Program-level Specification and Deductive Verification of Security Properties

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Christoph Scheben

aus Siegen

Tag der mündlichen Prüfung:	26. November 2014
Erster Gutachter:	Prof. Dr. Peter H. Schmitt Karlsruher Institut für Technologie (KIT)
Zweiter Gutachter:	Prof. Dr. Markus Müller-Olm Westfälische Wilhelms-Universität Münster

Acknowledgments

I would not have succeeded in writing this thesis without the support of many great people.

I would like to take this opportunity to express my special thanks and appreciation to Prof. Dr. Peter H. Schmitt for his—in every way—excellent guidance and support. Thanks for the advice, discussions, patience, care and trust, thanks for giving me a lot of time for research and special thanks for postponing your retirement until completion of my thesis!

I would also like to thank my second reviewer Prof. Dr. Markus Müller-Olm for investing all the time and energy in fulfilling this role.

I am grateful to my colleague Dr. Mattias Ulbrich who shared the office with me and who always was open for—sometimes more, sometimes less scientific but always exciting and inspiring—discussions, supporting me wherever he could in his kind and likeable way. I am also thankful to all my other former and current colleagues—too many to be listed all by name—for the teamwork, the nice working atmosphere and all the inspiring discussions. It was always fun to work with you. Likewise, I would like to thank all my student assistants for their tireless engagement and the nice and fruitful teamwork.

My sincere thanks go to my parents and family for their ongoing and unconditional support and love. Special thanks go also to all my friends, foremost Markus, for all the years of support in all situations of life.

Lastly, I would like to thank the German National Science Foundation (DFG) for financially supporting this thesis as part of the project "Program-level Specification and Deductive Verification of Security Properties" within priority programme 1496 "Reliably Secure Software Systems – RS3".

Abstract

Programs with publicly accessible interfaces (like web applications) are increasingly used to process confidential data. This makes it all the more important to control the information flow within such applications: confidential information must not leak to publicly accessible outputs. Already small programming errors may lead to critical information leaks, as the "Heartbleed bug" recently showed. Language-based information flow analysis, which is considered here, tries to prevent such bugs by program analysis and verification techniques.

Though a variety of sophisticated information flow analysis techniques and tools have been developed in the past, it was not feasible to verify complex information flow properties of open, object-oriented programs with expressive declassification (controlled release) of information. An instance of such kind of programs are electronic voting systems. In those systems, secrecy of votes is an important property which could not be proven on the code level up to recently.

This thesis shows how highly precise specification and deductive verification of language-based secure information flow can be made feasible. The approach does not rely on fixed approximations, but makes use of the precision provided by the underlying (relatively complete) calculus for Java Dynamic Logic.

On the specification side, the thesis presents an extension of the Java Modeling Language (JML), the defacto standard language for behavioral specification of Java code, for fine-grained, knowledge-based, modular information flow specifications including declassification. These specifications can be translated to a novel formalization of language-based secure information flow in Java Dynamic Logic and subsequently be analyzed with the deductive software verification system KeY, as shown in a further contribution. The formalization is formulated in self-composition style, a semantically precise, but costly formulation of language-based secure information flow. This thesis shows how the efficiency of self-composition style reasoning can be improved considerably. These improvements as well as a contribution on modular self-composition verification are indispensable prerequisites for the scalability of the self-composition approach.

Though the optimized self-composition approach is feasible, as proven by a case study on a simplified electronic voting system, it is still a heavyweight

approach. Many programs, or at least parts of them, can be verified with less precise but more efficient techniques. To this end, a more efficient, approximate information flow calculus is presented. Compared to similar approaches from literature, it has the advantage that the logic which underlies the calculus is chosen such that it is possible to switch on-the-fly from the approximate reasoning to self-composition style reasoning if higher precision is needed at some point during the proof. Furthermore, it is shown that an integration of approximate and precise techniques can also be achieved on the specification level. To this end, a semantically correct translation of the most important specification elements of a security type-checking approach, the prominent Java Information Flow approach, into the JML extension is presented. Because both techniques are modular on the method level, this allows to verify each method with the technique most appropriate for it.

All techniques except for the last two have been implemented in the KeY system and successfully tested on examples from literature and, in cooperation with the research group of Prof. Ralf Küsters from the University of Trier, on an implementation of a simple electronic voting system.

Deutsche Zusammenfassung (Abstract in German)

Programme mit öffentlich zugänglichen Interfaces (wie z. B. Webapplikationen) werden zunehmend zur Verarbeitung von vertraulichen Daten genutzt. Dies macht es umso wichtiger, den Fluss von Informationen in solchen Applikationen zu kontrollieren: vertrauliche Informationen dürfen nicht über öffentlich zugängliche Ausgaben preis gegeben werden. Schon kleine Programmierfehler können zu kritischen Informationspreisgaben führen, wie der "Heartbleed bug" kürzlich gezeigt hat. Sprach-basierte Informationsflussanalyse, welche hier betrachtet wird, versucht solche Fehler mit Hilfe von Programmanalyse- und Verifikationstechniken zu verhindern.

Obwohl in der Vergangenheit einige ausgefeilte Informationsflussanalysetechniken und Tools entwickelt worden sind, war es nicht möglich komplexe Informationsflusseigenschaften von offenen, objektorientierten Programmen mit ausdrucksstarker Deklassifikation (kontrollierter Freigabe) von Informationen zu verifizieren. Ein Beispiel für diese Art von Programmen sind elektronische Abstimmungssysteme. In solchen Systemen ist das Wahlgeheimnis eine wichtige Eigenschaft, welche auf Programmtextebene bis vor kurzem nicht bewiesen werden konnte.

Diese Dissertation zeigt wie die hochpräzise Spezifikation und deduktive Verifikation von sprach-basiertem sicherem Informationsfluss realisiert werden kann. Der Ansatz verwendet keine fixen Approximationen, sondern nutzt die Präzision des zugrunde liegenden (relativ vollständigen) Kalküls für Dynamische Logik für Java.

Auf der Spezifikationsseite präsentiert diese Dissertation eine Erweiterung der Java Modeling Language (JML), der defacto Standardsprache für die Verhaltensspezifikation von Java Code, um feinkörnige, wissensbasierte und modulare Informationsflussspezifikationen einschließlich Deklassifikation. Diese Spezifikationen können in eine neuartige Formalisierung von sprach-basiertem sicherem Informationsfluss in Dynamischer Logik für Java übersetzt und anschließend mittels des deduktiven Softwareverifikationssystems KeY analysiert werden, welches in einem weiteren Beitrag gezeigt wird. Die Formalisierung ist im Selbstkompositionsstil formuliert, einer semantisch präzisen, aber teuren

Formulierung von sprach-basiertem sicherem Informationsfluss. Diese Dissertation zeigt wie die Effizienz des Schlussfolgerns im Selbstkompositionsstil erheblich verbessert werden kann. Diese Verbesserungen sowie ein Beitrag zu modularer Selbstkompositionsverifikation sind unabdingbar für die Skalierbarkeit des Selbstkompositionsansatzes.

Obwohl der optimierte Selbstkompositionsansatz praktikabel ist, was durch eine Fallstudie zu einem einfachen elektronische Abstimmungssystem belegt wird, ist es dennoch ein schwergewichtiger Ansatz. Viele Programme, oder zumindest Teile von Ihnen, können mit weniger präzisen aber dafür effizienteren Techniken verifiziert werden. Zu diesem Zweck wird ein effizienterer, approximater Informationsflusskalkül präsentiert. Verglichen mit ähnlichen Ansätzen aus der Literatur hat der Kalkül den Vorteil, dass die Logik, die dem Kalkül zugrunde liegt, so gewählt ist, dass es möglich ist spontan vom approximaten Schlussfolgern zum Selbstkompositionsstil zu wechseln, falls höhere Präzision an irgendeinem Punkt im Beweis benötigt wird. Des weiteren wird gezeigt, dass eine Integration von approximaten und präzisen Techniken auf Spezifikationsebene erreicht werden kann. Hierzu wird eine semantisch korrekte Übersetzung der wichtigsten Spezifikationselemente eines Sicherheitstypprüfungsansatzes, dem prominenten Java Information Flow Ansatz, in die JML Erweiterung präsentiert. Da beide Techniken modular auf Methodenebene arbeiten, erlaubt dies die Verifikation jeder Methode mit der Technik, die am geeignetsten für sie ist.

All Techniken außer den letzten beiden sind im KeY-System implementiert und erfolgreich an Beispielen aus der Literatur und, in Kooperation mit der Forschergruppe von Prof. Ralf Küsters von der Universität Trier, an einer Implementierung eines einfachen elektronischen Abstimmungssystems getestet worden.

Contents

List of Figures

List of Listings

1 Introduction

Programs with publicly accessible interfaces (like web applications) are increasingly used to process confidential data. This makes it all the more important to control the information flow within such applications: confidential information must not leak to publicly accessible outputs. Already small programming errors may lead to critical information leaks, as the "Heartbleed bug" recently showed. To cope with the problems arising from these developments, in the last decades a discipline called *information flow control* emerged (Lampson [1973], Denning [1976], Cohen [1977], Goguen and Meseguer [1982]). Information flow control prevents information leaks by a variety of techniques, ranging from dynamic monitoring approaches to static program analysis and verification techniques. This thesis contributes to static information flow control. More precisely, it presents advances in the field of *language-based information flow analysis*.

Language-based information flow analysis is concerned with the analysis of program code, in contrast to other fields of information flow control which aim at the analysis of abstract systems, modeled for instance by finite state machines. Code level verification provides far reaching guarantees, but is—in particular for real world programming languages—inherently difficult. In general, the question whether a program has secure information flow or not is undecidable for Turing complete languages. Therefore, a fully automatic, sound and precise information flow verification technique for Turing complete programming languages cannot exist.

In the past, a variety of sophisticated information flow analysis techniques and tools have been developed. As in functional verification, the proposed techniques can be divided into lightweight (that is, automatic, but approximate) and heavyweight (that is, semiautomatic, but precise) approaches.

Popular lightweight approaches are security type systems (a prominent example in this field is the Java Information Flow (JIF) system by Myers [1999a]), the analysis of program dependence graphs for graph-theoretical reachability properties (Hammer et al. [2006]), specialized approximate information flow calculi based on Hoare like logics (Amtoft et al. [2006]) and the usage of abstraction and ghost code for explicit tracking of dependencies (Pan [2005], Bubel

et al. [2008], van Delft [2011]). A popular heavyweight approach is to state information flow properties by self-composition (Barthe et al. [2004], Darvas et al. [2005]) and use off-the-shelf software verification systems to check for them. An alternative is to formalize information flow properties in higher-order logic and use higher-order theorem provers for the verification of those properties, as presented for instance by Nanevski et al. [2011].

Lightweight approaches are usually efficient and scale well on large programs, but do not have the necessary precision to express and verify complex information flow properties of programs with controlled release of information. An instance of such kind of programs are electronic voting systems. In those systems, secrecy of votes is an important property which could not be proven by approximate approaches so far. Heavyweight approaches on the other hand were, until recently, applicable to artificially small examples only.

This thesis contributes to the deductive verification of complex information flow properties of open, object-oriented programs with controlled release of information. To this end, the thesis proposes several improvements of the self-composition approach, making a highly precise analysis of sequential Java programs feasible. The feasibility of the approach is proven by a case study on a simplified electronic voting system, carried out in cooperation with the research group of Prof. Ralf Küsters from the University of Trier. Though the optimized self-composition approach is practicable, it is still a heavyweight approach. In order to lighten the burden of the verification process, the present thesis proposes a combination of self-composition style reasoning with a novel approximate information flow calculus and—orthogonal to this approach—a combination with the JIF approach.

The presented techniques are used for the verification of sequential Java programs. The considered subset of Java explicitly covers exceptions, object creation and static initialization. It mainly does not include concurrency, floating point arithmetic and generics. In the context of this thesis sequential Java is considered to be deterministic.

The verification of information flow properties of object-oriented programs and the fine-grained specification of information flow properties pose special problems, as the next two sections show.

1.1 Information Flow in Object-Oriented Programs.

Object-oriented programs pose particular challenges for the verification of secure information flow. If variables of object type are observable, then the classical notion of secure information flow—secret inputs may not influence public

outputs—classifies almost all programs as insecure: the problem traces back to the creation of new objects and the fact that the mere existence of secret objects may influence the choice of a new observable object. In object-oriented programming languages like Java, where references are opaque (that is, references can be compared by the identity comparison operation == only), a lot of programs rejected by the classical notion of secure information flow are indeed secure. Therefore, it is reasonable to replace the classical notion by a less restrictive version which uses an adopted notion of indistinguishability of program states, as used for instance by Hedin and Sands [2005], Hansen and Probst [2006], Barthe et al. [2013], Banerjee and Naumann [2002], Beringer and Hofmann [2007], Naumann [2006]. We call the adopted notion *object-sensitive secure information flow*.

Lightweight information flow analyses usually check for object-sensitive secure information flow, but it has been an open research question to find a feasible precise formalization of this notion. The present thesis addresses this issue and derives a feasible precise formalization from an investigation into the concept of object-sensitive secure information flow itself.

1.2 Fine-Grained Specification of Information Flow Properties.

The verification of information flow properties relies on a formal specification of those properties. In language-based information flow analysis these specifications are usually given in form of annotations to the program code.

A well-known approach in this area is the JIF approach by Myers [1999a]. Though the approach belongs to the most successful information flow verification techniques for Java, the specifications are not suitable for highly precise information flow verification as targeted in this thesis. Banerjee et al. [2008] propose a promising scheme for the specification (and verification) of expressive declassification policies. The approach is based on a self-defined, non-object-oriented programming language, but the authors present some ideas how the technique could be extended to object-oriented languages as well. An alternative is the defacto standard language for behavioral specification of Java code: the Java Modeling Language (JML), introduced by Leavens et al. [2006, 2008]. The language was designed for the specification of functional properties, but there exist several approaches for the specification of information flow properties in JML as well. Warnier [2006] and Haack et al. [2008] encode sufficient conditions for information flow properties into pre- and postconditions of JML method contracts. This approach, however, is not suitable for highly precise

information flow verification. An alternative is the extension of JML by new keywords. Dufay et al. [2005] introduce new JML-keywords which directly define relations between the program variables of two self-composed executions. This is flexible because general relational properties can be expressed in this way. On the other hand, it seems to be questionable whether these specifications can be verified with other techniques than self-composition or relational verification in general.

This thesis builds upon the above ideas by proposing an extension of JML for the specification of complex information flow properties including expressive declassification which is suitable for highly precise information flow verification.

1.3 Contributions.

Overall, this thesis shows how highly precise specification and deductive verification of language-based secure information flow can be made feasible. The approach does not rely on fixed approximations, but makes use of the precision provided by the underlying (relatively complete) calculus for Java Dynamic Logic.

The main contributions of this thesis are:

- An extension of the Java Modeling Language (JML), the defacto standard language for behavioral specification of Java code, for fine-grained, knowledge-based, modular information flow specifications including expressive declassification (Chapter 4).
- A translation of the JML extensions to a novel formalization of language-based secure information flow in Java Dynamic Logic, formulated in the semantically precise self-composition style (Section 5.1).
- A considerable improvement of the efficiency of self-composition style reasoning (Section 5.2) and a technique for modular self-composition verification (Section 5.3). Both contributions are indispensable prerequisites for the scalability of the self-composition approach.
- An optimized formalization of object-sensitive secure information flow in Java Dynamic Logic (Chapter 6), derived from an investigation into the concept of object-sensitive secure information flow itself.
- An approximate information flow calculus with the ability to switch on-the-fly from approximate reasoning to self-composition style reasoning if higher precision is needed at some point during the proof (Chapter 7).

- An integration of approximate and precise techniques on the specification level with the help of a semantically correct translation of the most important specification elements of JIF into the JML extension (Chapter 8). Because both techniques are modular on the method level, this allows verifying each method with the technique most appropriate for it.
- A case study on a simplified electronic voting system which proves the feasibility of the optimized self-composition approach (Chapter 9). The case study was carried out in cooperation with the research group of Prof. Ralf Küsters from the University of Trier.
- An implementation of all techniques described in Chapters 4, 5 and 6. The implementation extends the KeY system and is accessible via Java Web Start on the website `http://www.key-project.org/DeduSec/`.

1.4 Publications.

Parts of the work presented in this thesis have already been published on well established conferences and workshops.

