adapted from [Hunt and Sands, 2008] and it does comply with its erasure policy. Program 29 (given in the Appendix) includes three methods that invoke nine simple example methods (seven interferent methods and two non-interferent ones) taken from [Sabelfeld and Sands, 2009; Warnier, 2005]. Since Programs 30 and 31 are counterexamples, the figures correspond to the generation time and size of the abstract traces. The certificates of Programs 29 and 19 correspond to the full annotated versions.

Note the correlation between both the source size and the cyclomatic complexity, with the size and generation time of the certificates. The experiments are very encouraging since the reduction in size of the certificate is very significant (at least two orders of magnitude in all cases), the quotient "Red. Cert. Size/Full Cert. Size" ranging from 1.94% in Example 29 to 0.45% for Example 11. When the time employed to generate the full and reduced certificates is compared, the reduced certificate generation time takes only 5.8% of the full certificate generation time for the biggest source size of the full annotated version of Example 29.

Chapter 8

The JavaPCC Certification Environment

This chapter presents the proof of concept derived from the methodology introduced in this thesis. The proof of concept consists of two Web tools. The tools allow one to certificate properties of Java source code by providing appropriate abstract domains for different safety properties while hiding the technical details of the method to the user. The tools has been devised to be easily extendable to other properties and domains. Both tools can generate the three different certificates described in Chapter 4. The *full certificate* includes all Maude equations and rules (together with the corresponding matching substitutions) used in the rewriting proof. According to the different treatment of rules and equations in Maude, an extremely *reduced rules certificate* can be delivered by just recording the rewrite steps given with the rules, while the rewritings with the equations are omitted. This is justified by the fact that, in Maude, reducing with equations is deterministic (under the assumptions of confluence and termination) and also because Maude is very efficient at doing it. Finally, the *reduced labels certificate* only records the labels of the applied rules.

The Web tool introduced in [Alba-Castro et al., 2009b] implements the abstract certification technique of Chapter 5 that considers user defined integer arithmetic type safety properties of Java integer functions. This tool can also certify Java methods that obey input–output data non–interference user defined policies [Alba-Castro et al., 2009a]. The tool is publicly available at `http://zenon.dsic.upv.es:8080/rewritingLogic/control` The tool automatically generates the Maude encoding of the abstraction, as well as the search command containing the initial state that includes the wrapped, supplemented Java program, together with the final state, which depends on the expected method outcome.

A snapshot of this Web tool is shown in Figure 8.1. The snapshot shows a non–interference example and its corresponding reduced labels certificate.

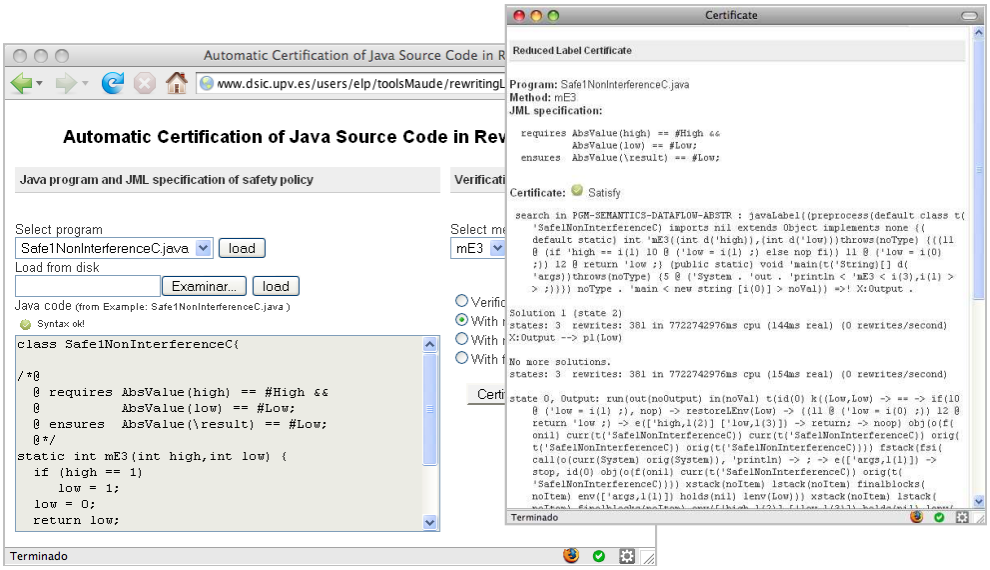

Figure 8.1: Web interface snapshot.

A completely redesigned, easy to use Web tool was developed in [Alba-Castro et al., 2010c] that implements the abstract certification technique for confidentiality of complete Java classes introduced in Chapters 6 and 7, and applies to certify non–interference and erasure policies in sequential, deterministic Java programs. The certification system JavaPCC can verify non–interference alone and erasure with or without non–interference. The system is a new, totally redesigned implementation of the technique of Chapter 6 introduced in [Alba-Castro et al., 2010a] in order to (i) deal with erasure, (ii) improve both functionality and performance, and (iii) make it easier to use and extend. The system is publicly available at `http://zenon.dsic.upv.es:8080/certificateX/`.

Figure 8.2 shows the JavaPCC main page, with an example code that complies with the JML-like annotated non–interference policy. The Java code can be loaded from a file containing the Java program and the JML-like annotations, or by selection of one of several predefined examples. The security certificate (either full or reduced) is automatically generated by the tool, by clicking on the Certify! button. Figure 8.3 shows a full certificate snapshot and the corresponding reduced rules certificate snapshot.

The main features of the JavaPCC Web tool are:

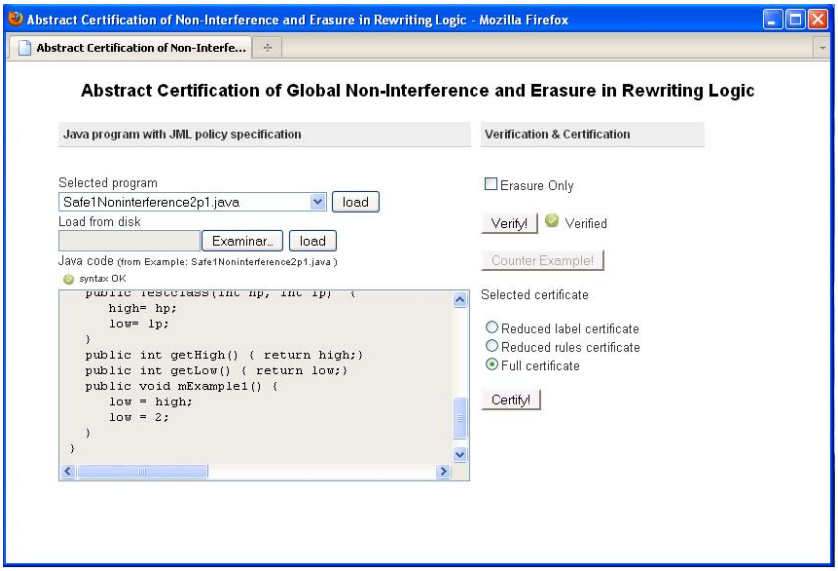

Figure 8.2: JavaPCC main page snapshot.

- It is implemented in the Maude programming language, which implements rewriting logic, and provides a formal analysis infrastructure (such as state-space breadth-first search) with competitive performance (see [Farzan et al., 2004a]). Note that Maude could also be used as the infrastructure required for the proof validation process at the consumer side. Actually, it suffices to check that each abstract rewriting step in the certificate is valid and no other valid rewritings have been disregarded, which essentially amounts to using the matching infrastructure within the rewriting engine. This is simple, trustworthy, and based on well-understood engineering and mathematical principles.

- The Java operational semantics has 2635 lines of Maude code [Şerbănuţă et al., 2009]. The abstract operational Java semantics was developed as a source–to–source transformation and consists of 419 lines. If the code consumer has to assume that our Java abstract semantics is trusted, but not the original Java semantics, neither the standard reduction engine, then, in our current system, the *trusted computing base* (TCB)[1] is less than a sixth of the size of the original Java semantics

[1]Recall that the TCB is the part of the code that is used to check if other code can be

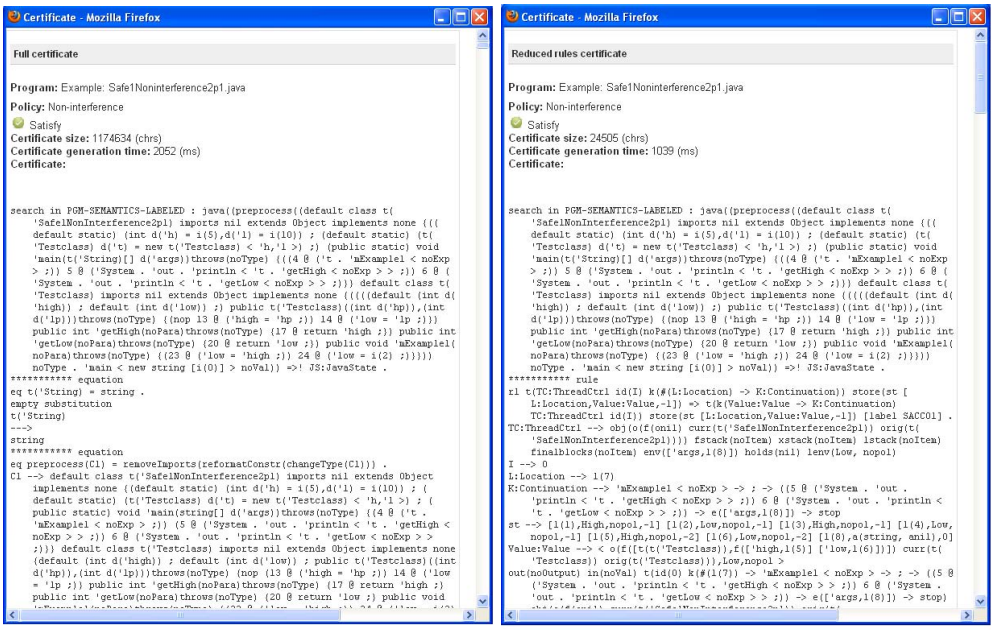

Figure 8.3: Full and reduced rules certificate snapshots.

and even much smaller than other verification and certification systems -see PCC Section 1.1.1, numeral 3 on page 6 and FPCC Subsection on page 10).

- In order to automate the processing of JML-like clauses, the tool uses the Javacc compiler construction tool together with a subset of the JML grammar. Then it generates the encoded Maude abstraction and the Maude search command that contains the Java program that includes the Java operators corresponding to the `erase` annotations if any. It also uses the Java wrapper program that is available at [Farzan et al., 2007] to transform the supplemented Java program into a Maude term to build the initial state.

- The Web interface allow us to both make the abstract certificate technique publicly available on-line and to hide the technical details to every possible user.

safely run, and that is assumed it can be trusted.

If the code consumer does not trust the original Java semantics, neither the standard reduction engine (or the Maude interpreter), we have to include them into the TCB. Then, the size of the TCB of our framework for source level certificate validation is increased, including: i) the size of the Maude core interpreter (the Maude prelude has 2384 lines of code and the Maude core 102.271 lines of code); ii) the size of the original Java semantics (2635 lines of code); and the size of our abstract Java semantics (419 lines of code). Then, the TCB total size (at source level) is 107.709 lines of code.