Parts of Chapter 2 are based on Scheben and Schmitt [2012]; Chapter 3 is a revised and extended version of parts of Scheben and Schmitt [2012] and Beckert et al. [2014]; Chapter 4 is a revised and extended version of Scheben and Schmitt [2012]; Chapter 5 is a revised version of Scheben and Schmitt [2014] and parts of Scheben and Schmitt [2012]; Chapter 6 is a revised version of Beckert et al. [2014]. Chapter 9 is loosely related to Küsters et al. [2013] and Bormer et al. [2012]. Finally, Ahrendt et al. [2014] gives a brief summary of the work presented in this thesis. Chapters 7 and 8 have not been published before. They appear for the first time in print. The introduction (Chapter 1) and the conclusion (Chapter 10) cover parts of all mentioned publications.

Further conference and workshop publications by the author not directly related to this thesis are Kuntz et al. [2012], Geisler and Scheben [2007], Scheben [2006].

1.5 Outline.

The next chapter gives a short introduction into Java Dynamic Logic and the Java Modeling Language. Both are used intensively throughout the thesis. Based on a formal attacker model, Chapter 3 then explains and formalizes when

a program is considered secure with respect to its information flow. The formalization of secure information flow is followed by the main contributions of the thesis as listed in Section 1.3. The thesis concludes with a summary and an outlook on future work in Chapter 10.

2 Foundations

Parts of this chapter are based on Scheben and Schmitt [2012].

This chapter gives a short introduction to Java Dynamic Logic and the Java Modeling Language. An in-depth presentation of Java Dynamic Logic can be found in Weiß [2011]. For a more comprehensive introduction to JML see Leavens et al. [2006, 2008] and Weiß [2011]. It is assumed that the reader is familiar with typed first order logic, as presented for instance in Beckert et al. [2007b].

2.1 Java Dynamic Logic

Java Dynamic Logic (Java DL) is an adaption of first order Dynamic Logic (Harel et al. [2000]) to the Java programming language. It is an extension of typed first order logic, augmented by

- modal operators $[\alpha]$ (called *box modality*) and $\langle\alpha\rangle$ (called *diamond modality*) for sequential Java programs α, and
- updates $\{u\}$ (which can be understood as a kind of explicit substitutions).

The syntax is defined on the basis of a *signature* $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{V}, (\mathcal{T}, \sqsubseteq), \gamma, \mathrm{Prg})$ consisting of

- a set of *function symbols* $\mathcal{F}$,
- a set of *predicate symbols* $\mathcal{P}$,
- a set of *variable symbols* $\mathcal{V}$,
- a set of *types* $\mathcal{T}$ with a subtype relation $\sqsubseteq$ of the form depicted in Figure 2.1,
- a *static typing function* γ, and
- a sequential Java program Prg.[1]

[1] Confer Beckert et al. [2007b] or Weiß [2011] for a detailed description of which Java programs are allowed.

The set of function symbols is divided into a set of *program variables* PV and a set of "rigidly" interpreted function symbols RF. According to Weiß [2011], terms, formulas and updates are defined recursively by the following grammar:[2]

$$\begin{aligned}
\mathrm{Trm}_A ::= \quad & x \mid f(\mathrm{Trm}_{B_1'}, \ldots, \mathrm{Trm}_{B_n'}) \mid \{\mathrm{Upd}\}\mathrm{Trm}_A \\
\mathrm{Frm} ::= \quad & \mathit{true} \mid \mathit{false} \mid p(\mathrm{Trm}_{B_1'}, \ldots, \mathrm{Trm}_{B_n'}) \mid \\
& \neg\mathrm{Frm} \mid \mathrm{Frm} \wedge \mathrm{Frm} \mid \mathrm{Frm} \vee \mathrm{Frm} \mid \mathrm{Frm} \rightarrow \mathrm{Frm} \mid \mathrm{Frm} \leftrightarrow \mathrm{Frm} \mid \\
& \forall x; \mathrm{Frm} \mid \exists x; \mathrm{Frm} \mid [\alpha]\mathrm{Frm} \mid \langle\alpha\rangle\mathrm{Frm} \mid \{\mathrm{Upd}\}\mathrm{Frm} \\
\mathrm{Upd} ::= \quad & \mathrm{v} := \mathrm{Trm}_{A'} \mid \mathrm{Upd} \,||\, \mathrm{Upd} \mid \{\mathrm{Upd}\}\mathrm{Upd} \mid \mathrm{Upd}; \mathrm{Upd}
\end{aligned}$$

where x is a variable of type A, $f : B_1 \times \ldots \times B_n \rightarrow A$ is a function symbol (that is, a rigidly interpreted function symbol or a program variable), $p : B_1 \times \ldots \times B_n$ is a predicate symbol, $B_i' \sqsubseteq B_i$, α is a program fragment (a "subprogram") of Prg, v is a program variable of type A and $A' \sqsubseteq A$. Trm_A denotes the set of terms of (exact) type A, Frm denotes the set of formulas and Upd denotes the set of updates. The set of all terms is defined as $\mathrm{Trm} ::= \bigcup_A \mathrm{Trm}_A$. Note that logical variables must not occur in programs, and program variables must not be quantified.

Java DL formulas are interpreted in *Kripke structures* $\mathcal{D} = (D, I, \mathcal{S}, \delta, P)$, where

- D is a set of values, called *domain* or *universe*,
- I is a function, called *interpretation*, which assigns a meaning to the predicate symbols $\mathcal{P}$ and the rigid function symbols RF,
- $\mathcal{S}$ is a set of *states* which consists of all functions assigning values to the program variables PV,
- $\delta : D \rightarrow \mathcal{T}$ is a *dynamic typing function* and
- P is a set of *transition relations* $\rho_\alpha : \mathcal{S} \times \mathcal{S}$, one for each legal program fragment (or "subprogram") α of Prg, such that $(s_1, s_2) \in \rho_\alpha$ if and only if α started in s_1 terminates normally in s_2.

As usual in first order logic, free variables are interpreted by a variable assignment β. Up to $\{u\}t$, $\{u\}\phi$, $[\alpha]\phi$ and $\langle\alpha\rangle\phi$, formulas and terms are evaluated as in first order logic. The notation $t^{\mathcal{D},s,\beta}$ denotes the evaluation of term t in Kripke structure $\mathcal{D}$, state s and variable assignment β, whereas $\mathcal{D}, s, \beta \vDash \phi$ expresses that formula ϕ evaluates to true in $(\mathcal{D}, s, \beta)$.

If $\mathcal{D}$ and β can be deduced from the context, we sometimes abbreviate $t^{\mathcal{D},s,\beta}$ by t^s and $\mathcal{D}, s, \beta \vDash \phi$ by $s \vDash \phi$. A tuple $(\mathcal{D}, s)$ of a Kripke structure $\mathcal{D}$ and a state s

[2] The depicted grammar has been slightly simplified.

is called a *model* of ϕ if $\mathcal{D}, s, \beta \vDash \phi$ holds for all variable assignments β. The case that ϕ evaluates to false is denoted by $\mathcal{D}, s, \beta \nvDash \phi$.

The semantics of $\{u\}t$, $\{u\}\phi$, $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ is defined as follows:

- $\mathcal{D}, s, \beta \vDash [\alpha]\phi$ holds if and only if $\mathcal{D}, s_2, \beta \vDash \phi$ holds for all states s_2 such that $(s, s_2) \in \rho_\alpha$. In other words, $\mathcal{D}, s, \beta \vDash [\alpha]\phi$ holds if and only if $\mathcal{D}, s_2, \beta \vDash \phi$ is true for all states s_2 such that α started in s terminates normally in s_2.

  Note that this definition does not require α to terminate. If α terminates, then s_2 is uniquely determined: sequential Java is considered to be deterministic.

- $\mathcal{D}, s, \beta \vDash \langle\alpha\rangle\phi$ holds if and only if there exists a state s_2 such that $(s, s_2) \in \rho_\alpha$ and $\mathcal{D}, s_2, \beta \vDash \phi$. In other words, $\mathcal{D}, s, \beta \vDash \langle\alpha\rangle\phi$ holds if and only if there exists a state s_2 such that α started in s terminates normally in s_2 and $\mathcal{D}, s_2, \beta \vDash \phi$ is true.

  This means in particular that $\langle\alpha\rangle$ requires α to terminate in the uniquely determined state s_2. Note that $\langle\alpha\rangle$ is the dual of $[\alpha]$, that is, $\mathcal{D}, s, \beta \vDash [\alpha]\phi$ holds if and only if $\mathcal{D}, s, \beta \vDash \neg\langle\alpha\rangle\neg\phi$ holds.

- $(\{u\}t)^{\mathcal{D},s,\beta} = t^{\mathcal{D},s^u,\beta}$ with $s^u = val_{\mathcal{D},s,\beta}(u)(s)$ where

  - $val_{\mathcal{D},s',\beta}(x := t)(s)$ is the state s^t_x defined as

$$s^t_x(y) = \begin{cases} t^{\mathcal{D},s',\beta} & \text{if } y = x \\ s(y) & \text{else} \end{cases},$$

  - $val_{\mathcal{D},s',\beta}(u_1 \,||\, u_2)(s) = val_{\mathcal{D},s',\beta}(u_2)(s'')$ with $s'' = val_{\mathcal{D},s',\beta}(u_1)(s)$,

  - $val_{\mathcal{D},s',\beta}(\{u_1\}u_2)(s) = val_{\mathcal{D},s'',\beta}(u_2)(s)$ with $s'' = val_{\mathcal{D},s',\beta}(u_1)(s)$, and

  - $val_{\mathcal{D},s',\beta}(u_1; u_2)(s) = val_{\mathcal{D},s',\beta}(u_1||\{u_1\}u_2)(s)$.

- $\mathcal{D}, s, \beta \vDash \{u\}\phi$ holds if and only if $\mathcal{D}, s^u, \beta \vDash \phi$ holds where s^u is defined as before as $s^u = val_{\mathcal{D},s,\beta}(u)(s)$.

Updates are comparable to substitutions, but in contrast to substitutions they are part of the logic itself.

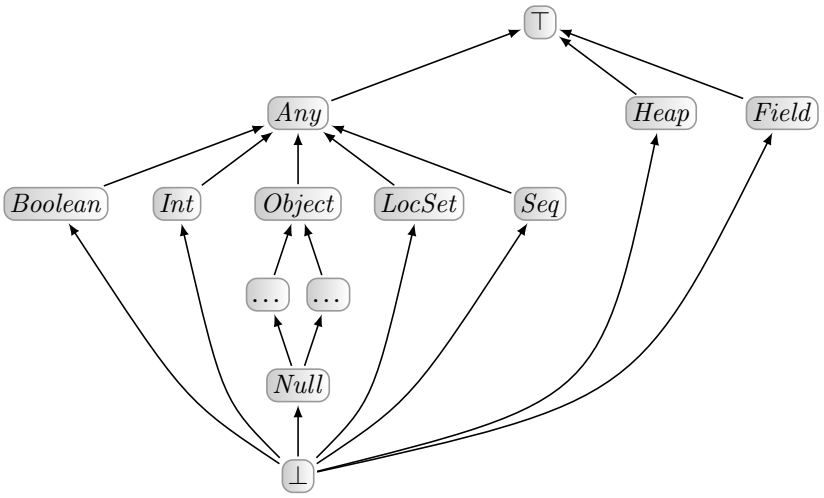


Figure 2.1: Java DL type hierarchy (from Ahrendt et al.)

2.1.1 Type Hierarchy

To check if the dynamic type of a term t is a subtype of A, Java DL contains special predicates $instance_A$, one for each type A, where $I(instance_A)(x) = true$ if and only if $\delta(x) \sqsubseteq A$ (for $x \in D$). Additionally, Java DL contains special predicates $exactInstance_A(t)$ for checking whether the dynamic type of a term t is of exact type A, that is, $I(exactInstance_A)(x) = true$ if and only if $\delta(x) = A$. For short we use $t \sqsubseteq A$ for $instance_A(t)$ and $eInst_A$ for $exactInstance_A$.

Any type hierarchy $(\mathcal{T}, \sqsubseteq)$ of Java DL has the form depicted in Figure 2.1. Beside the usual types $Boolean$, Int and $Object$ the Java DL type hierarchy contains the special types $Heap$, $Field$, $LocSet$ and Seq which require a short explanation.

2.1.2 Fields, Heaps and Object Creation

The type $Field$ represents the set of all Java fields. Java DL uses tuples (o, f) of objects o and fields f to address *heap locations*. The current heap of a Java program is represented by a special program variable $\texttt{heap} \in \text{PV}$ of type $Heap$. The elements of type $Heap$ represent heap states. A field accesses $o.f$ is formalized as in the theory of arrays (Mccarthy [1962], Bradley et al. [2006]) by $select_A(\texttt{heap}, o, f)$. Similarly, $store(\texttt{heap}, o, f, v)$ stores a value v in location (o, f) on the current heap. The creation of an object o is modeled by storing

the value `true` in the special location $(o, created)$. In Java DL objects may not be deallocated. Therefore, the value of $(o, created)$ cannot be modified by $store$ directly. Instead, Java DL uses the special constructor $create(h, o)$ to create an object o on heap h.

Java DL takes only few assumptions on the creation of new objects: if in Java a new object is created, it is assumed that (1) the newly created object is different from $null$; and (2) if the heap is wellformed (see below), then no already created object will be (re)created. The order in which objects are created is deterministic, but underspecified.

Within a program, field accesses always refer to the current heap state represented by `heap`. The same holds for assignments and object creations. The logic, however, may talk about different heap states. In Java DL it is even possible to quantify over all heap states. This is useful, for instance, to express noninterference in Java DL (see Chapter 5).

Not all heap states represented in the logic can actually be reached by running Java programs. The heap state where all (countably infinitely many) objects are created, for instance, exists in the logic, but is not reachable by any program. All heap states reachable by Java programs have the following properties: (1) all objects stored on the heap are either created or the `null` object; (2) only finitely many objects are created; and (3) all locations stored on the heap in location sets (see Section 2.1.3 below) refer to created objects or the `null` object only. If a heap state h fulfills these properties, then h is called wellformed. The set of wellformed heap states h is characterized by the predicate $wellFormed(h)$.

2.1.3 Location Sets

In the context of modular program verification it is useful to be able to talk about the set of heap locations modified or accessed by a method. In Java DL sets of heap locations are represented by the elements of type $LocSet$. Location sets are build up from the following constructors:

- $\dot{\emptyset} : LocSet$
- $singleton : Object \times Field \to LocSet$
- $\dot{\cup} : LocSet \times LocSet \to LocSet$
- $\dot{\cap} : LocSet \times LocSet \to LocSet$
- $\setminus : LocSet \times LocSet \to LocSet$

2.1.4 Sequences

Terms of type Seq represent finite sequences. A sequence may contain elements of different types. The term $\langle 5, \texttt{true} \rangle$, for instance, is an element of Seq. Sequences may also be nested.

Similar to location sets, sequences can be build up from the constructors

- $seqEmpty : Seq$,
- $seqSingleton : Any \to Seq$ and
- $seqConcat : Seq \times Seq \to Seq$.

Instead of

$$seqConcat(seqSingleton(v_1), seqConcat(\ldots, seqSingleton(v_n)))$$

we usually write $\langle v_1, \ldots, v_n \rangle$. Sequences can also be constructed with the help of the generalized quantifier $seq\{i\}(from, to, e)$ where e is a term of arbitrary type with free variable i and where $from$ and to are terms of type Int defining the range of the variable i. Its semantics is defined by

$$\begin{aligned} &(seq\{i\}(from, to, e))^{\mathcal{D},s,\beta} \\ &= \langle (e[i/n])^{\mathcal{D},s,\beta},\ (e[i/n+1])^{\mathcal{D},s,\beta},\ \ldots,\ (e[i/m-1])^{\mathcal{D},s,\beta} \rangle \end{aligned}$$

if $from^{\mathcal{D},s,\beta} = n < m = to^{\mathcal{D},s,\beta}$, and $(seq\{i\}(from, to, e))^{\mathcal{D},s,\beta} = \langle\rangle$ else. Here $e[i/n]$ is the term obtained from e by replacing all occurrences of the variable i by the integer n.

Values contained in sequences can be accessed by $seqGet_A : Seq \times Int \to A$. For short, we usually write $R[i]$ instead of $seqGet_{Any}(R, i)$.

2.1.5 Substitutions

As usual, $[x_1/t_1, \ldots, x_n/t_n]$ denotes a substitution. The application of a substitution $[x_1/t_1, \ldots, x_n/t_n]$ on a term t, denoted by $t[x_1/t_1, \ldots, x_n/t_n]$, yields the term obtained from t by simultaneously replacing all occurrences of variable x_i by term t_i.

2.1.6 Calculus

Java DL comes with a sequent calculus. Sequents have the form $\Gamma \Longrightarrow \Delta$, where Γ (called *antecedent*) and Δ (called *succedent*) are finite sets of formulas. The meaning of sequents is that of their *meaning formula* $\bigwedge \Gamma \rightarrow \bigvee \Delta$. The calculus consists of a set of *schematic rules* which operate on sequents. Schematic rules have the form

$$\frac{\Gamma_1 \Longrightarrow \Delta_1 \quad \ldots \quad \Gamma_n \Longrightarrow \Delta_n}{\Gamma \Longrightarrow \Delta}$$

where the *conclusion* $\Gamma \Longrightarrow \Delta$ and the *premisses* $\Gamma_1 \Longrightarrow \Delta_1$, …, $\Gamma_n \Longrightarrow \Delta_n$ may contain *schema variables*. Schema variables are placeholders which are substituted by terms or formulas if the rule schema is applied on a concreted sequent. Rules are always applied bottom up, whereas the direction of logical consequence is top down. A rule is sound, if the universal validity of the premisses implies the universal validity of its conclusion. Rules without premiss are called *axiom*.

A special kind of schematic rules are *rewrite rules*. They have the form $x_1 \rightsquigarrow x_2$ where x_1 and x_2 are either formulas or terms of the same type. The rule $x_1 \rightsquigarrow x_2$ replaces an occurrence of x_1 in a sequent by x_2. A rewrite rule is sound, if x_1 is logically equivalent to x_2: in this case the meaning of the sequent remains the same.

A simple example of a schematic rule is "andRight":

$$\mathsf{andRight}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \wedge \psi, \Delta}$$

The rule expresses that if ϕ and ψ are valid in context Γ, Δ, then also $\phi \wedge \psi$ is valid in context Γ, Δ. Another example is the rule "assignLocal":

$$\mathsf{assignLocal}\ \frac{\Gamma \Longrightarrow \{u\}\{\mathtt{a} := t\}[\pi\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\pi\ \mathtt{a}\ =\ t;\ \omega]\phi, \Delta}$$

where $\mathtt{a}$ is a program variable and t is a side-effect-free, normally terminating expression. The rule converts a Java assignment into an update. Modalities occurring in schematic rules usually have the form $[\pi\ p;\ \omega]$, where π is a *non-active Java prefix*, p is the *active statement* and ω is the remaining Java program. The non-active prefix contains opening braces of blocks, labels, beginnings of try-catch-finally blocks etc., whereas the active statement is the next command to be executed.

In the sequent calculus proofs are represented by *closed proof trees*. The nodes of a proof tree are annotated with sequents, where the children of each node

N are annotated with (instantiations of) the premisses of a rule schema which is applicable to the sequent of N. Each inner node is additionally annotated with the (instantiated) rule which relates the node to its children. A branch of a proof tree is closed, if its leaf is annotated with an axiom. A proof tree is closed if all branches of the tree are closed.

Java DL formulas are automatically generated by the KeY tool from Java programs annotated with specifications written in the Java Modeling Language. The next section gives a short introduction to the Java Modeling Language.

2.2 Java Modeling Language

The Java Modeling Language (JML) introduced by Leavens et al. [2006, 2008] is a popular language for the behavioral specification of Java code. It adopts the design by contract (DBC) methodology (Meyer [1988]). Java expressions enriched with specification constructs such as quantifiers are used to write assertions, such as pre- and postconditions and invariants. Weiß [2011] introduced a dialect of JML, called JML*, which is suitable for modular specifications. JML* extends JML mainly by the concept of dynamic frames (Kassios [2011]). The approaches presented in this thesis are based on JML*, but in most places the text does not distinguish the two versions.

This section shortly explains the JML* specification entities most important in the context of this thesis, namely method contracts, loop invariants, class invariants, ghost fields and model fields. For a more comprehensive introduction to JML and JML* see for instance Weiß [2011].

Listing 2.1 shows a simple implementation of a password checker. User names and passwords are represented as integers and stored in arrays `names` and `passwords`, respectively, where the password stored at position i belongs to the user name stored at that same position. The integer `numOfFailedChecks` holds the current number of consecutively unsuccessful login attempts. Login attempts are checked by the method `check` which takes a user name and a password as parameters. If there exists an index i such that the array `names` contains the passed user name and at which the array `passwords` contains the password, then the method returns `true` and resets `numOfFailedChecks` to zero. Otherwise, `check` returns `false` and increments `numOfFailedChecks`.

The source code of the example is annotated with JML specifications. Each specification is included in a special comment, either starting with `//@` or with `/*@`. The following sections explain the intuitive meaning the specifications.

```
1  class PasswordFile {
2    private int[] names, passwords;
3    private int numOfFailedChecks;
4    //@ invariant names.length == passwords.length;
5
6    /*@ public model \locset rep;
7      @ private represents rep =
8      @   \set_union(\locset(names, passwords, numOfFailedChecks),
9      @   \set_union(names[*], passwords[*]));
10     @ accessible rep : rep;
11     @*/
12
13   /*@ private normal_behavior
14     @   ensures     \result ==
15     @               (\exists int i; 0<=i && i<names.length;
16     @                   names[i]==user && passwords[i]==password);
17     @ public normal_behavior
18     @   assignable  rep;
19     @*/
20   public boolean check(int user, int password) {
21     /*@ loop_invariant  0 <= i && i <= names.length &&
22       @                 ( \forall int j; 0 <= j && j < i;
23       @                   !(   names[j]==user
24       @                     && passwords[j]==password) );
25       @ assignable  numOfFailedChecks;
26       @ decreases   names.length - i;
27       @*/
28     for (int i = 0; i < names.length; i++) {
29       if (names[i] == user && passwords[i] == password) {
30         numOfFailedChecks = 0;
31         return true;
32       }
33     }
34     numOfFailedChecks++;
35     return false;
36   }
37 }
```

Listing 2.1: Example of a password checker in Java with a full functional JML-specification.

2.2.1 Method Contracts

The most basic concept in JML are method contracts. Method contracts are placed in front of Java methods. They specify a contract between the caller of a method and the method itself. A contract usually consists of a precondition which has to be fulfilled by the caller and a postcondition which is guaranteed by the method if the precondition holds.

Listing 2.1 shows in lines 13 to 19 two contracts for the method `check`. Preconditions are specified with the help of the keyword **`requires`**. If the keyword is missing—as in this case—the precondition is implicitly defined as **`true`**. Therefore, the postcondition of the contract holds for any invocation of `check`. Postconditions are specified with the help of the keyword **`ensures`**. The postcondition in lines 14 to 16 says that the result of the method is **`true`** if and only if there exists an index i at which the array `names` equals the value passed by the parameter `user` and at which the array `passwords` equals the value passed by the parameter `password`. If the postcondition is missing—as in the second contract—it is implicitly defined as **`true`**.

`ensures` clauses give guarantees for normal termination only. In case exceptions are thrown, additional clauses are needed to specify which exceptions are allowed to be thrown and what postcondition should hold in such a case (see for instance Weiß [2011]). The contracts of `check` specify with the help of the keyword **`normal_behavior`** that `check` does not throw exceptions.

Similar to Java, JML supports visibility modifiers. They follow essentially the same rules as in Java, but JML has one additional rule for visibility that matters in the context of this thesis: an annotation may not refer to names (for instance field names) that are more hidden than the visibility of the annotation itself. In the example, the first contract is private. It is visible within the class `PasswordFile` only. Therefore it may talk about the private fields of `PasswordFile`. In contrast, the second contract is publicly visible and may refer to public names only.

Beside pre- and postconditions, method contracts can specify which heap locations may be altered by a call to the corresponding method. This is done with the help of the keyword **`assignable`**. The second contract of `check` specifies that at most the locations defined by the model field **`rep`** may be changed. What model fields are and how they can be used is discussed in the next section.

2.2.2 Model Fields

Model fields can be interpreted as functions from the heap to their specified type. They are declared like usual Java fields, but within JML annotations and with the additional **model** modifier. In the example, line 6 declares the model field **rep** of type **\locset**. The type **\locset** is JML* specific. Its elements are sets of heap locations, where a heap location is a pair of an object and a field (see Section 2.1.3). The evaluation of model fields is defined by **represents** annotations. **represents** annotations constrain the function which is represented by a model field. In the example (lines 7 to 9), the model field **rep** represents the heap locations

$$\{(\texttt{this}, \texttt{names}), (\texttt{this}, \texttt{passwords}), (\texttt{this}, \texttt{numOfFailedChecks})\}$$
$$\cup \bigcup_i (\texttt{this.names}, arr(i)) \cup \bigcup_i (\texttt{this.passwords}, arr(i)) \ .$$

Note that, for instance, the evaluation of `this.names` depends on the heap. Therefore, **rep** is indeed a function from the heap to sets of heap locations.

Model fields are used to abstract from implementation details. This is in particular useful for the specification of interfaces. In the example, the model field **rep** is used to enforce the principle of information hiding. All fields occurring in the set of heap locations abstracted by **rep** are private and thus not visible outside of `PasswordFile`. Therefore, those fields may not be used in a public contract for `check`. Still, we want to express that only private heap locations are changed by `check`, such that callers can be sure that none of "their" heap locations change by a call to `check`. This can be achieved by the usage of the model field **rep** in the contract of `check` whose definition is not known by the caller (the represents annotation is private). Callers need to know only two things: (1) that their "own" heap locations are disjoint from the locations represented by **rep**; and (2) that the set of locations represented by **rep** does not change if a location outside of **rep** is changed. Item (2) is ensured by line 10. The **accessible** annotation specifies on which heap locations the function represented by the model field depends. In the example, the specification says that the result of **rep** depends at most on the locations in **rep** itself. This kind of specification is called self-framing (see Kassios [2011] or Weiß [2011]). It exactly expresses (2). Item 1 has to be specified by the caller and checked on every call of `check`.

Chapter 4 frequently uses model fields of type sequence (note that the type sequence is also JML* specific):

```
/*@ model \seq pwdFileManager;
  @ represents pwdFileManager =
```

```
@     \seq(names,
@          \seq_def(int i; 0; names.length; names[i]),
@          passwords,
@          \seq_def(int i; 0; passwords.length; passwords[i]));
@*/
```

The first line declares the model field `pwdFileManager` of type sequence. Lines 2 to 6 define `pwdFileManager` to be the finite sequence

$$\langle \texttt{names}, \langle \texttt{names}[0], \ldots, \texttt{names}[\texttt{names.length} - 1] \rangle, \\ \texttt{passwords}, \langle \texttt{passwords}[0], \ldots, \texttt{passwords}[\texttt{passwords.length} - 1] \rangle \rangle \ . \tag{2.1}$$

As the example shows, sequences can be nested.

2.2.3 Ghost Fields

Ghost fields serve the same purpose as model fields, but are handled slightly differently. They are declared as usual Java fields, but within JML annotations and with the additional modifier **ghost**. For instance, the annotation

```
//@ ghost \seq pwdFileManager;
```

declares the ghost field `pwdFileManager` of type sequence. Ghost fields can be used within specifications only, but otherwise behave like usual Java fields. They can be assigned with the help of special **set** statements within the body of a method. **set** statements have to be included in JML comments. Except for the preceding **set** keyword the syntax is as in usual assignments:[3]

```
/*@ set pwdFileManager =
  @     \seq(names,
  @          \seq_def(int i; 0; names.length; names[i]),
  @          passwords,
  @          \seq_def(int i; 0; passwords.length; passwords[i]));
  @*/
```

In order to ensure that the ghost field `pwdFileManager` always represents the finite sequence (2.1), as it is the case for the model field from the last section, it is appropriate to use a class invariant (see next section) which expresses this equality.

[3] Unfortunately, the convenient syntax used in the expression `seq(names, ...)` cannot be parsed in `set` statements in the current version of KeY. The expression has to be replaced by a combination of `seq_concat` and `seq_singleton` expressions instead. For an example for the syntax currently supported by KeY see Section 4.4.1.

Usually, theorem provers can handle ghost fields more efficiently than model fields. Model fields, on the other hand, might be more attractive to specifiers because they cause less specification overhead. This thesis usually prefers ghost fields, not least because this allows the handling of bigger programs. Moreover, the additionally required **set** statements explicitly describe the transitions from one abstract state to another. This documents the behavior of the abstraction and is therefore valuable for understanding the code, even if it leads to additional specification overhead.

2.2.4 Class Invariants

Intuitively, class invariants are properties which hold for all objects of a class—ideally—throughout the execution of the whole program. There are cases, though, where it is unavoidable to violate class invariants. For instance, multiple program variables cannot be updated at once in Java: let `a`, `b` and `c` be three fields. Then the invariant `c == a + b` has to be violated each time `a` is assigned a new value.

In JML* all class invariants of the object referenced by **this** hold before and after the execution of a method of the object, but this default behavior can be overwritten. The modifier **helper** in front of a method disables the default behavior. Moreover, if `o` is an expression of object type, then the expression **\invariant_for**`(o)` can be used to refer to the invariant of `o`, for instance in pre- and postconditions or other invariants. Thus, it is possible to enforce class invariants as needed, but also to leave them open.

Class invariants are defined by JML annotations starting with the keyword **invariant**. Line 4 shows such a definition. The invariant says that the lengths of the arrays `names` and `passwords` coincide.

The class invariant of an object becomes a heap-dependent predicate in Java DL: the invariant of object o holds in heap h if and only if the expression `java.lang.Object::<inv>(`h`,`o`)` evaluates to true. For short, we usually write o.`<inv>` instead of `java.lang.Object::<inv>(heap,`o`)`.

2.2.5 Loop Invariants

In order to support the theorem prover in proving properties about programs, JML allows the specifier to state auxiliary lemmas within the body of a method. The most important auxiliary lemmas for software verification systems are loop invariants. Loop invariants state properties which hold before and after each execution of the loop body. In the example, lines 21 to 24 state that before and

after each execution of the loop body the control variable `i` is in the range between zero (inclusive) and `names.length` (inclusive) and that for all positions `j` in between zero and `i` either the passed user name does not equal the one of `names[j]` or the passed password does not equal the one of `passwords[j]`.

Similar to method contracts, loop invariants can be augmented by **`assignable`** clauses which describe a set of heap locations which may be modified at most by the loop. In the example, the loop may modify at most the heap location (`this`, `numOfFailedChecks`). If termination of a loop has to be proven (as it is usually the case) the loop invariant can additionally be augmented by a ranking function. The ranking function is specified behind the **`decreases`** keyword. In the example, line 26 specifies that the value of `names.length - i` decreases in each iteration of the loop and that it is bounded by zero.

3 Language-Based Secure Information Flow

This chapter is a revised and extended version of parts of Scheben and Schmitt [2012] and Beckert et al. [2014].

Programs may leak information in a number of different ways. Usually, information leaks are categorized into explicit leaks, implicit leaks and side channels. Explicit leaks are the most obvious ones and easy to detect. If `low` is a publicly observable program variable (also called "low variable") and if `high` is a program variable containing secret information (also called "high variable"), then the program `low = low + high` has an explicit information leak: the secret is disclosed explicitly by an assignment. Implicit leaks are more subtle. They disclose information through the control flow of programs. For instance, the program **`if`**`(high > 0) {low = 0;}` **`else`** `{low = 1;}` is insecure, too: if `high` is greater than 0, then the final value of `low` is 0, else it is 1. The information whether `high` is greater than 0 or not is leaked. It goes without saying that in complex programs these leaks become much more subtle. Finally, information may leak by physical properties like time, power consumption, heat generation and others. These kinds of leakages are called side channels. Like most work on language-based information flow analysis, this thesis is concerned with explicit and implicit leaks only.

In the following, the notation of language-based secure information flow will be formalized. As a first step towards a formalization, the capabilities of attackers have to be fixed.