In order to safely run the executable code that corresponds to the certified source code, the code consumer has to use a reliable Java infrastructure, i.e. a compiler that translates Java source code to executable bytecode, and the JVM that corresponds to her computing platform. This means that the code consumer needs, either to trust the corresponding available compiler, i.e. to include this compiler into the TCB, or to use a verified compiler (i.e. a compiler that produces executable code that is semantically equivalent to the source program). In the case of the C language, there are some verified and certified compilers for subsets of the C language that could be used (for instance the Compcert compiler [Leroy, 2009]). However, as far as we know there is no certified compiler for the full Java language. Moreover, since there exists no certified Java Virtual Machine for any platform, we need to include into the TCB the JVM itself as in [Appel and Wang, 2002; Franz et al., 2003; Pirzadeh and Dubé, 2008b]. This is also the case of the PCC frameworks that certify bytecode of [Wildmoser et al., 2005; Beringer et al., 2003; Gilmore and Prowse, 2005; Chander et al., 2005b; Besson et al., 2005] (see Sections 1.1.1, and 1.1.2).

Let's talk about including the Java compiler and the JVM into the TCB. The Java SDK compiler consists of 32.000 lines of code [Appel and Wang, 2002]. The size of the JVM, which was measured by Appel and Wang [Appel and Wang, 2002], ranges from 54.200 to 229.000 lines of code. The standard classic JVM version for Java 2 version 1.3.0 with the non-optimizing compiler has 54.200 lines of code [2] and it does not include the security API. If we include the standard security API, this standard security JVM infrastructure has 87.200 lines of code [Appel and Wang, 2002]. The standard JVM version for Java 2 version 1.3.0 with the optimizing compiler (HotSpot) has 229.100 lines of code, and the corresponding secure JVM has 257.600 lines of code

[2] The safe JVM includes the just-in-time compiler, the byte-code verifier, the linker, the garbage collector, and the core run-time system

[Appel and Wang, 2002]. Therefore, when we include the Java infrastructure into the TCB, the size of our TCB increases in 32.000 lines of code because of the compiler, and an additional 87.000 (or 257.600) lines of code increment because of the JVM. Then, the total size of the TCB at the bytecode level is 226.909 (or 397.309) lines of code. The source level component (the Maude interpreter, the standard Java semantics and our abstract semantics) of this TCB, amounts for 47% (or 27%) of the TCB total size.

We benchmarked the system by using several program examples available within the package distribution (most of them have been documented in this thesis). As we showed in Sections 5.3, 6.5 and 7.5, the experiments are very encouraging, since they show that the reduction in size and generation time of the certificates is very significant in all cases.

Chapter 9

Conclusions

As far as we know, this thesis develops the first sound and fully automatic certification technique that applies to the verification of sequential deterministic source Java code, regarding safety and confidentiality (security) properties. Our methodology relies on an abstract extended semantics for Java written in rewriting logic that can be model checked in Maude by using Maude's breadth-first search space exploration. This technique can certify some simple integer arithmetic properties and non–interference of Java function methods. It can also certify non–interference and erasure with and without non–interference of *complete* Java programs.

The proposed methodology features quality attributes (notably reliability and security, but also good performance) through rigorous mechanisms which integrate a wide range of well-established programming language techniques (abstract interpretation, program semantics, meta-programming, etc). Our approach is based on the rewriting logic semantics specification of the full Java 1.4 language given in [Farzan et al., 2007; Şerbănuţă et al., 2009]. Since we inherit from Maude and the Java rewriting semantics its competitive performance (see [Farzan et al., 2004a]), we have a scalable technique that can be further refined to certifying industrial complex Java programs with exceptions and threads. Besides, different safety policies can be defined by using different (abstract) terms denoting the states that should not be reached. Nevertheless, that semantics does not support most Java API classes, and thus, the analysis and verification of real Java programs that use the Java API is not yet possible.

The thesis provides a program-level extended semantic definition of Java arithmetic properties, as well as non–interference, and erasure with and without non–interference, together with a corresponding abstract notion of the considered properties, and a rigorous connection between the approximated abstract properties and the model that we consider. In the extended semantics, the considered properties (e.g. confidentiality) become *safety* properties, and we formally demonstrate that the *safety* property in the extended seman-

tics entails the corresponding semantic confidentiality (security) property in the standard Java semantics, i.e. the confidentiality analysis is proved *sound*. The abstract over-approximated semantics is also proved *sound*, as well as the considered abstract confidentiality properties.

For both kinds of properties considered in this thesis (simple integer arithmetic and confidentiality), the technique is only an approximation (i.e. incomplete), as shown by some of the examples considered before. However, this is not a limitation of our approach but a natural consequence of the undecidability of the considered properties. In the case of integer arithmetic, the *incompleteness* can be sometimes reduced by refining the abstraction at the program state level, for instance by considering relational abstraction of program variables as shown in Section 5.2.2. On the other hand, regarding confidentiality properties, the *incompleteness* could be reduced by refining the abstraction at the program trace level, i.e. by recording not only the initial and final variable confidentiality labels but also the intermediate ones.

Currently, the abstract rewriting logic Java computations are recorded in order to construct the certificate that ensures that the program satisfies the desired property. The certificates are abstract rewriting sequences that contain the rewriting steps, together with the applied equations and rules of the abstract Java semantics. The JavaPCC system can generate full certificates that include all rewrite steps, or reduced certificates that omit the matching substitution and applied rules. The system was benchmarked by using several program examples. The experiments are very encouraging, and show that the reduction in certificate size and generation time is very significant in all cases.

Certificates are encoded as rewriting sequences which can be checked in the abstract Java semantics written in Maude at the consumer side by standard reduction, thus providing support for Proof-Carrying Code, at the source code level. By turning a potentially infinite state space of a Java program into a finite abstract space, the abstract semantics not only makes the approach feasible, but also greatly reduces the size of the certificates that must be checked at the consumer's end.

While most PCC proposals deliver certificates for assembler code or bytecode, i.e. executable code, our certificates are produced at the source level code, similarly to the ACC proposal of [Albert et al., 2005a] that certificates Ciao programs at the source code level too (see Section 1.1.2). Also the Tinman framework of [Mok and Yu, 2002a; Mok and Yu, 2002b] delivers resource consumption certificates at the source level for C programs (see Sec-

tion 1.1.1), and the code synthesis proposals [Whalen et al., 2002; Schumann, 2003; Denney and Trac, 2008; Vargun, 2006; Alpuente et al., 2010a] generate certificates at source code level as well, for the high-level programming languages C, C++, Oz and Maude (see Section 1.1.5).

Since the Maude interpreter (or an standard rewriting logic engine) and the original Java semantics both are not certified, they have to be included into the TCB that the code consumer has to use in order to check our source level delivered certificate. The inclusion of the Maude interpreter in the TCB is comparable to other proposals that include the interpreter or compiler of some high-level language. For instance, the certified program analysis in [Chang et al., 2006; Chlipala, 2007] includes the Ocaml compiler (see Section 1.1.3) and the code synthesis of [Alpuente et al., 2010a; Alpuente et al., 2010b] includes the Maude interpreter (see Section 1.1.5). In order to avoid the use of the full Core Maude interpreter (or an standard rewriting logic engine) a simple standard reduction engine could be developed by the code consumer. This simpler reduction engine will reduce the TCB.

We would like to emphasize that this certification methodology can also be extended to other programming languages by simply replacing the concrete semantics by a semantics for the programming language at hand. In particular, the developed technique can be extended to analyze bytecode programs in order to deliver certificates at bytecode level. This way, we could avoid trusting the available Java compilers, or to develop a verified and certified Java compiler (i.e. a compiler that produces executable code that is semantically equivalent to the source program).

Currently, our approach cannot deal with interferent programs that have object pointer aliasing (see Section 6.5). As future work, we plan to extend the confidentiality analysis in this thesis in order to consider non–interference of programs with pointer aliasing, exceptions and threads. Also, it could be further extended for more relaxed non–interference policies in order to support the release or downgrading of some secret data, i.e. declassification [Sabelfeld and Sands, 2009]. The analysis can also be improved by considering more sophisticated confidentiality attackers which would observe not only part of the initial and final program states, but also intermediate states, i.e. object states immediately before and after any invocation of its public methods [Zanardini, 2007].

To the best of our knowledge, this is the first sound implemented framework that effectively supports verification and certification of non–interference

and erasure policies of sequential non-deterministic Java programs.

Bibliography

[Ádám Darvas and Müller, 2007] Ádám Darvas and Müller, P.: 2007, *Formal encoding of JML Level 0 specifications in Jive*, Technical Report 559, ETH Zurich

[Ahrendt et al., 2007] Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P., and Schmitt, P.: 2007, Verifying object-oriented programs with KeY: A tutorial, in *Formal Methods for Components and Objects*, Vol. 4709 of *Lecture Notes in Computer Science*, pp 70–101, Springer Berlin / Heidelberg, Germany

[Alba-Castro et al., 2008] Alba-Castro, M., Alpuente, M., and Escobar, S.: 2008, Automatic certification of Java source code in rewriting logic, in *12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007)*, Vol. 4916 of *Lecture Notes in Computer Science*, pp 200–217, Springer-Verlag, Berlin Heidelberg, Germany

[Alba-Castro et al., 2009a] Alba-Castro, M., Alpuente, M., and Escobar, S.: 2009a, Automated certification of non-interference in rewriting logic, in *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, Vol. 5596 of *Lecture Notes in Computer Science*, pp 182–198, Springer-Verlag, Berlin Heidelberg, Germany

[Alba-Castro et al., 2010a] Alba-Castro, M., Alpuente, M., and Escobar, S.: 2010a, Abstract certification of global non-interference in rewriting logic, in *Proc. 8th Int. Symp. Formal Methods for Components and Objects (FMCO 2009), Revised Lectures.*, Vol. 6286 of *Lecture Notes in Computer Science*, pp 105–124, Springer-Verlag, Berlin Heidelberg, Germany

[Alba-Castro et al., 2010b] Alba-Castro, M., Alpuente, M., and Escobar, S.: 2010b, Approximating non-interference and erasure in rewriting logic, in *12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2010) Sept. 23-26, Timisoara, Romania*, pp 124–132, IEEE Computer Society, Los Alamitos, CA USA

[Alba-Castro et al., 2010c] Alba-Castro, M., Alpuente, M., and Escobar, S.: 2010c, Confidentiality certification of source Java code in JavaPCC, in *Proceedings of the 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010)*, Vol. 35 of *Electronic Communications of the EASST*, To appear

[Alba-Castro et al., 2009b] Alba-Castro, M., Alpuente, M., Escobar, S., Ojeda, P., and Romero, D.: 2009b, A tool for automated certification of Java source code in Maude, in *Revised selected papers of Spanish Conference on Programming and Computer Languages, VIII Jornadas sobre Programación y Lenguajes (PROLE 2008)*, Vol. 248 of *Electronic Notes in Theoretical Computer Science*, pp 19–29, Elsevier, Amsterdam, Netherlands

[Albert et al., 2006] Albert, E., Arenas, P., Puebla., G., and Hermenegildo, M.: 2006, Reduced certificates for abstraction-carrying code, in *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2006)*, Vol. 4079 of *Lecture Notes in Computer Science*, pp 163–178, Springer-Verlag, Berlin Heidelberg, Germany

[Albert et al., 2011] Albert, E., Arenas, P., Puebla, G., and Hermenegildo, M.: 2011, Certificate size reduction in abstraction-carrying code, *Theory and Practice of Logic Programming* **FirstView**, 1

[Albert et al., 2005a] Albert, E., Puebla, G., and Hermenegildo, M.: 2005a, An abstract interpretation-based approach to mobile code safety, in *Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2004)*, Vol. 132 of *Electronic Notes in Theoretical Computer Science*, pp 113–129, Elsevier, Amsterdam, Netherlands

[Albert et al., 2005b] Albert, E., Puebla, G., and Hermenegildo, M.: 2005b, Abstraction-carrying code, in *In Proc. of LPAR'04*, Vol. 3452 of *Lecture Notes in Artificial Intelligence*, pp 380–397, Springer-Verlag, Berlin Heidelberg, Germany