3.1 Attacker Model

Attackers watch program runs and observe two things about them: Firstly, they know the program code. This is formalized by the assumption that they know which initial state of a program run relates to which final state. Secondly, they can observe parts of the initial and the final program state. What attackers are able to observe of those states is described by sets of so-called *observation*

expressions. Observation expressions can be thought of as arbitrary Java DL terms or JML expressions:

Definition 1. *An* observation expression *can be:*

1. *A program variable (including method parameters).*
2. $e.f$ *for* e *an expression of type* C *and* f *a field declared in* C *(including ghost and model fields).*
3. $e[t]$ *if* e *is an expression of array type, and* t *of integer type.*
4. $op(e_1, \ldots, e_k)$ *if* op *is a data type operation and* e_i *expressions of matching type.*
5. *The usual conditional operator* $b\ ?\ e_1 : e_2$ *(*e_1*,* e_2 *have to be of the same type).*
6. *The sequence definition operator* $seq\{i\}(from, to, e)$*. Its semantics is defined by*

$$\begin{aligned}&(seq\{i\}(from, to, e))^s\\ &= \langle (e[i/n])^s,\ (e[i/n+1])^s,\ \ldots,\ (e[i/m-1])^s \rangle\end{aligned}$$

if $from^s = n < m = to^s$*, and* $(seq\{i\}(from, to, e))^s = \langle\rangle$ *else. Here* $e[i/n]$ *is the expression obtained from* e *by replacing all occurrences of the variable* i *by the literal* n*.*

Attackers can observe the values of observation expressions and additionally know which value belongs to which expression. One can imagine that the expressions and the corresponding evaluations are printed on a screen. Attackers may compute any computable function on the observed values and compare the results by equality. The set of observation expressions describing the information observable in initial program states might differ from the one describing the information observable in final program states.

Technically, this thesis uses sequences of observation expressions (which are themselves observation expressions), instead of sets, to describe the information observable by attackers. The concatenation of two observation expressions R_1 and R_2 is denoted by $R_1; R_2$.

Let R be an observation expression. If s is the initial or the final state of a program run, then, formally, attackers are able to observe the tuple (R, R^s), where $R^s = \langle e_1^s, \ldots, e_k^s \rangle$ if $R = \langle e_1, \ldots, e_k \rangle$. Because attackers can compute any computable function on the observed values, they can deduce in particular that e_i^s is the value of the expression e_i in state s (for $1 \leq i \leq k$) and they can compare any two values, $e_i^s = e_j^{s'}$, for any pair of states s and s'.

The above attacker model does not assume that attackers who can observe an object o automatically can also observe any location reachable by o. In particular when modular information flow reasoning is studied later on, such an

attacker model would be too strong. On the other hand, an attacker with the capabilities to observe all reachable locations can be modeled with the above attacker model with the help of recursively defined ghost (or model) fields. Java programs as attackers can also be modeled, see Section 9.1.

3.2 Formal Definition of Secure Information Flow

Given the attacker model of Section 3.1, this section defines what it means for a program to have secure information flow. Firstly, it is helpful to introduce a notion for the indistinguishability of two states.

Definition 2 (Agreement of states)**.** *Let R be an observation expression.*

Two states s, s' agree on R, abbreviated by agree(R, s, s'), if and only if $R^s = R^{s'}$.

Thus, two states s and s' agree on R if an attacker cannot distinguish them: $f(R, R^s)$ equals $f(R, R^{s'})$ for any function f. The following definition states what it means for a program α (when started in a state s) to allow information flow only from R_1 to R_2 under condition ϕ, denoted by flow$(s, \alpha, R_1, R_2, \phi)$.

Definition 3 (Conditional Noninterference)**.** *Let α be a program, R_1, R_2 observation expressions and ϕ a formula.*

Program α allows information to flow only from R_1 to R_2 when started in s_1 under condition ϕ, denoted by flow$(s_1, \alpha, R_1, R_2, \phi)$, if and only if for all states s_1', s_2, s_2' such that α started in s_1 terminates in s_2 and α started in s_1' terminates in s_2' the following applies:

$$\textit{if} \quad s_1 \vDash \phi, s_1' \vDash \phi \textit{ and agree}(R_1, s_1, s_1') \quad \textit{then} \quad \textit{agree}(R_2, s_2, s_2').$$

flow(α, R_1, R_2, ϕ) denotes the case that flow$(s_1, \alpha, R_1, R_2, \phi)$ holds for all states s_1; flow(α, R_1, R_2) abbreviates flow$(\alpha, R_1, R_2, true)$.

The observation expressions R_1, R_2 describe the publicly available information of the initial and final state of a program run. In simple cases R_i is build up of those program variables and fields which are considered low. Usually, R_1 and R_2 will coincide. However, to declassify an expression e_{decl}, for instance, one would choose $R_1 = R_2; e_{decl}$.

The next theorem shows that Definition 3 fits to the attacker model of Section 3.1. If flow(s_1, α, R_1, R_2) holds, then attackers with the capabilities of Section 3.1 are not able to distinguish s_1 by execution of α from any other initial state s_1' which agrees on R_1 with s_1.

Theorem 4. *If flow(s_1, α, R_1, R_2) holds and if non-termination is not observable, then attackers with the capabilities of Section 3.1 are not able to distinguish s_1 by execution of α from states s_1' with $(R_1, R_1^{s_1'}) = (R_1, R_1^{s_1})$.*

Proof. Assume two states s_1, s_1' such that

$$(R_1, R_1^{s_1'}) = (R_1, R_1^{s_1}). \tag{3.1}$$

If α terminates if started in s_1, then let s_2 denote this final state. Similarly, if α terminates if started in s_1', then let s_2' denote this final state. Attackers are able to observe at most the tuples $(R_2, R_2^{s_2})$ and $(R_2, R_2^{s_2'})$ in addition to $(R_1, R_1^{s_1})$ and $(R_1, R_1^{s_1'})$. By Definition 2, the equation $R_1^{s_1'} = R_1^{s_1}$ implies agree(R_1, s_1, s_1'). Therefore, flow(s_1, α, R_1, R_2) ensures that agree(R_2, s_2, s_2') holds. By Definition 2 the predicate agree(R_2, s_2, s_2') holds if and only if $R_2^{s_2} = R_2^{s_2'}$. Thus, it holds if and only if

$$(R_2, R_2^{s_2}) = (R_2, R_2^{s_2'}) \tag{3.2}$$

holds.

Because of (3.1) and (3.2), the result of $\delta((R_2, R_2^{s_2}), (R_1, R_1^{s_1}))$ equals the result of $\delta((R_2, R_2^{s_2'}), (R_1, R_1^{s_1'}))$ for any function δ. Therefore, attackers cannot tell by the observation of the initial and the final state of a run of α whether α was started in s_1 or s_1'. Thus, they cannot distinguish s_1 by execution of α from s_1'. □

Conditional noninterference is compositional in the following sense.

Lemma 5 (Compositionality of *flow*)**.** *Let α_1, α_2 be programs, α_1;α_2 their sequential composition. If flow$(s_1, \alpha_1, R_1, R_2, \phi_1)$, flow$(s_2, \alpha_2, R_2, R_3, \phi_2)$, $\vDash \phi_1 \rightarrow [\alpha_1]\phi_2$ and α_1 started in s_1 terminates in s_2 and α_2 started in s_2 terminates in s_3 then flow$(s_1, \alpha_1; \alpha_2, R_1, R_3, \phi_1)$ holds.*

Proof. Let s_1', s_2', s_3' be a second set of states such that α_1 started in s_1' terminates in s_2', α_2 started in s_2' terminates in s_3' and $s_1' \vDash \phi_1 \rightarrow [\alpha_1]\phi_2$. Assume that the precondition ϕ_1 holds in s_1 and s_2, in other words, assume $s_1 \vDash \phi_1$ and $s_1' \vDash \phi_1$. Additionally, assume agree(R_1, s_1, s_1'). Then, flow$(s_1, \alpha_1, R_1, R_2, \phi_1)$ implies agree(R_2, s_2, s_2'), $s_1 \vDash \phi_1 \rightarrow [\alpha_1]\phi_2$ implies $s_2 \vDash \phi_2$ and $s_1' \vDash \phi_1 \rightarrow [\alpha_1]\phi_2$ implies $s_2' \vDash \phi_2$. Finally, agree(R_2, s_2, s_2'), $s_2 \vDash \phi_2$, $s_2' \vDash \phi_2$ and flow$(s_2, \alpha_2, R_2, R_3, \phi_2)$ ensure agree(R_3, s_3, s_3'). □

3.3 Example

This section illustrates the above basic definitions by the frequently used password checker example. The example will be extended to a small banking system in the next chapter.

Listing 2.1 shows the considered implementation. As discussed in detail in Section 2.2, it consists of a class `PasswordFile` with two private arrays, `names` and `passwords`, which store the user names and their corresponding passwords at the same index. Obviously, the length of those two arrays has to coincide. This is formulated with the help of an JML-invariant in line 4. Further, the class contains a method `check` which takes a user name and a password. It checks whether there exists an index i at which the array `names` contains the user name and at which the array `passwords` contains the password. If such an index exists, the method returns true, otherwise false.

The arrays `names` and `passwords` and their contents are usually considered confidential whereas the parameters `user` and `password` as well as the not explicitly named **`return`**-variable are considered public (because callers know the user name and password they enter and learn the return value of the call). This security requirement can be expressed naturally by

$$\text{flow}(\texttt{boolean check}(\ldots)\{\ldots\}, \langle \texttt{user}, \texttt{password} \rangle, \langle \texttt{result} \rangle) \tag{3.3}$$

where `result` denotes a special variable containing the result of the method.

As the case studies show (Chapter 9), it is a quite common pattern to split the verification of (3.3) into showing that (i) the security requirement holds in case the class invariant is valid, that is,

$$\text{flow}(\texttt{boolean check}(\ldots)\{\ldots\}, \langle \texttt{user}, \texttt{password} \rangle, \langle \texttt{result} \rangle), \texttt{self.<inv>})$$

and (ii) that the class invariant cannot be broken. Here, `<inv>` denotes the observer symbol for class invariants, see Section 2.2.4.

Note that neither the information flow property (3.3) nor

$$\text{flow}(\texttt{boolean check}(\ldots)\{\ldots\}, \langle \texttt{user}, \texttt{password} \rangle, \langle \texttt{result} \rangle, \texttt{self.<inv>})$$

is fulfilled: the method `check` necessarily leaks some information about the contents of `names` and `passwords`—the information whether there exists an index i at which the array `names` contains the passed user name and at which

the array `passwords` contains the passed password. This intentional information release is usually dealt with by declassification. As mentioned in Section 3.2, declassification can be expressed by flow directly:

$$\begin{aligned}&\mathrm{flow}(\texttt{boolean check(int user, int password)}\{\ldots\},\\ &\quad\langle\texttt{user}, \texttt{password},\\ &\qquad\exists i.(\quad 0 \le i \wedge i < \texttt{names}.\mathit{length}\\ &\qquad\qquad\wedge \texttt{names}[i] = \texttt{user} \wedge \texttt{passwords}[i] = \texttt{password})\rangle,\\ &\quad\langle\texttt{result}\rangle,\\ &\quad\texttt{self.<inv>})\end{aligned}$$

allows exactly the intended flow of information.

3.4 Multilevel Noninterference

Section 3.2 defines noninterference for a classification into low and high symbols only. Usually this is found to be too coarse for practicable information flow analysis. Most existing analysis tools use *lattices of security levels* instead (see for instance Myers [1999a]). In a security lattice information may flow from lower levels to higher ones only. *Security policies* are usually defined as mappings of program variables to security levels of the lattice (see for instance Denning and Denning [1977]).

Let α be a program with program variables PV, let $\mathcal{L} = (L, \rightsquigarrow)$ be a security lattice with security levels L and flow relation $\rightsquigarrow$, let $p : \mathrm{PV} \to L$ be a security policy and let $x \in \mathrm{PV}$. We denote the set of variables $\{y \mid p(y) \rightsquigarrow p(x)\}$ which belong to the same or a lower security level than x by $\mathrm{PV}_{\rightsquigarrow p(x)}$.

Let $\preceq$ be an arbitrary ordering on PV. Any subset X of PV defines in combination with $\preceq$ a sequence. By abuse of notation this sequence is denoted by X, too.

Definition 6 (Multilevel Noninterference)**.** *Let α be a program with program variables* PV *and let $\mathcal{L} = (L, \rightsquigarrow)$ be a security lattice with security levels L and flow relation $\rightsquigarrow$. Let further $p : \mathrm{PV} \to L$ be a security policy.*

Program α allows information to flow only according to $\mathcal{L}$ and p when started in s_1 under condition ϕ, denoted by flow$(s_1, \alpha, \mathcal{L}, p, \phi)$, if and only if for all $x \in \mathrm{PV}$ and for all states s_1', s_2, s_2' such that α started in s_1 terminates in s_2 and α started in s_1' terminates in s_2', we have

$$\textit{if} \quad s_1 \vDash \phi,\ s_1' \vDash \phi \textit{ and agree}(\mathrm{PV}_{\rightsquigarrow p(x)}, s_1, s_1') \quad \textit{then} \quad \textit{agree}(x, s_2, s_2').$$

flow$(\alpha, \mathcal{L}, p, \phi)$ *denotes the case that flow*$(s_1, \alpha, \mathcal{L}, p, \phi)$ *holds for all states* s_1*. Additionally, flow*$(\alpha, \mathcal{L}, p)$ *abbreviates flow*$(\alpha, \mathcal{L}, p, \mathit{true})$.

As well known in literature, multilevel noninterference can be expressed by two-level noninterference (as defined in Section 3.2):

Lemma 7. *Let* α *be a program with program variables* PV *and let* $\mathcal{L} = (L, \rightsquigarrow)$ *be a security lattice with security levels* L *and flow relation* $\rightsquigarrow$*. Let further* $p : \mathrm{PV} \to L$ *be a security policy.*

flow$(s_1, \alpha, \mathcal{L}, p, \phi)$ *if and only if flow*$(s_1, \alpha, \mathrm{PV}_{\rightsquigarrow p(x)}, x, \phi)$ *holds for all* $x \in \mathrm{PV}$.

Proof. By Definition 6 flow$(s_1, \alpha, \mathcal{L}, p, \phi)$ holds if and only if

> for all $x \in \mathrm{PV}$ and for all states s_1', s_2, s_2' such that α started in s_1 terminates in s_2 and α started in s_1' terminates in s_2', we have
> if $s_1 \vDash \phi$, $s_1' \vDash \phi$ and agree$(\mathrm{PV}_{\rightsquigarrow p(x)}, s_1, s_1')$ then agree(x, s_2, s_2').

By Definition 3 the last part,

> for all states s_1', s_2, s_2' such that α started in s_1 terminates in s_2 and α started in s_1' terminates in s_2', we have
> if $s_1 \vDash \phi$, $s_1' \vDash \phi$ and agree$(\mathrm{PV}_{\rightsquigarrow p(x)}, s_1, s_1')$ then agree(x, s_2, s_2'),

holds if and only if flow$(s_1, \alpha, \mathrm{PV}_{\rightsquigarrow p(x)}, x, \phi)$.

Hence, flow$(s_1, \alpha, \mathcal{L}, p, \phi)$ if and only if flow$(s_1, \alpha, \mathrm{PV}_{\rightsquigarrow p(x)}, x, \phi)$ holds for all $x \in \mathrm{PV}$. □

This thesis usually considers sets of two-level noninterference properties flow$(s_1, \alpha, R_1, R_2, \phi)$. As Lemma 7 shows, this is at least as expressive as using security lattices.

3.5 Discussion

The definition of secure information flow is as usual except that it allows for finer-grained specifications by the usage of observation expressions instead of variables and fields only. This is in particular useful for the specification of declassification, but also allows for knowledge-based specifications (see Chapter 4). Furthermore, the definition—and the thesis in general—considers conditional information flow which is studied only rarely. As already observed by Amtoft et al. [2008], conditional information flows are useful in modular

information flow verification. Section 5.4 gives a comprehensive overview on related work on conditional information flow.

4 JML Extensions for Specifying Secure Information Flow

This chapter is a revised and extended version of Scheben and Schmitt [2012].

A convenient way to specify properties of programs is annotating them in a formal specification language utilized for the programming language under consideration. The most popular approach for specifying functional properties of Java programs is JML. This chapter shows how JML can be extended to the specification of information flow properties.

Specifying information flow properties for variables and fields of object type poses particular problems, as will be shown in Chapter 6. This chapter does not consider those particularities but handles variables and fields of object type like variables and fields of primitive type. This is safe but potentially more restrictive than necessary. Chapter 6 shows how the JML extension presented here can be augmented such that variables and fields of object type are treated in a less restrictive way.

The JML extension is designed to be suitable for highly precise, fully modular information flow verification. To this end, it allows writing information flow specifications for interfaces without knowing their implementations as well as the integration of functional and information flow specifications. Additionally, it is designed to be suitable for a broad range of verification techniques.

Most existing practicable information flow analysis tools use lattices of security levels and annotate variables and fields with levels of this lattice (see for instance Myers [1999a]). This approach poses the problem that the specifier has to figure out which program variable has to be assigned which security level by an investigation of the high level security specifications. This can be nontrivial, as can be seen in Section 4.2. Therefore, this chapter will present a knowledge-based approach which concentrates on the specification of the knowledge which actors of a system may have about the system. Because multilevel noninterference can be expressed by two-level noninterference (see Section 3.4), any security policy implies a specification in this approach, but not

necessarily vice versa, as will be shown in Section 4.2. Knowledge-based specifications promise to be easier deduceable from high-level security requirements than suitable classical security policies.

The next section will illustrate the approach with the help of a banking example. To abstract from program level details, the motivation will use UML diagrams. The subsequent definitions in Section 4.3 will be on the Java level again.

4.1 Illustration of Knowledge-Based Specification

Figure 4.1 shows a use-case diagram and a class diagram for a banking example. The actors in this example are bank-customers and bank-employees. Customers can view the balance of their accounts and draw money while employees may observe the balance of all accounts. A reasonable security requirement is that customers may know at most the data belonging to their accounts while employees may know everything except the passwords of the accounts. This requirement is illustrated in the object diagram of Figure 4.2 for three customers and one employee. Each kind of smiley represents an actor and marks the fields which are allowed to be observed/known by this actor.

Figure 4.3 summarizes the above requirements at the level of a class diagram. The fields observable by an actor are defined with the help of observation expressions (Definition 1). The symbols `userAccounts` and `bankAccounts` denote the shown associations between the classes `Bank` and `UserAccount` in the first case and between `UserAccount` and `BankAccount` in the second. On the programming language level these are implemented as fields of type `UserAccount[]` and `BankAccount[]`, see Listings 4.2 and 4.3. The example uses one observation expression per actor and class to define what is observable of an object by an actor. The observation expression $\langle$`this`, `balance`, `id`$\rangle$ with the name customerViewOnBankAccount, for instance, is used to express that customers which may observe an object o of class `BankAccount` may also observe the value of o.`balance` and o.`id`. Which objects of class `BankAccount` a customer is permitted to observe is defined one level higher, in the class `UserAccount`. Here, the observation expression customerView defines that a customer who is able to observe a user account o may additionally observe (1) o.`userID`, (2) o.`incorrectLogins`, (3) the object o.`bankAccounts` including its contents as defined by the expression customerViewOnBankAccount, and (4) o.`password` including its contents. What an employee may see is defined in a similar fashion.

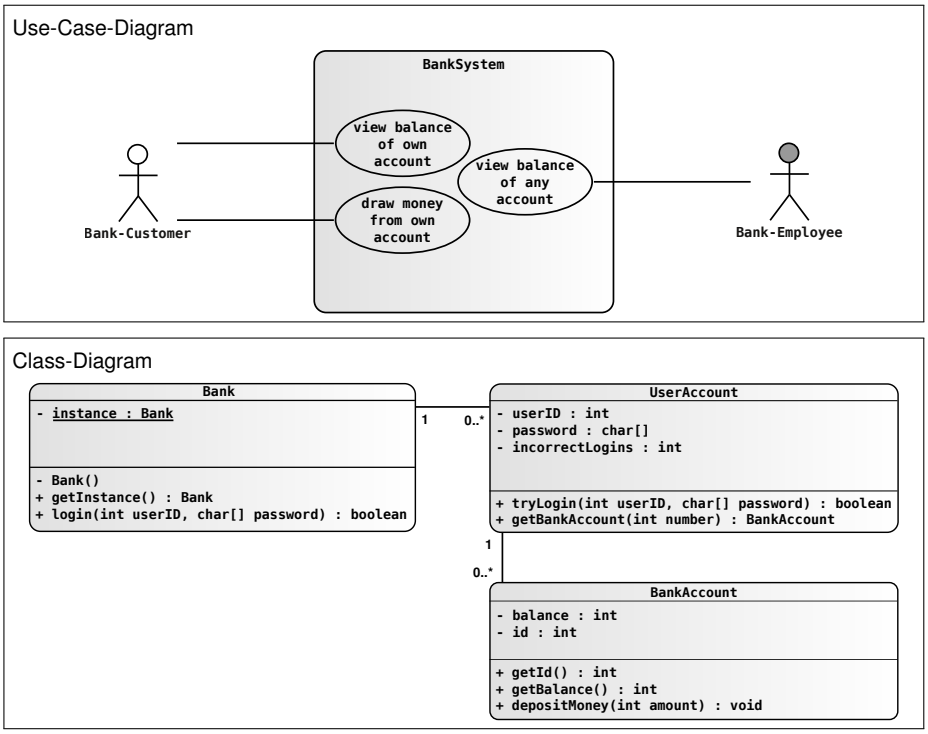


Figure 4.1: Banking scenario: use-case and class diagram.

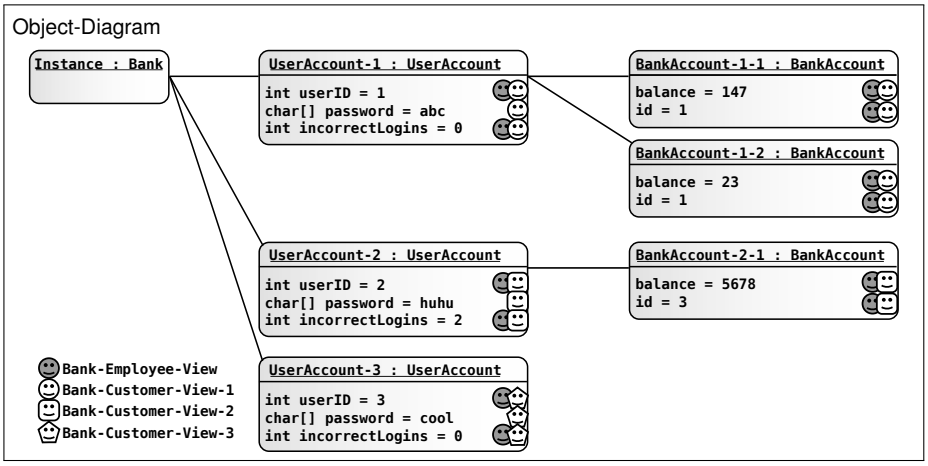


Figure 4.2: Banking scenario: object diagram with annotated views.

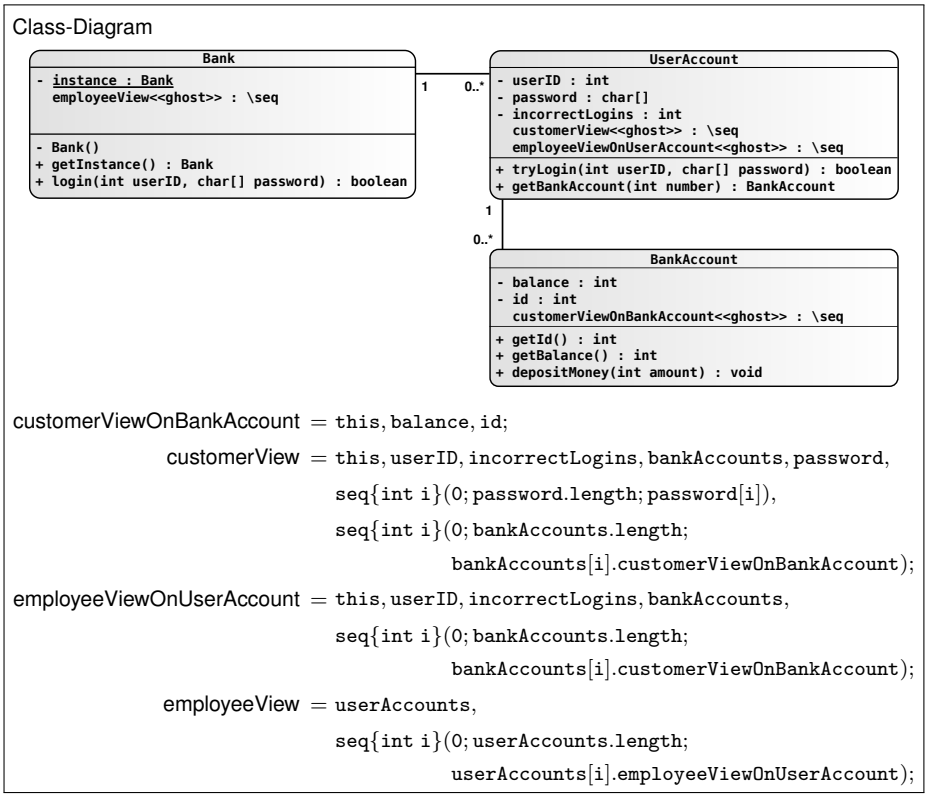


Figure 4.3: Class diagram of Figure 4.1 with annotated views.

How Java code can be annotated with named observation expressions is shown in Section 4.3.5 and illustrated on the banking example in Section 4.4. In the following, the expression observable by an actor will be called the *view* of the actor on the system. In some cases actors may observe different information in the prestate and in the poststate of a program run. In this case the views of actors consist of a prestate view, usually denoted by R_1, and a poststate view, usually denoted by R_2. Intuitively a program α is secure if and only if no actor can learn anything new by an execution of α. Thus, a program is secure if and only if flow(α, R_1, R_2) holds for all views (R_1, R_2) of all considered actors.

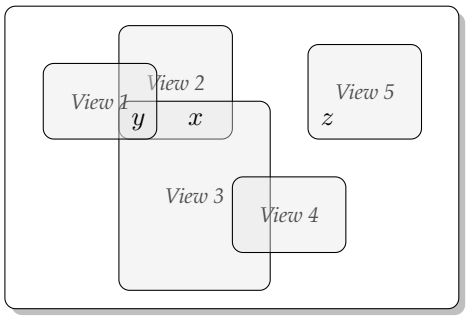
The observation expressions above are build up from fields and sequences of fields only. More complex specifications can be written by using more complex observation expressions. Examples where this is useful can be found in Section 4.4 and Chaper 9. Declassification and erasure can be captured by choosing different observation expressions for the initial and final states of a program

run, see Section 3.2.

4.2 Knowledge-Based Specification vs. Classical Security Policies

Classical security policies of lattice approaches can be expressed with the help of sets of views by the usual translation of multilevel noninterference to two-level noninterference, see Section 3.4. The other way around, the translation of knowledge-based information flow specifications into classical security policies is in general infeasible, because classical security policies constraint the information flow on the granularity of variables/fields only and cannot constraint the information flow of more general expressions like observation expressions. The view $(\langle \mathtt{x} + \mathtt{y} < 0 \rangle, \langle \mathtt{result} == 0 \rangle)$, for instance, cannot be translated into a classical security policy. Still, a knowledge-based specification can be translated into classical security policies, if all pre- and postviews occurring in the specification are sequences of variables and fields only.

Example. *The following figure depicts five views. For simplicity, the prestate view coincides with the poststate view for all depicted views. In other words, all views have the form (R, R), where R is a sequence of variables and fields.*



The rectangles overlap in those parts where the corresponding observation expressions contain the same variables. The variable x, for instance, is part of Views 2 and 3, y is part of Views 1, 2 and 3 and z is part of View 5 only. In this example, information may flow form y to x because x is contained in all views, in which y occurs. On the other hand, information may not flow the other way around. Otherwise, actors which are allowed to observe View 1 would be able to learn the value of x, which they are not supposed to know. Similarly, no information flow is allowed between z and x. The above observations suggest that the set of views can be translated to a security lattice consisting of sets of variables and fields with set-union and set-intersection as join and

meet operators. The security policy maps each variable to the intersection of all views it is part of.

In general, information may flow from y to x if and only if $x \in R_2$ implies $y \in R_1$ for all views (R_1, R_2). Let $dep(x)$ denote the set of variables on which the final value of x may depend on, that is, $dep(x) := \{y \mid x \in R_2 \Rightarrow y \in R_1 \text{ for all views } (R_1, R_2)\}$. The security lattice is the power set of the set of all variables and fields occurring in the program under consideration with set-intersection and set-union as meet and join operators. The security policy is defined by $x \mapsto dep(x)$ for every variable and field x. Information may flow from y to x if the level $dep(y)$ is a subset of the level $dep(x)$.

If the information flow behavior of a program is specified with the help of a security lattice, the specifier essentially has to do the above translation by hand. Knowledge-based specifications avoid this extra step.

4.3 Extending JML* for Noninterference Specifications

JML* (Weiß [2011]) is designed as a *design by contract* (Meyer [1988]) style specification language. To achieve a natural integration of information flow and functional specifications, the JML* extension uses design by contract style for the specification of information flow properties, too. Conditional noninterference with declassification and erasure is specified by information flow method contracts. Similar to functional method contracts, which specify the functional behavior of methods, information flow contracts specify the information flow behavior of methods.

4.3.1 Information Flow Method Contracts

The information flow behavior of a method is specified in the JML* extension with the help of a new method contract clause, the *determines clause*. A method contract may have multiple determines clauses. Each determines clause defines a restriction on the information flow, as illustrated in the following example.

Example. *Consider the method* `tryLogin` *from the class* `UserAccount` *of the banking example (Figure 4.3). The security property requires that bank customers and employees do not learn anything by execution of the method. Therefore, the final value of bankCustomerView needs to depend at most on itself. The same holds for the employee view. This is expressed by determines clauses as follows:*

```
determines_clause =
  "determines", ( expressions | "\nothing" ),
  "\by",        ( expressions | "\itself" | "\nothing" ),
  {   [ "\declassifies", expressions ]
    | [ "\erases",       expressions ] };
expressions = expression, { ",", expression };
```

Listing 4.1: EBNF of determines clauses.

```
/*@ determines  bankCustomerView \by \itself;
  @ determines  employeeViewOnUserAccount \by \itself;
  @ assignable  this.*;
  @*/
public boolean tryLogin(int userID, char[] password)
```

Here, bankCustomerView and employeeViewOnUserAccount are the observation expressions of Figure 4.3. Note that the determines clauses restrict the information flow to locations mentioned in bankCustomerView and employeeViewOnUserAccount only. To ensure that information does not leak to other locations, the contract has to be augmented by an assignable clause. The assignable clause in this example ensures that at most fields of the **this** *object are modified and therefore excludes the possibility of an information flow to locations other then the ones mentioned in bankCustomerView and employeeViewOnUserAccount.*

The syntax of determines clauses is defined by the Extended Backus–Naur Form (EBNF) of Listing 4.1. Here, `expression` is a usual JML expression.

The semantics of the determines clause is defined with the help of conditional noninterference (Definition 3): Let R_{post} be defined as the concatenation of the expressions behind the **determines** keyword and the expressions behind the **\erases** keywords. Let R_{pre} be defined as the concatenation of the expressions behind the **\by** keyword and the expressions after the **\declassifies** keywords, where **\nothing** is identified with the empty sequence and where **\itself** is identified with R_{post}. Let further ϕ_{pre} be the precondition of the contract defined as usual by requires clauses and class invariants. A method `m` fulfills a determines clause if and only if flow$(\mathtt{m}, R_{pre}, R_{post}, \phi_{pre})$ is valid.

Beside methods, also blocks and loops can be annotated by information flow specifications. The next two sections show how this is done. They are followed by a note on information flow class invariants in Section 4.3.4 and a note on naming views in Section 4.3.5.

4.3.2 Information Flow Block Contracts

Information flow block contracts are very similar to information flow method contracts, but they are annotated to blocks instead of methods and accordingly specify the information flow behavior of blocks.

Information flow contracts augment functional block contracts (Wacker [2012]) by determines clauses. The syntax of those determines clauses is the same as the one for method contracts, see Listing 4.1. A block `b` fulfills a determines clause if and only if flow$(\texttt{b}, R_{pre}, R_{post}, \phi_{pre})$ is valid, where R_{pre}, R_{post} and ϕ_{pre} are defined as in the last section.

An application of information flow block contracts can be found in Section 5.2.

4.3.3 Information Flow Loop Invariants

From a specification point of view, information flow loop invariants are not very different from information flow method or block contracts, too. The main differences are that R_{pre} has to coincide with R_{post} and that the guard of the loop has to be considered low in any case.