[Albert et al., 2004] Albert, E., Puebla, G., Hermenegildo, M., and López-García, P.: 2004, Some techniques for automated, resource-aware distributed and mobile computing in a multi-paradigm programming system, in *EURO-PAR 2004 Conference*, Vol. 3149 of *Lecture Notes in Computer Science*, pp 21–37, Springer-Verlag, Berlin Heidelberg, Germany

[Albert et al., 2005c] Albert, E., Puebla, G., Hermenegildo, M., and López-García, P.: 2005c, Abstraction carrying code and resource-awareness, in *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pp 1–11, ACM Press, New York, NY, USA

[Alpuente et al., 2010a] Alpuente, M., Ballis, D., Baggi, M., and Falaschi, M.: 2010a, Completeness of unfolding for rewriting logic theories, in *Proceedings of the 2010 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC2010)*, SYNASC '10, pp 43–52, IEEE Computer Society, Los Alamitos, CA, USA

[Alpuente et al., 2010b] Alpuente, M., Ballis, D., Baggi, M., and Falaschi, M.: 2010b, A fold/unfold transformation framework for rewrite theories extended to CCT, in *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pp 43–52, ACM, New York, NY, USA

[Alpuente et al., 2011] Alpuente, M., Ballis, D., Espert, J., and Romero, D.: 2011, Backward Trace Slicing for Rewriting Logic Theories, in *The 23rd International Conference on Automated Deduction CADE 2011*, Vol. 6803 of *Lecture Notes in Computer Science*, pp 34–48, Springer Berlin / Heidelberg, Germany

[Amtoft et al., 2006] Amtoft, T., Bandhakavi, S., and Banerjee, A.: 2006, A logic for information flow in object-oriented programs, in *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp 91–102, ACM, New York, NY, USA

[Appel, 2001] Appel, A.: 2001, Foundational proof-carrying code, in *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pp 247–258, IEEE Computer Society, Los Alamitos,CA USA

[Appel and Felten, 2001] Appel, A. and Felten, E.: 2001, *Models for security policies in proof-carrying code*, Technical Report TR-636-01, Princeton University

[Appel et al., 2003] Appel, A., Michael, N., Stump, A., and Virga, R.: 2003, A trustworthy proof checker, *Journal of Automated Reasoning* **31(3-4)**, 231

[Appel and Felty, 2000] Appel, A. W. and Felty, A. P.: 2000, A semantic model of types and machine instructions for proof-carrying code, in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '00)*, pp 243–253, ACM, New York, NY, USA

[Appel and Mcallester, 2001] Appel, A. W. and Mcallester, D.: 2001, An indexed model of recursive types for foundational proof-carrying code, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **23(5)**, 657

[Appel and Wang, 2002] Appel, A. W. and Wang, D. C.: 2002, *JVM TCB: Measurements of the Trusted Computing Base of Java Virtual Machines*, Technical report, University of Princeton

[Aspinall et al., 2004] Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., and Stark, I.: 2004, Mobile resource guarantees for smart devices, in *Proc. of the International Workshop CASSIS 2004*, Vol. 3362 of *Lecture Notes in Computer Science*, pp 1–26, Springer-Verlag, Berlin Heidelberg, Germany

[Avvenuti et al., 2003] Avvenuti, M., C.Bernardeschi, and Francesco, N. D.: 2003, Java bytecode verification for secure information flow, *ACM SIGPLAN Notices* **38(12)**, 20

[Balser et al., 2000] Balser, M., Reif, W., Schellhorn, G., Stenzel, K., and Thums, A.: 2000, Formal system development with KIV, in *Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, FASE '00, pp 363–366, Springer-Verlag, London, UK

[Banerjee and Naumann, 2002] Banerjee, A. and Naumann, D.: 2002, Secure information flow and pointer confinement in a Java-like language, in *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pp 239–253, IEEE Computer Society, Los Alamitos, CA USA

[Banerjee and Naumann, 2005] Banerjee, A. and Naumann, D.: 2005, Stack-based access control and secure information flow, *Functional Programming* **15(2)**, 131

[Barbuti et al., 2002] Barbuti, R., Bernardeschi, C., and Francesco, N. D.: 2002, Abstract interpretation of operational semantics for secure information flow, *Information Processing Letters* **83(22)**, 101

[Barthe et al., 2007a] Barthe, G., Burdy, L., Grégoire, J. C. B., Huisman, M., Lanet, J., Pavlova, M., and Requet, A.: 2007a, Jack- a tool for validation of security and behaviour of Java applications, in *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, Netherlands, November 7-10,2006, Revised Lectures*, Vol. 4709 of *Lecture Notes in Computer Science*, pp 152–174, Springer-Verlag, Berlin Heidelberg, Germany

[Barthe et al., 2004] Barthe, G., D'Argenio, P., and Rezk, T.: 2004, Secure information flow by self-composition, in *Proc. of the Workshop on Computer Security Foundations (CSFW '04)*, pp 100–114, IEEE Computer Society, Los Alamitos, CA USA

[Barthe et al., 2006] Barthe, G., Naumann, D., and Rezk, T.: 2006, Deriving an information flow checker and certifying compiler for Java, in *2006 IEEE Symposium on Security and Privacy*, pp 230–242, IEEE Computer Society, Los Alamitos, CA USA

[Barthe et al., 2007b] Barthe, G., Pichardie, D., and Rezk, T.: 2007b, A certified lightweight non-interference Java bytecode verifier, in *Proc. 16th European Symposium on Programming (ESOP 2007)*, Vol. 4421 of *Lecture Notes in Computer Science*, pp 125–140, Springer-Verlag, Berlin Heidelberg, Germany

[Barthe and Rezk, 2005] Barthe, G. and Rezk, T.: 2005, Non-interference for a JVM-like language, in *Proc. of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation (TLDI '05)*, pp 103–112, ACM, New York, NY, USA

[Bavera and Bonelli, 2008] Bavera, F. and Bonelli, E.: 2008, Type-based information flow analysis for bytecode languages with variable object field policies, in *Proceedings of the 2008 ACM symposium on Applied computing (SAC '08)*, pp 347–351, ACM, New York, NY, USA

[Beckert et al., 2007] Beckert, B., Hähnle, R., and Schmitt, P. H. (eds.): 2007, *Verification of Object-Oriented Software: The KeY Approach*, Vol. 4334 of

Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg, Germany

[Beringer and Hofmann, 2007] Beringer, L. and Hofmann, M.: 2007, Secure information flow and program logics, in *Proc. of 20th IEEE Computer Security Foundations Symposium (CSF '07)*, pp 233–248, IEEE Computer Society, Los Alamitos, CA USA

[Beringer et al., 2003] Beringer, L., Mackenzie, K., and Stark, I.: 2003, Grail: A functional form for imperative mobile code, *Electronic Notes in Theoretical Computer Science* **85(1)**, 21

[Besson et al., 2005] Besson, F., Jensen, T., and Pichardie, D.: 2005, *A PCC Architecture based on Certified Abstract Interpretation*, Technical Report RR-5751, INRIA Rennes

[Besson et al., 2007] Besson, F., Jensen, T., Pichardie, D., and Turpin, T.: 2007, *Result Certification for relational program analysis*, Technical Report 6333, INRIA

[Besson et al., 2006] Besson, F., Jensen, T. P., and Pichardie, D.: 2006, Proof-carrying code from certified abstract interpretation and fixpoint compression, *Theoretical Computer Science* **364(3)**, 273

[Bishop, 2004] Bishop, M.: 2004, *Introduction to Computer Security*, Addison-Wesley Professional

[Burdy and Pavlova, 2006] Burdy, L. and Pavlova, M.: 2006, Java bytecode specification and verification, in *Symposium on Applied Computing Proceedings of the 2006 ACM symposium on Applied computing*, pp 1835 – 1839, ACM, New York, NY, USA

[Burdy et al., 2005] Burdy, L., Y.Cheon, Cok, D., Ernst, M., Kiniry, J., Leavens, G., Rustan, K., Leino, M., and Poll, E.: 2005, An overview of JML tools and applications, *International Journal on Software Tools for Technology Transfer* **7(3)**, 212

[Chalin et al., 2008] Chalin, P., James, P., and Karabotsos, G.: 2008, JML4: Towards an industrial grade IVE for Java and next generation research platform for JML, in *Verified Software: Theories, Tools, Experiments*, Vol.

5295 of *Lecture Notes in Computer Science*, pp 70–83, Springer Berlin / Heidelberg, Germany

[Chalin et al., 2007] Chalin, P., James, P. R., and Karabotsos, G.: 2007, An integrated verification environment for JML: architecture and early results, in *Proceedings of the 2007 conference on Specification and verification of component-based systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, pp 47–53, ACM, New York, NY, USA

[Chalin et al., 2005] Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E.: 2005, Beyond assertions: Advanced specification and verification with JML and ESC/Java2, in *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005*, Vol. 4111 of *Lecture Notes in Computer Science*, pp 342–363, Springer-Verlag, Berlin Heidelberg, Germany

[Chander et al., 2005a] Chander, A., Espinosa, D., Islam, N., Lee, P., and Necula, G.: 2005a, Enforcing resource bounds via static verification of dynamic checks, in *European Symposium on Programming (ESOP2005)*, Vol. 3444 of *Lecture Notes in Computer Science*, pp 311–325, Springer-Verlag, Berlin Heidelberg, Germany

[Chander et al., 2005b] Chander, A., Espinosa, D., Islam, N., Lee, P., and Necula, G.: 2005b, Jver: A Java verifier, in *Procs. Conference on Computer Aided Verification (CAV 2005)*, Vol. 3576 of *Lecture Notes in Computer Science*, pp 144–147, Springer-Verlag, Berlin Heidelberg, Germany

[Chander et al., 2007] Chander, A., Espinosa, D., Islam, N., Lee, P., and Necula, G.: 2007, Enforcing resource bounds via static verification of dynamic checks, *ACM Trans. Program. Lang. Syst.* 29

[Chang et al., 2006] Chang, B. E., Chlipala, A., and Necula, G. C.: 2006, A framework for certified program analysis and its applications to mobile-code safety, in *Verification, Model Checking, and Abstract Interpretation, 7th International Conference (VMCAI 2006)*, Vol. 3855 of *Lecture Notes in Computer Science*, pp 174–189, Springer-Verlag, , Berlin Heidelberg, Germany

[Chang et al., 2005] Chang, B. E., Chlipala, A., Necula, G. C., and Schneck, R. R.: 2005, The open verifier framework for foundational verifiers, in *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation (TLDI '05)*, pp 1–12, ACM, New York, USA

[Chen et al., 2006] Chen, F., Hills, M., and Roşu, G.: 2006, *A Rewrite Logic Approach to Semantics Definition,Design and Analysis of Object-Oriented Languages*, Technical Report UIUCDCS-R-2006-2702, University of Illinois Urbana-Champaign Department of Computer Science

[Chen et al., 2007] Chen, Y., Ge, L., Hua, B., Li, Z., Liu, C., and Wang, Z.: 2007, A pointer logic and certifying compiler, *Frontiers of Computer Science in China* **1**, 297

[Chlipala, 2007] Chlipala, A.: 2007, *Implementing Certified Programming Language tools in Dependent type theory*, *Ph.D. thesis*, University of California, Berkeley

[Chong and Myers, 2005] Chong, S. and Myers, A.: 2005, Language-based information erasure, in *Proc. of the 18th Computer Security Foundations Workshop (CSFW'05)*, pp 1–14, IEEE Computer Society, Los Alamitos, CA USA

[Chong and Myers, 2008] Chong, S. and Myers, A.: 2008, End-to-end enforcement of erasure and declassification, in *Proc. 21st Computer Security Foundations Symposium (CSF'08)*, pp 98–111, IEEE Computer Society, Los Alamitos, CA USA

[Clarkson and Schneider, 2008] Clarkson, M. R. and Schneider, F. B.: 2008, Hyperproperties, in *Proc. IEEE 21st Computer Security Foundations Symp. (CSF'08)*, pp 51 – 65, IEEE Computer Society, Los Alamitos, CA USA

[Clavel et al., 2002] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. F.: 2002, Maude: specification and programming in rewriting logic, *Theoretical Computer Science* **285(2)**, 187

[Clavel et al., 2003] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C.: 2003, The Maude 2.0 system, in R.