Again, information flow loop invariants augment functional loop invariants by determines clauses. This time, however, the syntax is slightly different, because R_{pre} has to coincide with R_{post}.

```
loop_determines =
  "determines", ( expressions | "\nothing" ), "\by", "\itself";
expressions = expression, { ",", expression };
```

Without loss of generality we define the semantics of determines clauses for loops with side-effect-free, normally terminating guards only: let R be defined as the concatenation of the guard g of the loop and the expressions behind the **`determines`** keyword. Let further **`\nothing`** and **`\itself`** be defined as in Section 4.3.1. Let ϕ_{inv} be the functional loop invariant defined as usual by maintains clauses. A loop `l` with loop body `body` fulfills a determines clause if and only if ϕ_{inv} is a functional loop invariant for `l` and flow$(\texttt{body}, R, R, \phi_{inv})$ is valid.

4.3.4 Information Flow Class Invariants

It seems to be reasonable to define information flow class invariants in analogy to functional class invariants because in many cases there are information flow properties which have to be fulfilled by all methods of a class. Since such

invariants would be "syntactic sugar" for particular information flow method contracts only, this is left for future work, however.

4.3.5 Naming Views

Sequences of observation expressions can be given names by using ghost or model fields. This has the advantage that such a sequence has to be defined only once and can be reused in different contexts by referring to a meaningful name.

Example. *Consider for instance the view* `customerViewOnBankAccount` *of Figure 4.3. This view can be defined in class* `BankAccount` *by a ghost field with the same name as follows:*

```
/*@ public ghost \seq customerViewOnBankAccount;
  @ public invariant customerViewOnBankAccount ==
  @     \seq(this, balance, id);
  @*/
```

Here, the first line defines a new ghost field of type sequence with the name `customerViewOnBankAccount`*. The second and third line then define with the help of a class invariant that the sequence* `customerViewOnBankAccount` *always equals the sequence* $\langle$`this`, `balance`, `id`$\rangle$.

A larger and more involved example will be considered in the next section.

4.4 Examples

4.4.1 Banking System

Listings 4.2 to 4.4 show an implementation of the class diagram of Figure 4.3 including a complete JML specification. The simplest class in this example is the class `BankAccount`. Its implementation is shown in Listing 4.2.

```
1 public class BankAccount {
2     private int balance;
3     private int id;
4
5     /*@ public ghost \seq customerViewOnBankAccount;
6       @ public invariant customerViewOnBankAccount ==
7       @     \seq(this, balance, id);
```

```
8         @*/
9
10      /*@ normal_behavior
11        @ determines customerViewOnBankAccount \by  \itself;
12        @ assignable \strictly_nothing;
13        @*/
14      public int getId() {
15          return id;
16      }
17
18      /*@ normal_behavior
19        @ determines customerViewOnBankAccount \by  \itself;
20        @ assignable \strictly_nothing;
21        @*/
22      public int getBalance() {
23          return balance;
24      }
25
26      /*@ normal_behavior
27        @ determines customerViewOnBankAccount \by  \itself
28        @            \declassifies amount;
29        @ assignable balance, customerViewOnBankAccount;
30        @*/
31      public void depositMoney(int amount) {
32          this.balance = this.balance - amount;
33          //@ set customerViewOnBankAccount = \seq(this, balance, id);
34      }
35  }
```

Listing 4.2: Example implementation of the class `BankAccount` of Figure 4.3.

The implementation of the three methods is straight forward. The contracts of `getId` and `getBalance` state that (1) no exception will be thrown; (2) the value of the view `customerViewOnBankAccount`—and therefore the value of the sequence $\langle$`this`, `balance`, `id`$\rangle$—in the final state depends at most on its value in the initial state; and (3) no locations are altered at all. Here, item (3) implies (2) and therefore (2) is redundant. Item (2), however, makes the information flow behavior of `getId` and `getBalance` explicit. The view `customerViewOnBankAccount` is defined in lines 5 to 8, as already discussed in Section 4.3.5. The contract of `depositMoney` is only marginally more complex. It states that (1) no exception will be thrown; (2) the value of the view `customerViewOnBankAccount` in the final state depends at most on its value in the initial state and on the (initial) value of `amount`; and (3) at most the locations

balance and customerViewOnBankAccount are modified. In this example, item (2) is not redundant, because balance is modified. The value of amount is in some sense declassified to customerViewOnBankAccount. Therefore, the dependency of customerViewOnBankAccount on amount is stated for presentational purposes by the **\declassifies** keyword. The contract

```
/*@ normal_behavior
  @ determines customerViewOnBankAccount
  @         \by customerViewOnBankAccount, amount;
  @ assignable balance, customerViewOnBankAccount;
  @*/
```

is semantically equivalent to the one of depositMoney. The **set** statement in the body of depositMoney is necessary, because customerViewOnBank-Account is a ghost field. It had not been necessary, if customerViewOnBank-Account had been defined as a model field (which are harder to handle during verification, however). Note that the assignment to customerViewOnBank-Account in the **set** statement is uniquely determined by the invariant: if any other value would be assigned, then the invariant would not hold after the execution of depositMoney any more.

Side Note. *The current KeY-System cannot parse the syntax*

```
\seq(this, balance, id)
```

in **set***-statements, though this would be convenient. Instead, KeY currently supports the following syntax only:*

```
/*@ set customerViewOnBankAccount =
        \seq_concat(\seq_singleton(this),
                    \seq_concat(\seq_singleton(balance),
                                \seq_singleton(id)));
*/
;
```

Here, the **;** *is no typo but has to be added for technical reasons.*

The implementation of the class UserAccount, shown in Listing 4.3, is a bit more involved.

```
1 public class UserAccount {
2     private /*@ spec_public */ int userID;
3     private /*@ spec_public */ char[] password;
4     private /*@ spec_public */ int incorrectLogins;
5     private BankAccount[] bankAccounts;
6
```

```
7     /*@ public ghost \seq employeeViewOnUserAccount;
8       @ public invariant employeeViewOnUserAccount ==
9       @     \seq( this, userID, incorrectLogins, bankAccounts,
10      @           (\seq_def int i; 0; bankAccounts.length;
11      @               bankAccounts[i].customerViewOnBankAccount)
12      @         );
13      @
14      @ public ghost \seq bankCustomerView;
15      @ public invariant bankCustomerView ==
16      @     \seq( this, userID, incorrectLogins, bankAccounts,
17      @           password,
18      @           (\seq_def int i; 0; password.length; password[i]),
19      @           (\seq_def int i; 0; bankAccounts.length;
20      @               bankAccounts[i].customerViewOnBankAccount)
21      @         );
22      @
23      @ public invariant 0 <= incorrectLogins && incorrectLogins <= 3;
24      @
25      @ accessible \inv : this.*, password[*], bankAccounts[*],
26      @                   \infinite_union(
27      @                       int i;
28      @                       (0 <= i && i < bankAccounts.length) ?
29      @                           bankAccounts[i].* : \empty );
30      @*/
31
32    /*@ normal_behavior
33      @ ensures    \result
34      @         == (   0 <= \old(incorrectLogins)
35      @             && \old(incorrectLogins) < 3
36      @             && userID == this.userID
37      @             && password.length == this.password.length
38      @             && (\forall int i; 0 <= i && i < password.length;
39      @                    password[i] == this.password[i]) );
40      @ determines  employeeViewOnUserAccount  \by  \itself
41      @             \declassifies
42      @                     0 <= incorrectLogins
43      @                  && incorrectLogins < 3
44      @                  && userID == this.userID
45      @                  && password.length == this.password.length
46      @                  && (\forall int i;
47      @                         0 <= i && i < password.length;
48      @                         password[i] == this.password[i])
49      @             \declassifies
```

```
50      @                    0 <= incorrectLogins
51      @                 && incorrectLogins < 3
52      @                 && userID == this.userID
53      @                 && (       password.length
54      @                         != this.password.length
55      @                     || (\exists int i;
56      @                            0 <= i && i < password.length;
57      @                            password[i] != this.password[i])
58      @                    );
59      @ determines  bankCustomerView  \by  \itself
60      @             \declassifies
61      @                    0 <= incorrectLogins
62      @                 && incorrectLogins < 3
63      @                 && userID == this.userID
64      @                 && password.length == this.password.length
65      @                 && (\forall int i;
66      @                        0 <= i && i < password.length;
67      @                        password[i] == this.password[i])
68      @             \declassifies
69      @                    0 <= incorrectLogins
70      @                 && incorrectLogins < 3
71      @                 && userID == this.userID
72      @                 && (       password.length
73      @                          != this.password.length
74      @                     || (\exists int i;
75      @                            0 <= i && i < password.length;
76      @                            password[i] != this.password[i])
77      @                    );
78      @ assignable  incorrectLogins, bankCustomerView,
79      @             employeeViewOnUserAccount;
80      @*/
81    public boolean tryLogin(int userID, char[] password) {
82        boolean userIDCorrect = (this.userID == userID);
83        boolean pwdCorrect =
84                    (this.password.length == password.length);
85        /*@ loop_invariant 0 <= i && i <= password.length;
86          @ loop_invariant pwdCorrect ==
87          @      (    password.length == this.password.length
88          @        && (\forall int j; 0 <= j && j < i;
89          @              password[j] == this.password[j])
90          @      );
91          @ assignable \strictly_nothing;
92          @ decreases password.length - i;
```

```
93            @*/
94          for(int i = 0; i < password.length && pwdCorrect; i++) {
95              pwdCorrect = (this.password[i] == password[i]);
96          }
97          boolean incorrectLoginsInRange =
98                  (0 <= incorrectLogins && incorrectLogins < 3);
99
100         if(userIDCorrect && incorrectLoginsInRange) {
101             this.incorrectLogins = pwdCorrect ?
102                                    0 : this.incorrectLogins + 1;
103         }
104         /*@ set bankCustomerView =
105             \seq( this, userID, incorrectLogins, bankAccounts,
106                   password,
107                   (\seq_def int i; 0; password.length; password[i]),
108                   (\seq_def int i; 0; bankAccounts.length;
109                        bankAccounts[i].customerViewOnBankAccount) );
110         */
111         /*@ set employeeViewOnUserAccount =
112             \seq( this, userID, incorrectLogins, bankAccounts,
113                   (\seq_def int i; 0; bankAccounts.length;
114                        bankAccounts[i].customerViewOnBankAccount) );
115         */
116         return userIDCorrect && pwdCorrect && incorrectLoginsInRange;
117     }
118
119     /*@ normal_behavior
120       @ determines  employeeViewOnUserAccount \by \itself;
121       @ determines  bankCustomerView \by  \itself;
122       @ assignable \strictly_nothing;
123       @*/
124     public /*@ nullable */ BankAccount getBankAccount(int number) {
125         if (number < 0 || bankAccounts.length <= number) {
126             return null;
127         }
128         return bankAccounts[number];
129     }
130 }
```

Listing 4.3: Example implementation of the class `UserAccount` of Figure 4.3.

The views `bankCustomerView` and `employeeViewOnUserAccount` are defined in lines 7 to 21 according to Figure 4.3 in the same manner as `customer-`

ViewOnBankAccount in the class BankAccount. Both, bankCustomerView and employeeViewOnUserAccount, comprise for any reachable BankAccount object o the view o.customerViewOnBankAccount. The class UserAccount contains an additional invariant (line 23) which guarantees that the number of consecutive incorrect login attempts is always in the range of zero to three. Once three consecutive incorrect login attempts occur, the account is considered as locked and cannot be accessed any more: the method tryLogin will always return **false**. As specified by the **accessible** clause of the invariant (line 25), the value of the invariant as a whole depends at most on (1) all locations of the UserAccount object itself; (2) all array elements of the password array; (3) all reachable BankAccount objects; and (4) all locations of all reachable BankAccount objects. This framing information is necessary for modular information flow reasoning, as will be discussed in Section 5.3.

The specification of the method getBankAccount is similar to the ones of getId and getBalance of the class BankAccount. The contract of tryLogin is more complicated. As the contracts before, it states that no exception will be thrown. Further, the result of the method is **true** if and only if the passed user id and password match the ones stored in the object and if the number of consecutive incorrect login attempts is smaller than three (and greater or equal zero). The implementation additionally ensures that the number of consecutive incorrect login attempts is incremented each time an incorrect login attempt occurs, until the maximum value of three is reached. The number is reset to zero if it is smaller than three and the correct user id and password are passed. Because the value of incorrectLogins may change, the values of the ghost variables employeeViewOnUserAccount and bankCustomerView have to be adjusted by **set** statements. This is done similar to the **set** statement in the method depositMoney of the class BankAccount. The objective of the information flow specifications of the contract in lines 40 to 77 is to specify that observers of the views employeeViewOnUserAccount and bankCustomerView do not learn anything new by the execution of tryLogin. A careful inspection of the source code, however, shows that observers of those views *can* learn something new, because the value of incorrectLogins is observable in both views. In both cases observers can learn whether or not (1) the number of consecutive incorrect login attempts is in the range of zero to two and the passed userID and password are correct; and whether or not (2) the number of consecutive incorrect login attempts is in the range of zero to two, the passed userID is correct and the passed password is incorrect. Because this information leak is intentional, the corresponding information is declassified to employeeViewOnUserAccount and bankCustomerView. Finally, line 78 specifies that tryLogin modifies at most the values of the locations (**this**, incorrectLogins), (**this**, bankCustomerView) and (**this**, employeeViewOnUserAccount).

Finally, Listing 4.4 shows the implementation of the class `Bank`. The specification is similar to the ones of the classes `UserAccount` and `BankAccount` except of one complication. We have to express that each `bankCustomerView` reachable through the array `userAccounts` depends only on itself (except for some unavoidable declassification). That is, we have to quantify over contracts. This can be achieved with the **`forall`** syntax of JML for contracts (see Raghavan and Leavens [2000]), as used in line 26 in the contract of `login`. The contract following line 26 needs to hold for any value of `anyID`.

Side Note. *Currently KeY does not support the* **`forall`** *syntax of JML for the quantification over contracts. However, instead of line 26 one can use an underspecified ghost (or model) field instead:*

```
//@ public ghost int anyID;
```

Because the ghost field `anyID` *may have any value, this has essentially the same effect as the quantification in line 26.*

```
1  public class Bank {
2
3      private UserAccount[] userAccounts;
4
5      /*@ public model \seq bankEmployeeView;
6        @ public represents bankEmployeeView =
7        @     \seq( userAccounts,
8        @           (\seq_def int i; 0; userAccounts.length;
9        @               userAccounts[i].employeeViewOnUserAccount) );
10       @
11       @ public invariant ( \forall int i;
12       @                        0 <= i && i < userAccounts.length;
13       @                        \invariant_for(userAccounts[i]) );
14       @ public invariant ( \forall int i;
15       @                        0 <= i && i < userAccounts.length;
16       @                        ( \forall int j;
17       @                                 i+1 <= j
18       @                              && j < userAccounts.length;
19       @                             \disjoint(userAccounts[i].*,
20       @                                       userAccounts[j].*)
21       @                         )
22       @                  );
23       @*/
24
25     /*@ normal_behavior
26       @ forall int anyID;
```

```
27      @ requires 0 <= anyID && anyID < userAccounts.length;
28      @ determines  \result
29      @             \by
30      @                 userID,
31      @                 (\seq_def int i; 0; password.length;
32      @                    password[i]),
33      @                 (   0 <= userID
34      @                  && userID < userAccounts.length
35      @                  && 0 <= userAccounts[userID].incorrectLogins
36      @                  && userAccounts[userID].incorrectLogins < 3
37      @                  && userID == userAccounts[userID].userID
38      @                  &&     password.length
39      @                     == userAccounts[userID].password.length
40      @                  && (\forall int i;
41      @                        0 <= i && i < password.length;
42      @                           password[i]
43      @                        == userAccounts[userID].password[i])
44      @                 )
45      @                 ? userAccounts[userID] : null
46      @             \declassifies
47      @                    0 <= userAccounts[userID].incorrectLogins
48      @                  && userAccounts[userID].incorrectLogins < 3
49      @                  && userID == userAccounts[userID].userID
50      @                  &&     password.length
51      @                     == userAccounts[userID].password.length
52      @                  && (\forall int i;
53      @                        0 <= i && i < password.length;
54      @                           password[i]
55      @                        == userAccounts[userID].password[i]);
56      @ determines  bankEmployeeView \by \itself
57      @             \declassifies userID
58      @             \declassifies
59      @                      0 <= userAccounts[userID].incorrectLogins
60      @                  && userAccounts[userID].incorrectLogins < 3
61      @                  && userID == userAccounts[userID].userID
62      @                  &&     password.length
63      @                     == userAccounts[userID].password.length
64      @                  && (\forall int i;
65      @                        0 <= i && i < password.length;
66      @                           password[i]
67      @                        == userAccounts[userID].password[i])
68      @             \declassifies
69      @                      0 <= userAccounts[userID].incorrectLogins
```

```
70      @                   && userAccounts[userID].incorrectLogins < 3
71      @                   && userID == userAccounts[userID].userID
72      @                   && (     password.length
73      @                          != userAccounts[userID].password.length
74      @                      || (\exists int i;
75      @                            0 <= i && i < password.length;
76      @                               password[i]
77      @                            != userAccounts[userID].password[i])
78      @                      );
79      @ determines  userAccounts[anyID].bankCustomerView \by \itself
80      @             \declassifies anyID == userID
81      @             \declassifies
82      @                      anyID == userID
83      @                   && 0 <= userAccounts[userID].incorrectLogins
84      @                   && userAccounts[userID].incorrectLogins < 3
85      @                   && userID == userAccounts[userID].userID
86      @                   &&     password.length
87      @                       == userAccounts[userID].password.length
88      @                   && (\forall int i;
89      @                          0 <= i && i < password.length;
90      @                             password[i]
91      @                           == userAccounts[userID].password[i])
92      @             \declassifies
93      @                      anyID == userID
94      @                   && 0 <= userAccounts[userID].incorrectLogins
95      @                   && userAccounts[userID].incorrectLogins < 3
96      @                   && userID == userAccounts[userID].userID
97      @                   && (     password.length
98      @                          != userAccounts[userID].password.length
99      @                      || (\exists int i;
100     @                            0 <= i && i < password.length;
101     @                               password[i]
102     @                            != userAccounts[userID].password[i])
103     @                      );
104     @*/
105   public /*@ nullable */ UserAccount login(int userID,
106                                            char[] password) {
107       UserAccount result = null;
108       if (0 <= userID && userID < userAccounts.length) {
109           boolean loginSuccessful =
110               userAccounts[userID].tryLogin(userID, password);
111           if (loginSuccessful) {
112               result = userAccounts[userID];
```

```
113             }
114         }
115         return result;
116     }
117
118 }
```

Listing 4.4: Example implementation of the class `Bank` of Figure 4.3.

4.4.2 Loop Invariants

The loop example shown below originates from a tutorial by Christian Hammer and was designed to demonstrate where approximate information flow analysis approaches usually have to give up.

```
//@ normal_behavior
//@ determines low \by \itself;
public void hammer(int secret) {
    int x = 0; int y = 0;

    //@ loop_invariant 0 <= y && y <= 10;
    //@ determines low, y, (y < 10 ? x : 0) \by \itself;
    //@ assignable low;
    //@ decreases 10 - y;
    while (y < 10) {
        print(x);
        if (y == 5) {
            x = secret; y = 9;
        }
        x++; y++;
    }
}

//@ normal_behavior
//@ determines low, x \by \itself;
//@ assignable low;
//@ helper
public void print(int x) { low = x; }
```

It has to be shown that `low` does not depend on `secret`. The method introduces two counters, `x` and `y`. Both are incremented within the loop body. `y` is used in the loop guard whereas `x` is printed in every loop iteration. In the

sixth iteration, when `y` has the value 5, `x` is assigned the secret. This secret is never printed, because `y` is assigned the value 9 in this case and incremented afterwards. Therefore, the loop will not entered again and `secret` will not be leaked to `low`.

The challenge of this example is that at the end of the last iteration of the loop `x` depends on `secret` and therefore approximate analysis usually will reject the program. In the JML extension, however, it is possible to express that `x` does not depend on `secret` as long as `y` is smaller than 10. This enables the verification techniques from Chapter 5 to correctly accept the program. Overall, the determines clause of the loop invariant states that after each loop iteration the values of `low`, `y` and `(y < 10 ? x : 0)` depend at most on initial values of these three expressions.

4.4.3 Block Contracts, Interface Specification and Interactive Programs

Examples for the specification of block contracts can be found in Sections 5.2 and 6.4, where the usefulness of block contracts in the context of information flow verification is discussed. An example for the specification of an interface can be found in the e-voting case-study in Chapter 9.1. This specification is also an example for the specification of interactive programs. Note that in the context of this thesis only deterministic programs are considered.

4.5 Discussion

Different attempts to extend JML for information flow specifications have been presented by Warnier [2006], Haack et al. [2008] and Dufay et al. [2005]. The approaches by Warnier [2006] and Haack et al. [2008] try to specify noninterference in JML by encoding sufficient conditions for noninterference into pre- and postconditions. This is less expressive than the approach of this chapter since the encoded conditions are only sufficient. Furthermore, they do not give hints how to encode declassifications, which is an important feature for the specification of real world programs.

Dufay et al. [2005] on the other hand introduce new JML-keywords which directly define relations between the program variables of two self-composed executions. In particular two keywords to distinguish the variables of the two runs are defined. This is flexible, because general relational properties can be expressed in this way. On the other hand, it seems to be questionable whether a different technique than self-composition (see Chapter 5) can be used to verify

those specifications. The approach presented in this chapter allows the application of other verification techniques, see Chapter 7. Beside the possibility to choose the best suited verification technique for a given program, this has the additional advantage that the specifier does not have to think in a self-composition manner but rather has to figure out only what is allowed to be known by whom. Thus, the specifications presented in this chapter are in this sense more abstract.

The JIF system by Myers [1999a] is another important approach on the specification and verification of information flow properties of Java programs. The core idea of JIF is to annotate variables with security policies and derive a security lattice from those annotations. Given the security lattice, Java programs are analysed with the help of type checking for undesirable information flow. The main advantage of the approach presented in this chapter is its higher expressiveness: on the one hand, information flow properties can be defined for arbitrary JML expressions and not for variables and fields only. On the other hand, the JML extension integrates functional and information flow specifications which makes it easier to use synergy effects of the two specifications. Finally, the declassification construct in JIF, which is used for delimited information release as introduced by Sabelfeld and Myers [2004], can be used within the implementation of a method only. This transgresses against the rule of clear separation of specification and implementation. In contrast, the declassifications presented here are part of the method contract and therefore clearly separated from the implementation.

Banerjee et al. [2008] propose a scheme for the specification (and verification) of expressive declassification policies. The approach is based on a self-defined, non object-oriented programming language, but the authors present some ideas how the technique could be extended to object-oriented languages. The declassification construct presented in this chapter complies with the scheme proposed in Banerjee et al. [2008]. The chapter shows how expressive declassification can be integrated into JML for the specification of sequential Java.

The presented approach can be used to write modular specifications. These specifications fit into the approach of dynamic frames (see for instance Schmitt et al. [2011]) and therefore comply to the principle of information hiding as used in object-oriented programming languages. Furthermore, it allows for fine-grained specifications by the usage of observation expressions instead of variables and fields only. This is in particular useful for the specification of declassification, but also allows for knowledge-based specifications as illustrated in Sections 4.1 and 4.4.

5 Verification of Secure Information Flow by Self-Composition

This chapter is a revised version of Scheben and Schmitt [2014] and parts of Scheben and Schmitt [2012].

Chapters 3 and 4 showed how information flow properties can be formalized and specified. This chapter presents techniques for the deductive verification of those properties.

5.1 Naive Self-Composition

Conditional noninterference can be formulated naturally in Java DL in self-composition style. Firstly, we formalize the notion of indistinguishability of states as given in Definition 2.

Lemma 8. *Let α be a program with local variables $\bar{\mathtt{x}}$, let $\mathcal{D}$ be a Kripke structure, s be a state and β be a variable assignment such that the state s is described by variables $\bar{x}$ and h, that is, $\mathtt{heap}^s = \beta(h)$ and $\bar{\mathtt{x}}^s = \beta(\bar{x})$. Let ϕ be a formula in which at most the program variables $\mathtt{heap}$ and $\bar{\mathtt{x}}$ occur.*

For any state s_2 the evaluation $\mathcal{D}, s_2, \beta \vDash (\{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}\phi)$ equals the evaluation $\mathcal{D}, s, \beta \vDash \phi$.

Proof. By the definition of the semantics of updates, for any state s_2 the evaluation $\mathcal{D}, s_2, \beta \vDash (\{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}\phi)$ equals $\mathcal{D}, s_2^u, \beta \vDash \phi$, where s_2^u is defined like s_2 except that $\mathtt{heap}^{s_2^u} = \beta(h)$ and $\bar{\mathtt{x}}^{s_2^u} = \beta(\bar{x})$. Because $\mathtt{heap}$ and $\bar{\mathtt{x}}$ are the only program variables occurring in ϕ, the evaluation $\mathcal{D}, s_2^u, \beta \vDash \phi$ equals $\mathcal{D}, s, \beta \vDash \phi$. □

Corollary 1. *Let α be a program with local variables $\bar{\mathtt{x}}$, let $\mathcal{D}$ be a Kripke structure, let s_1, s_2 be states and let β be a variable assignment such that the states s_1, s_2 are*

described by variables $\bar{x}_1$, h_1 and $\bar{x}_2$, h_2, respectively, that is, $\texttt{heap}^{s_i} = \beta(h_i)$ *and* $\bar{\texttt{x}}^{s_i} = \beta(\bar{x}_i)$*. Let R be an observation expression over the program variables in α.*

For any state s the formula

$$\begin{aligned} & obsEq(\bar{x}_1, h_1, \bar{x}_2, h_2, R) \\ \equiv\ & \{\texttt{heap} := h_1 \mid\mid \bar{\texttt{x}} := \bar{x}_1\}R = \{\texttt{heap} := h_2 \mid\mid \bar{\texttt{x}} := \bar{x}_2\}R \end{aligned}$$

is valid in Kripke structure $\mathcal{D}$ and state s if and only if agree(R, s_1, s_2) holds.

Proof. Follows directly by Lemma 8 and Definition 2 (Agreement of states). □

With the help of the predicate $obsEq(\bar{x}_1, h_1, \bar{x}_2, h_2, R)$ conditional noninterference can be formalized as follows.

Theorem 9. *Let α be a program with local variables $\bar{\texttt{x}}$ of types $\bar{X}$, let R_1, R_2 be observation expressions over the program variables in α and let ϕ be a formula. Let further h_1, h_1', h_2, h_2' be variables of type Heap and let $\bar{x}_1$, $\bar{x}_1'$, $\bar{x}_2$, $\bar{x}_2'$ be variables of type $\bar{X}$.*

The formula (with suggestive abbreviations $(\!|\ldots|\!)$ as defined below)

$$\begin{aligned} \Psi_{\alpha,\bar{\texttt{x}},R_1,R_2,\phi} \equiv \forall h_1, h_1', h_2, h_2'. \forall \bar{x}_1, \bar{x}_1', \bar{x}_2, \bar{x}_2'. \\ (\!|in\ s_1|\!)(\phi \wedge \langle\alpha\rangle(\!|save\ s_2|\!)) \wedge (\!|in\ s_1'|\!)(\phi \wedge \langle\alpha\rangle(\!|save\ s_2'|\!)) \\ \rightarrow \big(obsEq(\bar{x}_1, h_1, \bar{x}_1', h_1', R_1) \rightarrow obsEq(\bar{x}_2, h_2, \bar{x}_2', h_2', R_2)\big) \end{aligned}$$

is universally valid if and only if flow(α, R_1, R_2, ϕ) holds. Here the $(\!|\ldots|\!)$ abbreviate the formulas:

$$\begin{aligned} (\!|in\ s_i|\!) \equiv\ & \{\texttt{heap} := h_i \mid\mid \bar{\texttt{x}} := \bar{x}_i\} \quad (\!|save\ s_2|\!) \equiv\ (\texttt{heap} = h_2 \wedge \bar{\texttt{x}} = \bar{x}_2) \\ (\!|in\ s_i'|\!) \equiv\ & \{\texttt{heap} := h_i' \mid\mid \bar{\texttt{x}} := \bar{x}_i'\} \quad (\!|save\ s_2'|\!) \equiv\ (\texttt{heap} = h_2' \wedge \bar{\texttt{x}} = \bar{x}_2') \end{aligned}$$

Proof.

"⇒": Assume $\Psi_{\alpha,\bar{\texttt{x}},R_1,R_2,\phi}$. We need to show flow$(\alpha, R_1, R_2, \phi)$. To this end, let s_1, s_1', s_2, s_2' be states such that ϕ^{s_1}, α started in s_1 terminates in s_2, $\phi^{s_1'}$, α started in s_1' terminates in s_2', and agree(R_1, s_1, s_1'). We need to show agree(R_1, s_2, s_2').

We chose the variable assignment β such that $\beta(h_1) = \texttt{heap}^{s_1}$, $\beta(h_1') = \texttt{heap}^{s_1'}$, $\beta(h_2) = \texttt{heap}^{s_2}$, $\beta(h_2') = \texttt{heap}^{s_2'}$, $\beta(\bar{x}_1) = \bar{\texttt{x}}^{s_1}$, $\beta(\bar{x}_1') = \bar{\texttt{x}}^{s_1'}$, $\beta(\bar{x}_2) = \bar{\texttt{x}}^{s_2}$, and $\beta(\bar{x}_2') = \bar{\texttt{x}}^{s_2'}$. Then $\mathcal{D}, s, \beta \models (\!|in\ s_1|\!)(\phi \wedge \langle\alpha\rangle(\!|save\ s_2|\!))$ holds by Lemma 8 and by the definition of $\langle\alpha\rangle$ (for an arbitrary state s), because we have ϕ^{s_1} and α started in s_1 terminates in s_2. Similarly, $\mathcal{D}, s, \beta \models (\!|in\ s_1'|\!)(\phi \wedge \langle\alpha\rangle(\!|save\ s_2'|\!))$

holds, because we have $\phi^{s_1'}$ and α started in s_1' terminates in s_2'. Further, by Corollary 1, $\mathcal{D}, s, \beta \models \mathit{obsEq}(\bar{x}_1, h_1, \bar{x}_1', h_1', R_1)$ if and only if agree(R_1, s_1, s_1'). Therefore, $\mathcal{D}, s, \beta \models \mathit{obsEq}(\bar{x}_2, h_2, \bar{x}_2', h_2', R_2)$ holds. Again by Corollary 1, we conclude agree(R_1, s_2, s_2').

"$\Leftarrow$": Assume flow(α, R_1, R_2, ϕ). We need to show $\mathcal{D}, s, \beta \models \Psi_{\alpha, \bar{\mathtt{x}}, R_1, R_2, \phi}$. To this end, let β be an arbitrary variable assignment and let s_1, s_1', s_2, s_2' be states such that $\beta(h_1) = \mathtt{heap}^{s_1}$, $\beta(h_1') = \mathtt{heap}^{s_1'}$, $\beta(h_2) = \mathtt{heap}^{s_2}$, $\beta(h_2') = \mathtt{heap}^{s_2'}$, $\beta(\bar{x}_1) = \bar{\mathtt{x}}^{s_1}$, $\beta(\bar{x}_1') = \bar{\mathtt{x}}^{s_1'}$, $\beta(\bar{x}_2) = \bar{\mathtt{x}}^{s_2}$, and $\beta(\bar{x}_2') = \bar{\mathtt{x}}^{s_2'}$. We may assume (for an arbitrary state s) $\mathcal{D}, s, \beta \models (\!|\mathit{in}\, s_1|\!)(\phi \wedge \langle\alpha\rangle(\!|\mathit{save}\, s_2|\!))$, $\mathcal{D}, s, \beta \models (\!|\mathit{in}\, s_1'|\!)(\phi \wedge \langle\alpha\rangle(\!|\mathit{save}\, s_2'|\!))$ and $\mathcal{D}, s, \beta \models \mathit{obsEq}(\bar{x}_1, h_1, \bar{x}_1', h_1', R_1)$ and need to show that $\mathcal{D}, s, \beta \models \mathit{obsEq}(\bar{x}_2, h_2, \bar{x}_2', h_2', R_2)$ holds.