Nieuwenhuis (ed.), *Rewriting Techniques and Applications (RTA 2003)*, No. 2706 in Lecture Notes in Computer Science, pp 76–87, Springer Berlin / Heidelberg, Germany

[Clavel et al., 2005] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C.: 2005, *Maude Manual (version 2.2)*, SRI International

[Clavel et al., 2007] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C.: 2007, *All About Maude: A High-Performance Logical Framework*, Vol. 4350 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin Heidelberg, Germany

[Colby et al., 2000] Colby, C., Lee, P., Necula, G. C., Blau, F., Plesko, M., and Cline, K.: 2000, A certifying compiler for Java, in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, Vancouver, British Columbia, Canada*, pp 95–107, ACM, New York, NY, USA

[Corbett et al., 2000a] Corbett, J., Dwyer, M., Hatcliff, J., and Robby: 2000a, A language framework for expressing checkable properties of dynamic software, in *SPIN Model Checking and Software Verification*, Vol. 1885 of *Lecture Notes in Computer Science*, pp 205–223, Springer Berlin / Heidelberg, Germany

[Corbett et al., 2000b] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zheng, H.: 2000b, Bandera: Extracting finite-state models from Java source code, in *Proceedings of the 22nd International Conference on Software Engineering*, ICSE'2000, p. 439, IEEE Computer Society, Los Alamitos, CA, USA

[Cousot, 2004] Cousot, P.: 2004, Verification by abstract interpretation, in *International Symposium on Verification - Theory & Practice - Honoring Zohar Manna's 64th Birthday*, Vol. 2772 of *Lecture Notes in Computer Science*, pp 243–268, Springer-Verlag, Berlin Heidelberg, Germany

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R.: 1977, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *Conference Record of the Fourth*

ACM SYmposium on Principles of Programming Languages, pp 238–252, ACM, New York, NY, USA

[Cousot and Cousot, 1979] Cousot, P. and Cousot, R.: 1979, Systematic Design of Program Analysis Frameworks, in *Proc. of Sixth ACM Symp. on Principles of Programming Languages*, pp 269–282, ACM, New York, NY, USA

[Cousot and Cousot, 2002] Cousot, P. and Cousot, R.: 2002, On abstraction in software verification, in *Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002, Copenhagen, Denmark*, Vol. 2404 of *Lecture Notes in Computer Science*, pp 37–56, Springer-Verlag Berlin Heidelberg, Germany

[Cousot and Halbwachs, 1978] Cousot, P. and Halbwachs, N.: 1978, Automatic discovery of linear restraints among variables of a program, in *Proc. of 5Th. ACM Symp. on Principles of Programming Languages (POPL'78)*, pp 84–97, ACM, New York, NY USA

[Darvas et al., 2005] Darvas, A., Hahnle, R., and Sands, D.: 2005, A theorem proving approach to analysis of secure information flow, in *second international conference on Security in Pervasive Computing(SPC2005)*, Vol. 3450 of *Lecture Notes in Computer Science*, pp 193–209, Springer-Verlag, Berlin / Heidelberg, Germany

[Denney and Fischer, 2005] Denney, E. and Fischer, B.: 2005, Certifiable program generation, in *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE '05)*, Vol. 3676 of *Lecture Notes in Computer Science*, pp 17–28, Springer-Verlag, Berlin, Germany

[Denney and Fischer, 2006a] Denney, E. and Fischer, B.: 2006a, Annotation inference for the safety certification of automatically generated code, in *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE '06)*, pp 265–268, IEEE Computer Society, Los Alamitos, CA USA

[Denney and Fischer, 2006b] Denney, E. and Fischer, B.: 2006b, A generic annotation inference algorithm for the safety certification of automatically

generated code, in *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE'06)*, pp 121–130, ACM, New York, NY, USA

[Denney and Trac, 2008] Denney, E. and Trac, S.: 2008, A software safety certification tool for automatically generated guidance, navigation and control code, in *IEEE Aerospace Conference. Big Sky, MT*, IEEE Computer Society, Los Alamitos, CA USA

[Denning and Denning, 1977] Denning, D. E. and Denning, P. J.: 1977, Certification of programs for secure information flow, *Comm. ACM* **20(7)**, 504

[Detlefs et al., 2003] Detlefs, D., Nelson, G., and Saxe, J.: 2003, *Simplify: A Theorem prover for Program Checking*, Technical Report HPL-2003-148, Hewlet-Packard Systems Research

[Dowek et al., 2001] Dowek, G., Hardin, T., and Kirchner, C.: 2001, HOL-$\lambda\sigma$: An intentional first-order expression of higher-order logic, *Mathematical Structures in Computer Science* **11(1)**, 21

[Dufay et al., 2005] Dufay, G., Felty, A., and Matwin, S.: 2005, Privacy-sensitive information flow with JML, in *CADE20 Conferen. on Automated Deduction*, Vol. 3622 of *Lecture Notes in Computer Science*, pp 116–130, Springer-Verlag, Berlin Heidelberg, Germany

[Dwyer et al., 2001] Dwyer, M. B., Hatcliff, J., Joehanes, R., Laubach, S., Păsăreanu, C. S., Zheng, H., and Visser, W.: 2001, Tool-supported program abstraction for finite-state verification, in *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pp 177–187, IEEE Computer Society, Washington, DC, USA

[eclipse, 2011] eclipse: 2011, *Eclipse Java IDE*, Eclipse web page: `http://www.eclipse.org/downloads/moreinfo/java.php`

[Farzan et al., 2004a] Farzan, A., Chen, F., Meseguer, J., and Rosu, G.: 2004a, Formal analysis of Java programs in JavaFAN, in *Computer Aided Verification*, Vol. 3114 of *Lecture Notes in Computer Science*, pp 501–505, Springer-Verlag, Berlin Heidelberg, Germany

[Farzan et al., 2007] Farzan, A., Chen, F., Meseguer, J., and Rosu, G.: 2007, *JavaRL: The Rewriting Logic Semantics of Java*, Available at http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java

[Farzan et al., 2004b] Farzan, A., Meseguer, J., and Rosu, G.: 2004b, Formal JVM code analysis in JavaFAN, in *Proc. AMAST'04*, Vol. 3116 of *Lecture Notes in Computer Science*, pp 132–147, Springer-Verlag, Berlin Heidelberg, Germany

[Felty, 2005] Felty, A. P.: 2005, A tutorial example of the semantic approach to foundational proof-carrying code., in *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, Vol. 3467 of *Lecture Notes in Computer Science*, pp 394–406, Springer Berlin / Heidelberg, Germany

[Filliâtre and Marché, 2007] Filliâtre, J. and Marché, C.: 2007, The Why/Krakatoa/Caduceus platform for deductive program verification, in *19th International Conference on Computer Aided Verification*, Vol. 4590 of *Lecture Notes in Computer Science*, pp 173–177, Springer, Berlin, Germany

[Flanagan et al., 2002] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R.: 2002, Extended static checking for Java, in *Proceedings of the ACM SIG- PLAN 2002 Conference on Programming language design and implementation, PLDI 2002*, pp 234–245, ACM, New York, NY USA

[Focardi et al., 1994] Focardi, R., Gorrieri, R., Focardi, R., and Gorrieri, R.: 1994, A classification of security properties for process algebras, *Journal of Computer Security* **3**, 5

[Francesco and Martini, 2007] Francesco, N. D. and Martini, L.: 2007, Instruction-level security typing by abstract interpretation, *International Journal of Information Security* **6(2-3)**, 85

[Franz et al., 2003] Franz, M., Chandra, D., Gal, A., Haldar, V., Reig, F., and Wang, N.: 2003, A portable virtual machine target for Proof-Carrying Code, in *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, IVME '03, pp 24–31, ACM, New York, NY, USA

[Giacobazzi and Mastroeni, 2004] Giacobazzi, R. and Mastroeni, I.: 2004, Abstract non-interference: Parameterizing non-interference by abstract interpretation, in *Proc. of the 31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL'04*, pp 186–197, ACM, New York, NY, USA

[Gilmore and Prowse, 2005] Gilmore, S. and Prowse, M.: 2005, Proof carrying bytecode, *Electronic Notes in Theoretical Computer Science* **141(1)**, 3

[Goguen and Meseguer, 1982] Goguen, J. A. and Meseguer, J.: 1982, Security policies and security models, in *IEEE Symposium on Research in Security and Privacy*, pp 11–20, IEEE Computer Society, Los Alamitos, CA USA

[Grandy et al., 2008] Grandy, H., Bischof, M., Stenzel, K., Schellhorn, G., and Reif, W.: 2008, Verification of Mondex Electronic Purses with KIV: From a security protocol to verified code, in *FM 2008: Formal Methods*, Vol. 5014 of *Lecture Notes in Computer Science*, pp 165–180, Springer Berlin / Heidelberg, Germany

[Grossman and Morrisett, 2001] Grossman, D. and Morrisett, G.: 2001, Scalable certification for typed assembly language, in *Third International Workshop on Types in Compilation, TIC 2000, Montreal, Canada, September 21, 2000, Selected Papers*, Vol. 2071 of *Lecture Notes in Computer Science*, pp 117–146, Springer-Verlag, Berlin Heidelberg, Germany

[Hamid, 2005] Hamid, N.: 2005, *A Syntactic Approach to Foundational Proof-Carrying Code*, *Ph.D. thesis*, Yale University

[Hamid et al., 2002] Hamid, N., Shao, Z., Trifonov, V., Monnier, S., and Ni, Z.: 2002, A syntactic approach to foundational proof-carrying code, in *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pp 89–100, IEEE Computer Society, Los Alamitos,CA USA

[Hamid et al., 2003] Hamid, N., Shao, Z., Trifonov, V., Monnier, S., and Ni, Z.: 2003, A syntactic approach to foundational proof-carrying code, *Journal of Automated Reasoning* **31(3-4)**, 191

[Hamid and Shao, 2004] Hamid, N. A. and Shao, Z.: 2004, Interfacing Hoare logic and type systems for foundational proof-carrying code, in *Proc. 17th*

International Conference on the Applications of Higher Order Logic Theorem Proving (TPHOLs'04), Vol. 3223 of *Lecture Notes in Computer Science*, pp 118–135, Springer-Verlag, Berlin Heidelberg, Germany

[Hansen and Probst, 2006] Hansen, R. and Probst, C.: 2006, Non-interference and erasure policies for Java card bytecode, in *Proc. 6th International Workshop on Issues in the Theory of Security, WITS'06*

[Harren and Necula, 2005] Harren, M. and Necula, G. C.: 2005, Using dependent types to certify the safety of assembly code, in *Static Analysis Symposium (SAS 2005)*, No. 3672 in Lecture Notes in Computer Science, pp 155–170, Springer-Verlag, Berlin Heidelberg, Germany

[Harren, 2007] Harren, M. T.: 2007, *Dependent Types for Assembly Code Safety*, *Ph.D. thesis*, University of California, Berkeley

[Henzinger et al., 2002] Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G., Sutre, G., and Weimer, W.: 2002, Temporal-safety proofs for systems code, in *Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002*, Vol. 2404 of *Lecture Notes in Computer Science*, pp 526–538, Springer-Verlag, Berlin , Germany

[Hofmann and Jost, 2003] Hofmann, M. and Jost, S.: 2003, Static prediction of heap space usage for first order functional programs (extended version), in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'03*, pp 185–197, ACM, New York, NY, USA

[Hunt and Sands, 2006] Hunt, S. and Sands, D.: 2006, On flow-sensitive security types, in *Conf. record of the 33rd symposium on Principles of programming languages (POPL'06)*, pp 79–90, ACM, New York, NY USA

[Hunt and Sands, 2008] Hunt, S. and Sands, D.: 2008, Just forget it, the semantics and enforcement of information erasure, in *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems ESOP'08/ETAPS'08*, Vol. 4960 of *Lecture Notes in Computer Science*, pp 239–253, Springer-Verlag, Berlin Heidelberg, Germany

[Jacobs et al., 2005] Jacobs, B., Pieters, W., and Warnier, M.: 2005, Statically checking confidentiality via dynamic labels, in *Proceedings of the 2005 workshop on Issues in the theory of security (WITS '05)*, pp 50–56, ACM, New York, NY, USA

[Janota et al., 2007] Janota, M., Grigore, R., and Moskal, M.: 2007, Reachability analysis for annotated code, in *Proceedings of the 2007 conference on Specification and verification of component-based systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, pp 23–30, ACM, New York, NY, USA

[Jim et al., 2002] Jim, T., Morriset, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y.: 2002, Cyclone: A safe dialect of C, in *USENIX Annual Technical Conference, Monterey, CA, USA*, pp 275–288

[Klein and Nipkow, 2006] Klein, G. and Nipkow, T.: 2006, A machine-checked model for a Java-like language, virtual machine, and compiler, *ACM Trans. Program. Lang. Syst.* **28**, 619

[L. Jiang and Pan, 2007] L. Jiang, L. P. and Pan, X.: 2007, Handling information release and erasure in multi-threaded programs, in *Proc. IEEE Int. Conf. on Computational Intelligence and Security*, pp 824–828, IEEE Computer Society, Los Alamitos, CA USA

[Leavens et al., 2006] Leavens, G., Baker, A., and Ruby, C.: 2006, Preliminary design of JML: A behavioral interface specification language for Java, *ACM SIGSOFT Software Engineering Notes* **31**, 1

[Leavens et al., 2008] Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., and Zimmerman, D. M.: 2008, *JML Reference Manual (DRAFT)*, Available at : `http://www.eecs.ucf.edu/~leavens/JML/`

[Leroy, 2009] Leroy, X.: 2009, Formal verification of a realistic compiler, *Communications of the ACM* **52(7)**, 107

[Li et al., 2010] Li, Z., Zhuang, Z., Chen, Y., Yang, S., Zhang, Z., and Fan, D.: 2010, A certifying compiler for Clike subset of C language, pp 47–56, IEEE Computer Society, Los Alamitos, CA, USA

[MacKenzie and Wolverson, 2004] MacKenzie, K. and Wolverson, N.: 2004, Camelot and Grail: Resource-aware functional programming for the JVM, in *TFP 2004 Fifth Symposium on Trends in Functional Programming, Ludwig-Maximilians University, Munich, Germany*, Vol. 4, pp 29–46, Intellect

[Manna and Pnueli, 1995] Manna, Z. and Pnueli, A.: 1995, *Temporal Verification of Reactive Systems: safety*, Springer-Verlag, New York, NY, USA

[Marché et al., 2004] Marché, C., Paulin-Mohring, C., and Urbain, X.: 2004, The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML, *Journal of Logic and Algebraic Programming* **58(1–2)**, 89

[Matos and Boudol, 2005] Matos, A. and Boudol, G.: 2005, On declassification and the non–disclosure policy, in *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pp 226–240, IEEE Computer Society, Los Alamitos, CA USA

[Meseguer, 1992] Meseguer, J.: 1992, Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science* **96(1)**, 73

[Meseguer et al., 2003] Meseguer, J., Palomino, M., and Martí-Oliet, N.: 2003, Equational abstractions, in F. Baader (ed.), *Automated Deduction – CADE-19*, Vol. 2741 of *Lecture Notes in Computer Science*, pp 2–16, Springer Berlin / Heidelberg, Germany

[Meseguer and Roşu, 2007] Meseguer, J. and Roşu, G.: 2007, The rewriting logic semantics project, *Theoretical Computer Science* **373(3)**, 213

[Meyer and Poetzsch-Heffter, 2000] Meyer, J. and Poetzsch-Heffter, A.: 2000, An architecture for interactive program provers, in *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, TACAS '00, pp 63–77, Springer-Verlag, London, UK

[Mok and Yu, 2002a] Mok, A. and Yu, W.: 2002a, TINMAN: A resource bound security checking system for mobile code, in *Computer Security - ESORICS 2002: Proceedings 7th European Symposium on Research in*

Computer Security, Zurich, Switzerland, Vol. 2502 of *Lecture Notes in Computer Science*, pp 178–193, Springer, Berlin / Heidelberg, Germany

[Mok and Yu, 2002b] Mok, A. K. and Yu, W.: 2002b, Enforcing resource bound safety for mobile SNMP agents, in *Proceedings of 18th Annual Computer Security Applications Conference (ACSAC)*, pp 69–77, IEEE Computer Society, Los Alamitos, CA, USA

[Morrisett et al., 1999a] Morrisett, G., K.Crary, N.Glew, D.Grossman, R.Samuels, F.Smith, D.Walker, S.Weirich, and S.Zdancewic: 1999a, Talx86: A realistic typed assembly language, in *ACM SIGPLAN Workshop on Compiler Support for System Software,Atlanta, GA, USA*, pp 25–35

[Morrisett et al., 1999b] Morrisett, G., Walker, D., Crary, K., and Glew, N.: 1999b, From system F to typed assembly language, *ACM Trans. Program. Lang. Syst.* **21(3)**, 527

[Myers, 1999] Myers, A.: 1999, JFlow: Practical mostly-static information flow control, in *Proc. of the 26th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages POPL 1999*, pp 228–241, ACM, New York, NY USA

[Myers et al., 2001] Myers, A., Nystrom, N., Zheng, L., and Zdancewic, S.: 2001, *Jif: Java information flow*, Software release. Available at:http://www.cs.cornell.edu/jif

[Necula, 1997] Necula, G. C.: 1997, Proof carrying code, in *Proceedings of the 24th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages POPL 1997, Paris, France*, pp 106–119, ACM, New York, NY, USA

[Necula, 2001a] Necula, G. C.: 2001a, A scalable architecture for proof-carrying code, in *Functional and Logic Programming: Proceedings: 5th International Symposium, FLOPS 2001, Tokyo, Japan*, Vol. 2024 of *Lecture Notes in Computer Science*, pp 21–39, Springer-Verlag, Berlin / Heidelberg, Germany

[Necula, 2001b] Necula, G. C.: 2001b, *TouchStone, a certifying compiler for Java*, TouchStone Web Page. http://raw.cs.berkeley.edu/touchstone.html

[Necula and Lee, 1996] Necula, G. C. and Lee, P.: 1996, Safe kernel extensions without run time checking, in *Proceedings of the second USENIX symposium on Operating systems design and implementation OSDI 1996, Seattle, Washington, United States*, Vol. 30, pp 229–243, ACM, New York, NY, USA

[Necula and Lee, 1997a] Necula, G. C. and Lee, P.: 1997a, *Efficient Representation and Validation of Logical Proofs*, Technical Report CMU-CS-97-172, Carnegie Mellon University

[Necula and Lee, 1997b] Necula, G. C. and Lee, P.: 1997b, Research on proof-carrying code for untrusted-code security, in *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, p. 204, IEEE Computer Society, Los Alamitos, CA, USA

[Necula and Lee, 1998a] Necula, G. C. and Lee, P.: 1998a, The design and implementation of a certifying compiler, in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, PLDI 1998, Montreal, Quebec, Canada*, Vol. 33, pp 333 – 344, ACM Press, New York, NY, USA

[Necula and Lee, 1998b] Necula, G. C. and Lee, P.: 1998b, Safe untrusted agents using proof carrying code, in *Special Issue on Mobile Agent Security*, Vol. 1419 of *Lecture Notes in Computer Science*, pp 61 – 91, Springer-Verlag, Berlin Heidelberg, Germany

[Necula and Lee, 2004] Necula, G. C. and Lee, P.: 2004, The design and implementation of a certifying compiler, *SIGPLAN Not.* **39**, 612

[Necula and Rahul, 2001] Necula, G. C. and Rahul, S.: 2001, Oracle-based checking of untrusted software, in *Proceedings of the 28th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages, 2001 , London, United Kingdom*, pp 142 – 154, ACM, New York, NY, USA

[Necula and Schneck, 2002] Necula, G. C. and Schneck, R. R.: 2002, A gradual approach to a more trustworthy, yet scalable, proof-carrying code, in *Proceedings of the 18th International Conference on Automated Deduction (CADE'02), Copenhagen,*, Vol. 2392, pp 47 – 62, Springer-Verlag, Berlin Heidelberg, Germany

[Necula and Schneck, 2003a] Necula, G. C. and Schneck, R. R.: 2003a, Proof-carrying code with untrusted proof rules, in *Software Security - Theories and Systems Mext-NSF-JSPS Revised Papers of International Software Security Symposium, ISSS 2002 Tokyo, Japan, 2002*, Vol. 2609 of *Lecture Notes in Computer Science*, pp 283–298, Springer, Berlin / Heidelberg, Germany

[Necula and Schneck, 2003b] Necula, G. C. and Schneck, R. R.: 2003b, A sound framework for untrusted verification-condition generators, in *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS 2003)*, pp 248–260, IEEE Computer Society, Los Alamitos, CA, USA

[Ni and Shao, 2006] Ni, Z. and Shao, Z.: 2006, Certified assembly programming with embedded code pointers, in *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'96)*, Vol. 41, pp 320–333, ACM New York, NY, USA

[Owre, 2007] Owre, S.: 2007, *PVS Specification and Verification System, version 4.0*, Technical report, Computer Science Laboratory, SRI International, Available at http://pvs.csl.sri.com

[Pirzadeh and Dubé, 2008a] Pirzadeh, H. and Dubé, D.: 2008a, Encoding the program correctness proofs as programs in PCC technology, *Privacy, Security and Trust, Annual Conference on* **0**, 121

[Pirzadeh and Dubé, 2008b] Pirzadeh, H. and Dubé, D.: 2008b, VEP: a virtual machine for extended proof-carrying code, in *Proceedings of the 1st ACM workshop on Virtual machine security*, VMSec '08, pp 9–18, ACM, New York, NY, USA

[Poetzsch-Heffter and Müller, 2011] Poetzsch-Heffter, A. and Müller, P.: 2011, Jive web page: http://softech.informatik.uni-kl.de/Homepage/Jive

[Robby et al., 2006] Robby, Rodríguez, E., Dwyer, M. B., and Hatcliff, J.: 2006, Checking JML specifications using an extensible software model checking framework, *Int. J. Softw. Tools Technol. Transf.* **8**, 280

[Rose, 2003] Rose, E.: 2003, Lightweight bytecode verification, *J. Autom. Reason.* **31(3-4)**, 303, Kluwer Academic Publishers

[Sabelfeld and Myers, 2003] Sabelfeld, A. and Myers, A.: 2003, Language-based information-flow security, *IEEE Journal on Selected Areas in Communications* **21(1)**, 5

[Sabelfeld and Sands, 2009] Sabelfeld, A. and Sands, D.: 2009, Declassification: Dimensions and principles, *Journal of Computer Security* **17(5)**, 517

[Sannella et al., 2005] Sannella, D., Hofmann, M., Aspinall, D., Gilmore, S., Stark, I., Beringer, L., Loidl, H.-W., K.Mackenzie, Momigliano, A., and Shkaravska, O.: 2005, Mobile resource guarantees (project evaluation paper), in *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, Vol. 6, pp 211–226, Intellect

[Schneck, 2004] Schneck, R. R.: 2004, *Extensible Untrusted Code Verification*, *Ph.D. thesis*, University of California, Berkeley

[Schumann, 2003] Schumann, J.: 2003, Automated theorem proving in generation, verification, and certification of safety critical code, in *Automated Reasoning with Analytic Tableaux and Related Methods*, Vol. 2796 of *Lecture Notes in Computer Science*, pp 3–3, Springer, Berlin / Heidelberg, Germany

[Sekar et al., 2001] Sekar, R., Ramakrishnan, C., Ramakrishnan, I., and Smolka, S.: 2001, Model-carrying code (MCC): A new paradigm for mobile-code security, in *Proceedings of the New Security Paradigms Workshop (NSPW 2001)*, pp 23–30, ACM, New York, NY, USA

[Sekar et al., 2003] Sekar, R., Venkatakrishnan, V., Basu, S., Bhatkar, S., and DuVarney, D.: 2003, Model-carrying code: A practical approach for safe execution of untrusted, in *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*, pp 15–28, ACM, New York, NY, USA

[Şerbănuţă et al., 2009] Şerbănuţă, T. F., Roşu, G., and Meseguer, J.: 2009, A rewriting logic approach to operational semantics, *Information and Computation* **207**, 305

[Shao, 1997] Shao, Z.: 1997, An overview of the FLINT/ML compiler, in *Proc. ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, pp 85–98, ACM, New York, NY, USA

[Shao and Appel, 1995] Shao, Z. and Appel, A.: 1995, A type based certifying compiler for standard ML, in *Proc. ACM SIGPLAN conference on Programming language design and implementation (PLID'95)*, pp 116–129, ACM, New York, NY, USA

[Shroff et al., 2007] Shroff, P., Smith, S., and Thober, M.: 2007, Dynamic dependency monitoring to secure information flow, in *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pp 203–217, IEEE Computer Society, Los Alamitos, CA USA

[Stenzel, 2005] Stenzel, K.: 2005, *Verification of Java Card Programs*, *Ph.D. thesis*, Universität Augsburg

[Tarditi et al., 1996] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P.: 1996, TIL: A type-directed optimizing compiler for ML, *SIGPLAN Not.* **31**, 181

[Tarditi et al., 2004] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P.: 2004, TIL: A type-directed optimizing compiler for ML (with retrospective), in *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, pp 554–567, ACM, New York, NY, USA

[TeReSe, 2003] TeReSe (ed.): 2003, *Term Rewriting Systems*, Cambridge University Press, Cambridge

[TILT, 2006] TILT: 2006, *The TILT Compiler Project*, Web Page. http://www.cs.cornell.edu/home/jgm/tilt.html

[Together, 2011] Together: 2011, *Together IDE*, Together web page: http://www.borland.com/us/products/together/

[Tsukada, 2000] Tsukada, Y.: 2000, Mobile codes with interactive proofs: an approach to provably safe evolution of distributed software systems, in *Proceedings International Symposium on Principles of Software Evolution,2000*, pp 23–27, IEEE Computer Society, Los Alamitos, CA USA

[Tsukada, 2005] Tsukada, Y.: 2005, Interactive and probabilistic proof of mobile code safety, *Automated Software Engineering* **12(2)**, 237

[van den Berg and Jacobs, 2001] van den Berg, J. and Jacobs, B.: 2001, The LOOP compiler for Java and JML, in T. Margaria and W. Yi (eds.), *Tools and Algorithms for the Construction and Analysis of Software (TACAS01)*, Vol. 2031 of *Lecture Notes in Computer Science*, pp 299–312, Springer-Verlag, Berlin Heidelberg, Germany

[Vargun, 2006] Vargun, A.: 2006, *Code-carrying theory*, *Ph.D. thesis*, Rensselaer Polytechnic Institute

[Volpano et al., 1996] Volpano, D., Smith, G., and Irvine, C.: 1996, A sound type system for secure flow analysis, *Computer Security* **4(3)**, 167

[Warnier, 2005] Warnier, M.: 2005, *Language Based Security for Java and JML*, *Ph.D. thesis*, Radboud University Nijmegen

[Wasserrab et al., 2009] Wasserrab, D., Lohner, D., and Snelting, G.: 2009, On PDG-based noninterference and its modular proof, in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS '09)*, pp 31–44, ACM, New York, NY, USA

[Whalen et al., 2002] Whalen, M., Schumann, J., and Fischer, B.: 2002, Synthesizing certified code, in *FME 2002:Formal Methods Getting IT Right*, Vol. 2391 of *Lecture Notes in Computer Science*, pp 149–155, Springer, Berlin / Heidelberg, Germany

[Whalen et al., 2003] Whalen, M., Schumann, J., Fischer, B., and Whittle, J.: 2003, Certification support for automatically generated programs, in *Proceedings of the 36th Hawaii International Conference on System Sciences - 2003*, p. 10, IEEE Computer Society, Los Alamitos, CA USA

[Whitehead et al., 2004] Whitehead, N., Abadi, M., and Necula, G.: 2004, By reason and authority: A system for authorization of proof-carrying code, in *Procs. 17th IEEE Computer Security Foundations WorkShop (CSF 2004)*, pp 236–250, IEEE Computer Society, Los Alamitos, CA USA

[Wildmoser et al., 2005] Wildmoser, M., Chaieb, A., and Nipkow, T.: 2005, Bytecode analysis for proof carrying code, in *Proceedings of the 1st Workshop on Bytecode Semantics, Verification and Transformation*, Vol. 141

of *Electronic Notes in Theoretical Computer Science*, pp 19–34, Elsevier, Amsterdam, Netherlands

[Wildmoser and Nipkow, 2005] Wildmoser, M. and Nipkow, T.: 2005, Asserting bytecode safety, in *Proceedings of the 15th European Symposium on Programming (ESOP05)*, Vol. 3444 of *Lecture Notes in Computer Science*, pp 326–341, Springer-Verlag, Berlin Heidelberg, Germany

[Wildmoser et al., 2004] Wildmoser, M., Nipkow, T., Klein, G., and Nanz, S.: 2004, Prototyping proof carrying code., in *3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France*, pp 333–348, Kluwer, Boston, MA USA

[Wu, 2005] Wu, D.: 2005, *Interfacing Compilers, proof checkers, and Proofs for foundational proof-carrying code*, *Ph.D. thesis*, Princeton University

[Wu et al., 2003] Wu, D., Appel, A., and Stump, A.: 2003, Foundational proof checkers with small witnesses, in *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming (PPDP'03)*, pp 264–274, ACM Press, New York, NY, USA

[Xia and Hook, 2003a] Xia, S. and Hook, J.: 2003a, *Abstraction-carrying Code: a New Method to Certify Temporal Properties*, informal proceedings available at http://www.cs.ru.nl/$\sim$erikpoll/ftfjp/2003/FTfJP2003.pdf, Presented at the Workshop Formal Techniques for Java-like Programs, co-located with the 17th European Conference on Object-Oriented Programming (ECOOP 2003)

[Xia and Hook, 2003b] Xia, S. and Hook, J.: 2003b, Experience with abstraction-carrying code, *Electronic Notes in Theoretical Computer Science* **89(3)**, 17, Workshop on Software Model Checking (SoftMC 2003)

[Xia and Hook, 2003c] Xia, S. and Hook, J.: 2003c, *An Implementation of Abstraction-carrying Code*, Foundations of Computer Security" (proceedings of the LICS'03 workshop on Foundations of Computer Security), Technical Report TR-2003-04, Department of Computer Science, University of Ottawa, Ottawa, Canada, 26-27 June 2003.

[Xia and Hook, 2004] Xia, S. and Hook, J.: 2004, Certifying temporal properties for compiled C programs, in *Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, Vol. 2937 of *Lecture Notes in Computer Science*, pp 161–174, Springer, Berlin / Heidelberg, Germany

[Yiyun et al., 2007] Yiyun, C., Ge, L., Baojian, H., Zhaopeng, L., and Liu, C.: 2007, Design of a certifying compiler supporting proof of program safety, in *Theoretical Aspects of Software Engineering, 2007. TASE '07. First Joint IEEE/IFIP Symposium on*, pp 127 –138

[Yu and Mok, 2004] Yu, W. and Mok, A.: 2004, Formal specification and verification of resource bound security using PVS, in *Software Security Theories and Systems*, Vol. 3233 of *Lecture Notes in Computer Science*, pp 113–133, Springer-Verlag, Berlin Heidelberg, Germany

[Zanardini, 2007] Zanardini, D.: 2007, Analysing non–interference with respect to classes, in *Proc. 10th Italian Conference on Theoretical Computer Science (ICTCS'07)*, pp 57–69, World Scientific Publishing

[Zanotti, 2002] Zanotti, M.: 2002, Security typings by abstract interpretation, in *Proc. Symposium on Static Analysis, SAC'02*, Vol. 2477 of *Lecture Notes in Computer Science*, pp 360–375, Springer-Verlag, Berlin , Germany

[Zheng and Myers, 2007] Zheng, L. and Myers, A. C.: 2007, Dynamic security labels and static information flow, *International Journal of Information Security* **6(2)**, 67

Appendix A

Related work: a comparison

Table A.1: Low-level imperative languages.

Language	Security policies	Assumptions	Formalism	Proof	Technology	References
Assembler	Memory Safety	Non-trusted: code Trusted: VCGen, Proof checker, Inference rules	Hoare logic High-order logic Type-checking	A LF expression (type derivation)	ELF, VCGen	[Necula and Lee, 1996; Necula and Lee, 1997b]
Assembler extension for ML	Memory and ML Types safety				ELF, VCGen, LF type-checker	[Necula, 1997]
Assembler compiled from C	Memory and C Types safety				Certificant compiler, LF type-checker	[Necula and Lee, 1998a; Necula and Lee, 1998b]
Assembler	Safe memory access with memory and locks bounds				ELF, VCGen, LF Type-checker	[Necula and Lee, 1997b]
Assembler	Safe memory and file access; CPU cycles and bandwitdh bounds				ELF, VCGen, LF Type-checker	[Necula and Lee, 1998b]
Assembler translated from byte-code	Memory and Type Safety for Java (Simple J)				Certificant compiler, LF type-checker	[Colby et al., 2000; Necula, 2001b]

Low-level imperative languages

Continued from previous page

Language	Security policies	Assumptions	Formalism	Proof	Technology	References
Assembler compiled from ML	Safe type handling	Non-trusted: Code Trusted: Compiler	Typed λ calculus Type system		Certifying compiler FLINT/ML	[Shao and Appel, 1995; Shao, 1997]
Assembler translated from Simple J	Memory and Type Safety for Java	Non-trusted: Code, Inference rules Trusted: Proof checker	Hoare and High-order logic, Typed λ calculus	Type inference	Certifying compiler (VCGen, certifier), Coq prover	[Necula and Schneck, 2002; Necula and Schneck, 2003a]
Typed assembler TAL translated by Popcorn, Cyclone and Scheme	Safe memory and type handling	Non-trusted: Code, Compiler Trusted: Proof checker	Hoare Logic Type-checking	Typing annotations of code labels	Certifying compiler (VCGen, certifier), type-checker	[Morrisett et al., 1999b; Morrisett et al., 1999a; Grossman and Morrisett, 2001; Jim et al., 2002]
DEC/Alpha assembler traslated from ML	Safe types and arrays of dimension 1 and 2	Non-trusted: Code Trusted: Compiler	Type-checking	Typing annotations of program variables	Certifying compiler TIL	[Tarditi et al., 1996]
Sparc and Pentium-like Assembler	Safe memory handling	Non-trusted: Code, proof Trusted: Checker, Semantics, Policy	High-order and Hoare logic, Type-checking, Unification	A type derivationin LF	TWELF (LF implementation). λ Prolog and FLIT.	[Appel and Felty, 2000; Appel, 2001; Appel and Mcallester, 2001; Felty, 2005]
Assembler traslated from JVM bytecode	Java Type safety	No-trusted: Code Trusted: Inference rules, Proof checker		Witness of a type derivation	Proof checker is a logic programming interpreter	[Necula and Rahul, 2001; Necula, 2001a]

Low-level imperative languages

Continued from previous page

Language	Security policies	Assumptions	Formalism	Proof	Technology	References
FTAL, extended TAL: XTAL, certified asse. lang. CAP	Memory and type safe handling	Non-trusted: Code, Proof Trusted: Proof checker, Machine semantics, Policy	Calculus of Inductive Constructions CiC Hoare logic	Type derivations	FTAL compiler Coq theorem prover	[Hamid et al., 2003; Hamid et al., 2002; Hamid and Shao, 2004; Hamid, 2005; Ni and Shao, 2006]
Assembler	General high or low level policies, specialized by the code consumer	Non-trusted: Code Trusted: Static analyser, Model checker	Extended finite automata EFSA tabular resolution	EFSA behavior model	Static Analyser, consistency solver(XMC) and an execution monitor	[Sekar et al., 2001; Sekar et al., 2003]
Assembly language and bytecode	Memory safety, valid instruction code	Non-trusted: Code, Custom VCGen Trusted: Checker, VCGen kernel, VCGen decoder	Logic with simple types			[Necula and Schneck, 2003b]
Java Bytecode	Resource bounds: CPU, memory, disk, bandwidth, threads, handles	Non-trusted: Code, Verifier Trusted: Proof checker	Hoare logic, Theorem proving	LF expression with a type derivation	JVer verifier, Theorem prover Kettle	[Chander et al., 2005b]
Low level general: an example with Alpha assembler	General: example with safe types and safe arrays	Non-trusted: Code Trusted: Interactive probabilistic validator, Prover	Hoare logic, Arithm. of boolean formulae, Number Theory	coefficients of a low-grade polynomial	Interactive probabilistic Prover and Validator	[Tsukada, 2000; Tsukada, 2005]

Low-level imperative languages

Continued from previous page						
Language	Security policies	Assumptions	Formalism	Proof	Technology	References
SAL assembler translated from Jinja (Java subset) bytecode	Generic. Example with type safety and safe arithmetic operations (no overflow)	Non-trusted: Code, Inference rules Trusted: VCGen	High-order logic, directed graphs	Typed λ term	Theorem Prover Isabelle/ HOL, VCGen	[Wildmoser et al., 2004; Wildmoser et al., 2005; Wildmoser and Nipkow, 2005]
Bytecode	Array safe handling, integer variables in ranges	Non-trusted: Code, Fix point Trusted: Coq type-checker, Caml compiler Certified: Proof checker	Abstract Interpretation, Fix point semantics, Type-checking	Typed λ term	Coq theorem prover, Caml program extractor and Compiler	[Besson et al., 2005; Besson et al., 2006]
TALx86 assembler, Bytecode traslated by Cool	General; examples with type safety and memory safety	Non-trusted: Code, Verifier extensions Trusted: Strongest post-cond. gen., Proof checker, Fix point finder	Abstract Interpretation, Type-checking, Horn Logic CiC (an extension of CC)		Open Verifier: Post, Proof checker, Fix Point Extensions (verifiers): PCCExt, TALExt, CoolAid	[Chang et al., 2005]

Low-level imperative languages

Continued from previous page

Language	Security policies	Assumptions	Formalism	Proof	Technology	References
x86 assembler produced by TALx86, Bytecode	Safe types and memory	Non-trusted: Code, Analyzer Certified: Certifier Trusted: Spec. extractor, Ocaml comp., Coq type-checker	Abstract Interpretation, Theorem proving		Analyser Certifier Specification extractor Ocaml compiler Coq type-checker	[Chang et al., 2006]
x86 assembler translated from an CCured or Cqual annotated-C	Safe pointers and memory, General type safety policies	Non-trusted: Code, C compiler, Security tool (CCured, Cqual) Trusted: Verifier	Dependent type system, Type inference	Code annotations	CCured and Cqual tools, Type-checker prototype	[Harren and Necula, 2005; Harren, 2007]
Imperative Bytecode with arrays, procedures and global variables	Resource consumption, Memory safety, Safe arrays	Non-trusted: Code, Proof checker Trusted: Coq type-checker	Hoare logic, Relational abstraction, Linear constraint systems	A polyhedra inclusion	Relational bytecode analyser, Result checker, Coq type-checker	[Besson et al., 2005; Besson et al., 2006; Besson et al., 2007]
Bytecode with classes, objects, arrays and exceptions	Non–interference (privacy) based on a lattice of data confidentiality levels	Non-trusted: Code Certified: Proof checker Trusted: Coq type-checker	Type system, Information flow	Code annotations, Prover tactics	Analysers (PA, CDR, IF) Checkers (PA, CDR, IF) Coq prover	[Barthe et al., 2007b]

Low-level imperative languages

Continued from previous page						
Language	Security policies	Assumptions	Formalism	Proof	Technology	References
Assembler x86 traslated from PointerC	Dynamic memory and type safety, integer safety	Non-trusted: Code, Compiler Trusted: Coq proof checker	Type-checking, Hoare logic, separation logic, theorem proving	Code assertions (annotations)	Certifying compiler, Coq proof checker	[Yiyun et al., 2007; Chen et al., 2007]
Assembler translated from C	memory safety, control-flow safety, type safety (including numeric ranges) and resource bounds safety (code size, stack size, heap size and timeout)	Non-trusted: C compiler, Proof generator, Code Trusted: VEP virtual machine, proof checker and the VCGen	Theorem proving	An LF term	C compiler, Proof generator, VEP, VCGen, proof checker	[Pirzadeh and Dubé, 2008b; Pirzadeh and Dubé, 2008a]
Assembler x86 translated from Clike	Dynamic memory safety, type safety	Non-trusted: Code, compiler Trusted: Coq proof checker	Hoare logic, separation logic, theorem proving	Code annotations (assertions)	Certifying compiler, Coq proof checker	[Li et al., 2010]

Table A.2: Imperative C and C++ Languages.

Language	Security policies	Assumptions	Formalism	Proof	Technology	References
C	Correctness	Non-trusted: Code Trusted: Model checker, theorem prover BLAST	Abstract interpretation on the fly, CFA graphs	A LF expression	Model checker, BLAST. The validator is not included, but it can be used a PCC checker	[Henzinger et al., 2002]
C	Resource consumption: execution time, heap memory explicitly demanded and bandwidth	Non-trusted: Code Trusted: On-line and off-line checkers, traffic analyser	Extended Hoare logic, Abstract Interpretation, Tactic Theorem proving	Proof tactics for PVS prover and their parameters	Tactic Theorem prover PVS, Static Analyser, Dynamic Analyser	[Mok and Yu, 2002a; Mok and Yu, 2002b; Yu and Mok, 2004]
C	Temporal properties: liveness	Non-trusted: Code Trusted: Type checker Abstract model checker	Indexed type system, Abstract interpretation, Control flow automata, Model checking, LTL without next operator	Boolean program code with indexed types	C compiler to SDTAL language, ACCEPT/C, Indexed type checker, Abstract model checker	[Xia and Hook, 2003a; Xia and Hook, 2003c; Xia and Hook, 2003b; Xia and Hook, 2004]
Generated C and C++	Operator safety regarding partial functions, bounds array safety and variable initialization safety	Non-trusted: Annotation generator Trusted: VCGen, Theorem prover	First-order logic Hoare logic		AutoBayes generator, E-SETHEO, Theorem Prover	[Whalen et al., 2003; Whalen et al., 2002; Schumann, 2003; Whalen et al., 2003]

Imperative languages C and object-oriented C++

Continued from previous page						
Language	Security policies	Assumptions	Formalism	Proof	Technology	References
Generated C	Initialization safety and no out-of-bounds array accesses.	Untrusted: annotation generator Theorem prover Trusted: Domain theory VCGen and proof checker	First-order logic Hoare logic	The formulae derivations that are encoded as acyclic directed graphs	Annotation inference: Prolog, AutoBayes generator, First-order logic theorem Provers: E-SETHEO	[Denney and Fischer, 2005; Denney and Fischer, 2006b; Denney and Fischer, 2006a]
Generated C++	Functional correctness of generic types Type safety and termination			Generic proofs as functions	Athena theorem prover Functional programming	[Vargun, 2006]

Table A.3: Imperative object-oriented language Java.

Language	Security Policies	Assumptions	Formalism	Proof	Technology	References
Annotated Java	Resource consumption limits: CPU, memory, disk, bandwidth, files, database connections, threads	Non-trusted: Code, Prover Trusted: Theorem prover validator	Hoare logic, Weakest precondition calculus Dijkstra, linear inequations, Theorem proving, Satisfiability	A satisfiability proof	ESC/Java, Theorem Prover Simplify	[Chander et al., 2005a; Chander et al., 2007]

Table A.4: Functional Languages.

Language	Security Policies	Assumptions	Formalism	Proof	Technology	References
Low-level functional Grail; Grail compiled from functional high-level Camelot	Function Heap consumption: linearly bounded relatively to input size	Non-trusted: Code Trusted: Bytecode Functionalyser, Isabelle Theorem prover	Hoare and High-order logic, Extended type system LFD	LFD typing annotation	Certifying compiler Camelot, Grail compiler, bytecode functionalizer, Isabelle prover	[Hofmann and Jost, 2003; Beringer et al., 2003; MacKenzie and Wolverson, 2004; Aspinall et al., 2004; Gilmore and Prowse, 2005]
ML language	The even/odd property , ML types safety	Non-trusted: Code, Code proof Trusted: Proof checker, Machine Semantics, Policy	High-order and Hoare logic, Type-checking, Unification	A LF expression (represents a type derivation)	Type preserving ML compiler, the LTAL language, TWELF and Flit proof checker	[Wu, 2005; Wu et al., 2003]

Table A.5: Logic Languages.

Language	Security Policies	Assumptions	Formalism	Proof	Technology	References
Ciao (CLP)	Any policy specified in CLP as variable or procedure properties, as determinacy, termination, and non-failure: example with secure file access	Non-trusted: Code, Certificate (proof) Trusted: Validator (analysis result's checker)	Abstract Interpretation, Fix point semantics, CLP	The result of the abstract interpretation of the program	Ciao preprocessor CiaoPP Ciao analisis tools for determinism, termination, and no failure	[Albert et al., 2005a]
	Bounded memory for data and functions, function cost bounded, and functions without side-effects				Ciao preprocessor CiaoPP	[Albert et al., 2004; Albert et al., 2006]
Maude	Functional correctness	Non-trusted: Generated Code Trusted: Program transformation steps	Fold/ Unfold Program Transformation Rewriting Logic	Sequence of transformation steps	Maude narrowing extension	[Alpuente et al., 2010a; Alpuente et al., 2010b]

Appendix B

Code of Chapter 5 example programs

In this appendix, we briefly describe the program code of the thesis benchmarks.

Example 24. This example has the same required even result and has no parameters as Example 2, but it uses pre-increment, pre-decrement, post-decrement, post-increment, and several assignment operators (*=,+=,-=).

```
static int even16star()    {
   /*@    requires true;
     @    ensures AbsValue(\result)  == #even ;         @*/
     int x = 4;
     int y = x + 8;
     y++;
     x--;
     --y;
     ++x;
     x*=1;
     y+=3 - x;
     y-=3 - x;
     x+=11 + y;
     x-=11 + y;
     x = x + y - y;
     y = y + x - x;
 return x+y;
}
```

Appendix C

Code of Chapter 6 example programs

Example 25. The method `mExample1` is a Java version of Example 1 borrowed from [Warnier, 2005; Jacobs et al., 2005]:

```
class Safe1NonInterference2p1 {
        static  int h = 5, l = 10;
        //@ setLabel(h, High);
        static  Testclass  t = new Testclass(h,l);

public static void main(String[] args)  {
          t.mExample1();
          System.out.println(t.getHigh());
          System.out.println(t.getLow());
        }
}
class Testclass  {
  //
  int high; int low;
  //@ setLabel(high, High);
  public Testclass(int hp, int lp)  {
     high= hp;
     low= lp;
  }
  public int getHigh() { return high;}
  public int getLow() { return low;}
  public void mExample1() {
     low = high;
     low = 2;
  }
}
```

Example 26. The method `mEx2` is the Java method code version of a simple non–interferent example borrowed from [Hunt and Sands, 2006]:

```
class Safe1NonInterference39 {
        static int h = 10, l= 0;
        //@ setLabel(h, High);
        static Testclass t = new Testclass(h,l);
        public static void main(String[] args) {
          System.out.println(t.getHigh());
          t.mEx2();
```

```
          System.out.println(t.getLow());
        }
}
class Testclass {
        int high;
        //@ setLabel(high, High);
        int low;

        public Testclass(int hp, int lp)  {
          high= hp;
          low= lp;
        }
        public int getHigh() { return high;}
        public int getLow() { return low;}
        public void mEx2()         {
          low = high;
          low = 0;
          high = 0;
          low = high;
        }
}
```

Example 27. This example includes three simple methods in two classes: the non–interferent method included in the program of Example 25, an interferent method borrowed from [Warnier, 2005; Jacobs et al., 2005], and another non–interferent method borrowed from [Sabelfeld and Sands, 2009]:

```
class Safe1Noninterference44 {
        static  int h = 10, l = 0;
        //@ setLabel(h, High);
        static  Testclass  t = new Testclass(h,l);
        public static void main(String[] args)  {
          System.out.println(t.getHigh());
          System.out.println(t.getLow());
          t.mExamples1a2a3(h,l);
          System.out.println(t.getHigh());
          System.out.println(t.getLow());
        }
}
class Testclass {
        int high;
        //@ setLabel(high, High);
        int low;
        public Testclass(int hp, int lp)  {
          high = hp;
          low = lp;
        }
        public int getHigh() { return high;}
        public int getLow() { return low;}
        public void sethighlow(int hp, int lp) {
          high = hp;
          low = lp;
        }
```

```
        public void mExample1() {
          if (high > high)
             low = 0;
        }
        public void mExample2() {
          low = high;
          low = 2;
        }
        public void mExample3() {
          if (high > 2)
              low = 0;
        }
        public void mExamples1a2a3(int hp, int lp) {
          sethighlow(hp,lp);
          mExample3();
          mExample1();
          mExample2();
        }
}
```

Example 28. This benchmark includes six simple methods, the three methods included in the program of Example 27 and three other interferent methods also borrowed from [Warnier, 2005; Jacobs et al., 2005], including a method with a `while` loop and a method that calls another method:

```
class Safe1Noninterference45 {
        static  int h = 10, l = 0;
        //@ setLabel(h, High);
        static  Testclass  t = new Testclass(h,l);
        public static void main(String[] args)  {
          System.out.println(t.getHigh());
          System.out.println(t.getLow());
          t.mExample30a32a34(h,l);                // I
          t.mExamples1a2a3(h,l);                  // NI
          System.out.println(t.getHigh());
          System.out.println(t.getLow());
        }
}
class Testclass {
        int high;
        //@ setLabel(high, High);
        int low;
        public Testclass(int hp, int lp)  {
          high = hp;
          low = lp;
        }
        public int getHigh() { return high;}
        public int getLow() { return low;}
        public void sethighlow(int hp, int lp) {
          high = hp;
          low = lp;
        }
        public void mExample1() {
          if (high > high)
```

```
            low = 0;
        }
        public void mExample2() {
          low = high;
          low = 2;
        }
        public void mExample3() {
          if (high > 2)
              low = 0;
        }
        public void mExample32() {
          while (high > 0) {
            high--;
            low++;
          }
        }
        int decrementing(int i) {
            high = high - 1;
            return i ;
        }
        public void mExample34() {
          low = decrementing(high);
        }
        public void mExample30() {
           if (high > 2)
              low = 0;
          else
          low = 1;
        }
        public void mExamples1a2a3(int hp, int lp) {
          sethighlow(hp,lp);
          mExample3();
          mExample1();
          mExample2();
        }
        public void mExample30a32a34(int hp, int lp) {
          sethighlow(hp,lp);
          mExample30();
          sethighlow(hp,lp);
          mExample32();
          mExample34();
        }
}
```

Example 29. This program includes nine simple methods, the six examples included in the program of Example 29 plus three other interferent methods: two interferent variations of the loop of Example 13 and an interferent method with a return statement occurring within a conditional statement.

```
class Safe1Noninterference43 {
        static  int h = 10, l = 0;
        //@ setLabel(h, High);
        static  Testclass  t = new Testclass(h,l);
```

```
    public static void main(String[] args) {
          System.out.println(t.getHigh());
          System.out.println(t.getLow());
          t.mExample30a32a34(h,l);            // I
          t.mExample35a36a37(h,l);            // I
          t.mExamples1a2a3(h,l);               // NI
          System.out.println(t.getHigh());
          System.out.println(t.getLow());
        }
}
class Testclass {
        int high;
        //@ setLabel(high, High);
        int low;
        public Testclass(int hp, int lp)  {
          high = hp;
          low = lp;
        }
        public int getHigh() { return high;}
        public int getLow() { return low;}

        public void sethighlow(int hp, int lp) {
          high = hp;
          low = lp;
        }
    public void mExample1() {
          if (high > high)
          low = 0;
        }
        public void mExample2() {
          low = high;
          low = 2;
        }
        public void mExample3() {
       if (high > 2)
       low = 0;
        }
        public void mExample32() {
          while (high > 0) {
            high--;
            low++;
          }
        }
        int decrementing(int i) {
       high = high - 1;
           return i ;
        }
        public void mExample34() {
          low = decrementing(high);
        }
    public void mExample30() {
      if (high > 2)
         low = 0;
          else
         low = 1;
        }
```

```
        public void mExample33call() {
          low = mExample33();
        }
        public int mExample33() {
      if (high > 2)
          return 0;
          else
          return 1;
        }
        public void  mExample35() {
          while (high > 0) {
            high--;
            low++;
            if (low > high)
               break;
          }
        }
        public void mExample36()  {
          while (true) {
            high--;
            low++;
            if (high == 0)
               break;
          }
        }
        public void mExamples1a2a3(int hp, int lp) {
          sethighlow(hp,lp);
          mExample3();
          mExample1();
          mExample2();
        }
        public void mExample30a32a34(int hp, int lp) {
          sethighlow(hp,lp);
          mExample30();
          sethighlow(hp,lp);
          mExample32();
          mExample34();
        }
        public void mExample35a36a37(int hp, int lp) {
          sethighlow(hp,lp);
          mExample33();
          sethighlow(hp,lp);
          mExample35();
          sethighlow(hp,lp);
          mExample36();
        }
}
```

Appendix D

Code of Chapter 7 example programs

Example 30. This program is adapted from [Hunt and Sands, 2008] and does not comply with the required erasure policy:

```
class Safe1Erasure3p2 {
        static  Testclass t = new Testclass(3, -3, 6);
    public static void main(String[] args)  {
          //System.out.println(t.getX());
          //System.out.println(t.getY());
          //System.out.println(t.getZ());
          t.mE3();
          //System.out.println(t.getX());
          //System.out.println(t.getY());
          //System.out.println(t.getZ());
        }

}
class Testclass {
        int xh;   //@ setLabel(xh, High);
        int yl;
        int zl;
        public Testclass(int xp, int xy, int zp)  {
          //@ setLabel(xp, High);
          xh = xp;
          yl = xy;
          zl = zp;
         }
        public int getX() { return xh;}
        public int getY() { return yl;}
        public int getZ() { return zl;}
        public void mE3() {
          // erasure specification:
          /*@ up(zl, Low, Top);  @*/
          xh = xh + yl + zl;
          if (zl == 0)
              yl = yl + 1;
      zl = 0;
        }
}
```

Example 31. This program is a version of Example 19 that does not erase the required variables either:

```
class MedWebSite1p2 {
        static  MedicalDiagnosis t = new MedicalDiagnosis();
        public static void main(String[] args)  {
          t.mainLoop();
          //System.out.println(t.getmalaise());
          //System.out.println(t.getfever());
          //System.out.println(t.getinfluenza());
        }
}
class MedicalDiagnosis {
  // Symptons:
  boolean malaise;
  boolean fever;
  /* ... */
  // Diagnosis:
  boolean influenza;
    //@ setLabel(malaise, High);
    //@ setLabel(fever, High);
    //@ setLabel(influenza,High);

  /* ... */
  boolean userReqExit;
  public boolean getmalaise() { return malaise;}
  public boolean getfever() { return fever;}
  public boolean getinfluenza() { return influenza;}

  public MedicalDiagnosis() {
    //@ setLabel(f, High);
    boolean f = false;
    malaise = f;
    fever = f;
    /* ... */
    influenza = f;
    /* ...*/
    userReqExit = false;
  }
  public void getSymptons() {
    /* ...*/
    boolean tr = true;
    //@ setLabel(tr, High);
    malaise = tr;
    fever = tr;
   }
  public void getUserReq() {
    /* ... */
    userReqExit = true;
  }
  public void appEnd() {
    // erasure specification:
    /*@ erase(malaise, High, Top);
      @ erase(fever, High, Top);
      @ erase(influenza, High, Top); @*/
    /* ... */
  }
  public void exit() {
    /* ... */
```

```
  }
  public void diagnosis ()  {
    if (malaise && fever )
        influenza = true;
    /*
    else  if ....
    */
  }
  public void mainLoop() {
    while (! userReqExit) {
      getSymptons();
      diagnosis();
      getUserReq();
    };
    if (userReqExit ) {
      appEnd();
      exit();
    };
  }
}
```

Example 32. The code is adapted from [Hunt and Sands, 2008] and it does comply with its erasure policy.

```
class Safe1Erasure2p1 {
        static  Testclass t = new Testclass(3, -3, 6);
        public static void main(String[] args)  {
          //System.out.println(t.getX());
          //System.out.println(t.getY());
          //System.out.println(t.getZ());
          t.mE2();
          //System.out.println(t.getX());
          //System.out.println(t.getY());
          //System.out.println(t.getZ());
        }

}
class Testclass {
        int xh;   //@ setLabel(xh, High);
        int yl;
        int zl;

        public Testclass(int xp, int xy, int zp)  {
          //@ setLabel(xp, High);
          xh = xp;
          yl = xy;
          zl = zp;
         }
        public int getX() { return xh;}
        public int getY() { return yl;}
        public int getZ() { return zl;}
        public void mE2() {
          // erasure specification:
          /*@ erase(zl, Low, High);  @*/
          xh = xh + yl + zl;
```

```
          yl = yl + zl;
          zl = 0;
        }
}
```