By $\mathcal{D}, s, \beta \models (\!|\mathit{in}\, s_1|\!)(\phi \wedge \langle\alpha\rangle(\!|\mathit{save}\, s_2|\!))$ in combination with Lemma 8 and the definition of $\langle\alpha\rangle$ we get ϕ^{s_1} and α started in s_1 terminates in s_2. Similarly, $\mathcal{D}, s, \beta \models (\!|\mathit{in}\, s_1'|\!)(\phi \wedge \langle\alpha\rangle(\!|\mathit{save}\, s_2'|\!))$ implies that $\phi^{s_1'}$ holds and α started in s_1' terminates in s_2'. Further, by Corollary 1, agree(R_1, s_1, s_1') holds because we have $\mathcal{D}, s, \beta \models \mathit{obsEq}(\bar{x}_1, h_1, \bar{x}_1', h_1', R_1)$. Therefore, flow$(\alpha, R_1, R_2, \phi)$ implies agree(R_1, s_2, s_2'). Another appeal to Corollary 1 finally yields $\mathcal{D}, s, \beta \models \mathit{obsEq}(\bar{x}_2, h_2, \bar{x}_2', h_2', R_2)$. □

Though Theorem 9 can already be used to show noninterference with the KeY tool on small examples, the approach in its present form becomes infeasible on larger examples. The next section presents a more sophisticated version of self-composition style reasoning.

5.2 Efficient Self-Composition

Naive self-composition reasoning tends to be inefficient and does not easily lend itself to modular verification. The following example illustrates the efficiency issues of self-composition approaches. Let α be the program

```
l = l + h;
if (h != 0) { l = l - h; }
if (l > 0) { l--; }
```

and let `l` be the only low variable. Then α has no information leak: the value of `l` after line 2 is the same as the initial value of `l`. Thus the value of `l` after line 3 depends only on the initial value of `l`. The execution paths of α are sketched in Figure 5.1(a).

In the classical self-composition approach a copy α' of α is constructed by replacing all program variables by primed copies. In the following, the primed

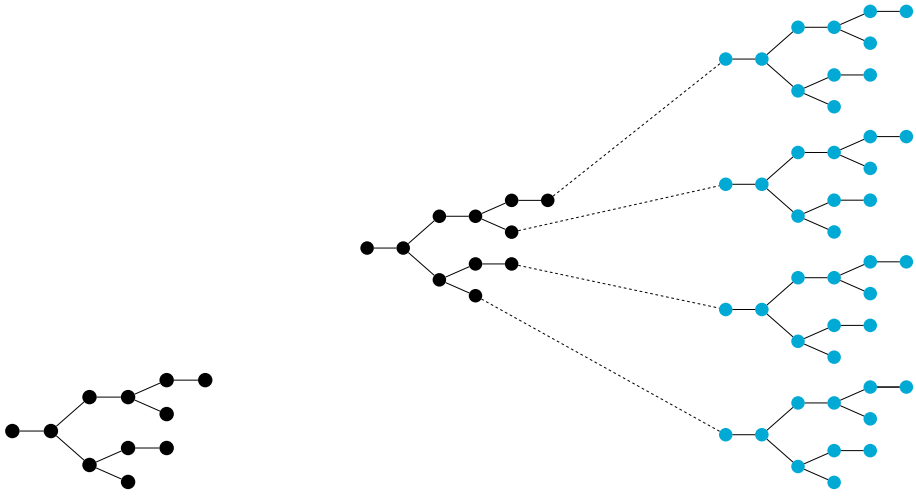


Figure 5.1: Sketch of the execution paths of (a) the original program and (b) the self-composed program.

counterpart of a program variable x is denoted by x′. Accordingly, ϕ' denotes the formula constructed from ϕ by replacing all program variables by their primed counterpart and the term t' denotes the counterpart of t. This leads to the following self-composed program $\alpha;\alpha'$:

```
l = l + h;
if (h != 0) { l = l - h; }
if (l > 0) { l--; }
l′ = l′ + h′;
if (h′ != 0) { l′ = l′ - h′; }
if (l′ > 0) { l′--; }
```

The execution paths of $\alpha;\alpha'$ are sketched in Figure 5.1(b).

h does not interfere with l in α if and only if $\alpha;\alpha'$ started in any state with $\texttt{l} = \texttt{l}'$ terminates in a state where $\texttt{l} = \texttt{l}'$ holds. Hence, in the self-composition approach essentially the outcome of any path through α has to be compared to the outcome of any path through α'. If n is the number of paths through α, this results in $O(n^2)$ comparisons of the low variables. In contrast, specialized information flow calculi, which consider α only once, have to check only the outcome of the n paths through α. This is one reason why self-composition often is considered to be inefficient. The other reason is that the symbolic ex-

ecution of $\alpha;\alpha'$, that is, the reduction of $\alpha;\alpha'$ into syntactic updates and case distinctions (see Beckert et al. [2007b], Section 3.4.5), is at least twice as costly as the symbolic execution of α.

The following section shows that if noninterference is formalized as in Theorem 9, then α needs to be symbolically executed only once. Further, inspired by the compositional reasoning of security type systems and specialized information flow calculi, Section 5.2.2 shows that the number of final symbolic states to be considered can be reduced considerably if α is compositional with respect to information flow. In this case only $O(n)$ final symbolic states have to be considered. Depending on the structure of the program, this number can be reduced further up to $O(\log(n))$.

The following argumentations are based on Dynamic Logic. Readers which are more familiar with weakest precondition calculi might prefer the presentation in Scheben and Schmitt [2014].

5.2.1 Reducing the Cost of the Symbolic Execution

We tackle the first problem, reducing the cost of the symbolic execution, by showing that it is possible to prove noninterference in self-composition style with the help of only one symbolic execution of α.

Let $\texttt{heap}$ and $\bar{\texttt{x}}$ be the program variables of α and let h_1, $\bar{x}_1$, h_2, $\bar{x}_2$, h_1', $\bar{x}_1'$, h_2' and $\bar{x}_2'$ be variables of appropriate type. We aim at finding formulas ψ and ψ' (without modalities) with only one symbolic execution of α which replace $\{\texttt{heap} := h_1 \parallel \bar{\texttt{x}} := \bar{x}_1\}\langle\alpha\rangle(\texttt{heap} = h_2 \land \bar{\texttt{x}} = \bar{x}_2)$ and $\{\texttt{heap} := h_1' \parallel \bar{\texttt{x}} := \bar{x}_1'\}\langle\alpha\rangle(\texttt{heap} = h_2' \land \bar{\texttt{x}} = \bar{x}_2')$ in Theorem 9.

Let $\mathcal{D}$ be a Kripke structure, s a state and β a variable assignment. The main step is finding a formula ψ—by symbolic execution of α—such that $\mathcal{D}, s, \beta \vDash \{\texttt{heap} := h_1 \parallel \bar{\texttt{x}} := \bar{x}_1\}\langle\alpha\rangle(\texttt{heap} = h_2 \land \bar{\texttt{x}} = \bar{x}_2)$ implies $\mathcal{D}_{ext}, s, \beta \vDash \psi$ for an extension $\mathcal{D}_{ext}$ of $\mathcal{D}$ by new Skolem symbols. (We need to consider extensions of $\mathcal{D}$, because the symbolic execution of α might introduce new Skolem symbols.) Note that the application of the Java DL calculus—which contains all necessary rules for the symbolic execution of α—on $\{\texttt{heap} := h_1 \parallel \bar{\texttt{x}} := \bar{x}_1\}\langle\alpha\rangle(\texttt{heap} = h_2 \land \bar{\texttt{x}} = \bar{x}_2)$ does not deliver the desired implication: it approximates $\{\texttt{heap} := h_1 \parallel \bar{\texttt{x}} := \bar{x}_1\}\langle\alpha\rangle(\texttt{heap} = h_2 \land \bar{\texttt{x}} = \bar{x}_2)$ in the wrong direction. We have to take an indirection.

Intuitively, the formula $\{\texttt{heap} := h_1 \parallel \bar{\texttt{x}} := \bar{x}_1\}\langle\alpha\rangle(\texttt{heap} = h_2 \land \bar{\texttt{x}} = \bar{x}_2)$ is valid in $(\mathcal{D}, s, \beta)$ if α started in state $s_1 : \texttt{heap} \mapsto h_1^\beta, \bar{\texttt{x}} \mapsto \bar{x}_1^\beta$ terminates in state $s_2 : \texttt{heap} \mapsto h_2^\beta, \bar{\texttt{x}} \mapsto \bar{x}_2^\beta$ (see Theorem 9). We calculate a formula ψ_{not} which is *at most* true if α started in $s_1 : \texttt{heap} \mapsto h_1^\beta, \bar{\texttt{x}} \mapsto \bar{x}_1^\beta$ does *not* terminate

in s_2 : heap $\mapsto h_2^\beta, \bar{\mathtt{x}} \mapsto \bar{x}_2^\beta$. Then $\psi = \neg\psi_{not}$ is *at least* true if α started in s_1 terminates in s_2. We obtain ψ_{not} by symbolic execution of $\{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}[\alpha](\mathtt{heap} \neq h_2 \vee \bar{\mathtt{x}} \neq \bar{x}_2)$: application of the Java DL calculus on the sequent $\Longrightarrow \{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}[\alpha](\mathtt{heap} \neq h_2 \vee \bar{\mathtt{x}} \neq \bar{x}_2)$ results in a set of sequents F_{seq}, where each $f_{seq} \in F_{seq}$ does not contain modalities any more. Let F be the set of meaning formulas for F_{seq}. We set $\psi_{not} = \bigwedge_{f \in F} f$.

Because the Java DL calculus is sound, the universal validity of the premisses of a rule implies the universal validity of its conclusion. Java DL rules fulfill even a slightly stronger property: rules which do not introduce new Skolem symbols have the property that the conclusion of the rule is valid in $(\mathcal{D}, s, \beta)$ if the premisses are valid in $(\mathcal{D}, s, \beta)$. In case new Skolem symbols are introduced, as for instance in the rule for object creation (see Weiß [2011] or—for a slightly modified version—Figure 7.6), the conclusion is valid in $(\mathcal{D}, s, \beta)$ if the premisses are valid in all extensions $(\mathcal{D}_{ext}, s, \beta)$ of $(\mathcal{D}, s, \beta)$ by interpretations of the new Skolem symbols. As a consequence we get that $\mathcal{D}, s, \beta \not\models \{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}[\alpha](\mathtt{heap} \neq h_2 \vee \bar{\mathtt{x}} \neq \bar{x}_2)$ implies $\mathcal{D}_{ext}, s, \beta \not\models \psi_{not}$ for an extension $\mathcal{D}_{ext}$ of $\mathcal{D}$ by new Skolem symbols. By the duality of box and diamond $\mathcal{D}, s, \beta \not\models \{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}[\alpha](\mathtt{heap} \neq h_2 \vee \bar{\mathtt{x}} \neq \bar{x}_2)$ is equivalent to $\mathcal{D}, s, \beta \models \{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} = h_2 \wedge \bar{\mathtt{x}} = \bar{x}_2)$. Further, ψ is defined as $\neg\psi_{not}$. Hence, $\mathcal{D}, s, \beta \models \{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} = h_2 \wedge \bar{\mathtt{x}} = \bar{x}_2)$ implies $\mathcal{D}_{ext}, s, \beta \models \psi$ for an extension $\mathcal{D}_{ext}$ of $\mathcal{D}$ by new Skolem symbols, as desired.

Given ψ, we observe that we obtain a formula ψ' such that $\mathcal{D}, s, \beta \models \{\mathtt{heap} := h_1' \mid\mid \bar{\mathtt{x}} := \bar{x}_1'\}\langle\alpha\rangle(\mathtt{heap} = h_2' \wedge \bar{\mathtt{x}} = \bar{x}_2')$ implies $\mathcal{D}, s, \beta \models \psi'$ by a simple renaming of the variables h_1, $\bar{x}_1$, h_2, $\bar{x}_2$ to h_1', $\bar{x}_1'$, h_2', $\bar{x}_2'$ and by the renaming of the new Skolem symbols $\bar{c}$ to new primed Skolem symbols $\bar{c}'$. The thus obtained formulas ψ and ψ' can be used to replace $\{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} = h_2 \wedge \bar{\mathtt{x}} = \bar{x}_2)$ and $\{\mathtt{heap} := h_1' \mid\mid \bar{\mathtt{x}} := \bar{x}_1'\}\langle\alpha\rangle(\mathtt{heap} = h_2' \wedge \bar{\mathtt{x}} = \bar{x}_2')$ in Theorem 9. Their calculation involves only one symbolic execution.

5.2.2 Reducing the Number of Comparisons

The second problem, reducing the number of comparisons, can be tackled with the help of compositional reasoning, if the structure of the program allows for it. Reconsider the initial example:

```
l = l + h;
if (h != 0) { l = l - h; }
if (l > 0) { l--; }
```

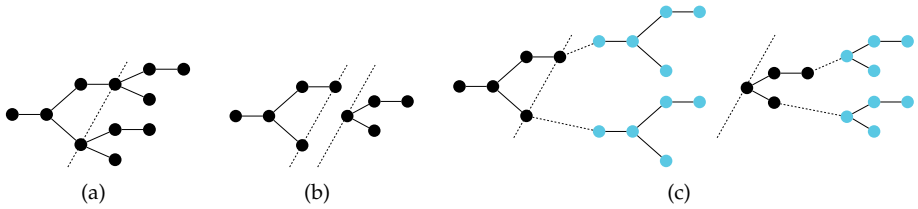


Figure 5.2: Reducing the verification overhead by compositional reasoning.

As discussed above, the first part, lines 1 and 2, and the second part, line 3, are noninterfering on their own. Therefore, by Lemma 5, the complete program is noninterfering. As illustrated in Figure 5.2, checking the two parts independently from each other results in less verification effort: the execution paths of each self-composed part on its own contains only four paths. Thus, altogether only eight comparisons have to be made to prove noninterference of the complete program. Checking the complete program at once would require (about) 12 comparisons.[1] We summarize the above observation in the following lemma.

Lemma 10. *Let α be a program with m branching statements.*

If α can be divided into m noninterfering blocks with at most one branching statement per block, then noninterference of α can be shown with the help of self-composition with $3m$ comparisons.

Because a program with m branching statements has at least $n = m + 1$ paths, Lemma 10 shows that the verification effort of self-composition approaches can be reduced from $O(n^2)$ comparisons to $O(n)$, if the program under consideration is compositional with respect to information flow. In the best case, a program with m branching statements has $\Omega(2^m)$ paths. In this case the verification effort reduces to $O(\log(n))$ comparisons, if the program under consideration is compositional with respect to information flow.

Unfortunately, the separation of the program into blocks is not always as simple as in the example above. Consider for instance the following program:

```
if (l > 0) { if (l % 2 == 1) { l--; } }
```

The program can be divided into blocks

[1] By symmetry the number of comparisons can be reduced further in both cases: in the first case $2 \cdot (2 + 1) = 6$ comparisons are sufficient, in the second case $4 + 3 + 2 + 1 = 10$ comparisons are enough.

$b_1 =$ `if (l % 2 == 1) { l--; }`

and

$b_2 =$ `if (l > 0) {` b_1 `}`.

To conclude that b_2 is noninterfering, it is necessary to use the fact that b_1 is noninterfering in the proof of b_2. Unfortunately, the self-composition approach does not easily lend itself to such compositional/modular verification, as shown in the next section.

5.3 Modular Self-Composition

If program α calls a block b, one (sometimes) does not want to look at its code but rather use a software contract for b, a contract that had previously been established by looking only at the code of b. This kind of modularization can also be applied to methods instead of blocks and is essential for the scalability of all deductive software verification approaches. With self-composition b is not only called in α, but b' is called in α'. This poses the technical problem of somehow synchronizing the calls of b and b' for contract application. This section shows how software contracts can be applied in self-composition proofs. An important feature of the approach is the seamless integration of information flow and functional reasoning allowing us to take advantage of the precision of functional contracts also for information flow verification, if necessary.

In the context of functional verification, modularity is achieved with the help of method contracts. We want to extend this approach to the verification of information flow properties. Firstly, we define *information flow contracts*.

Definition 11 (Information Flow Contract)**.** *An information flow contract (in short "flow contract") to a block (or method)* $\mathtt{b}$ *with local variables* $\bar{\mathtt{x}} := (\mathtt{x}_1, \ldots, \mathtt{x}_n)$ *of types* $\bar{A} := (A_1, \ldots, A_n)$ *is a tuple* $\mathcal{C}_{\mathtt{b},\bar{\mathtt{x}}} = (Pre, R_1, R_2)$*, where (1)* Pre *is a formula which represents a precondition and (2)* R_1*,* R_2 *are observation expressions which represent the low expressions in the pre- and post-state.*

A flow contract $\mathcal{C}_{\mathtt{b},\bar{\mathtt{x}}} = (Pre, R_1, R_2)$ *is valid if and only if for all states* s *the predicate* $\mathit{flow}(\mathtt{b}, R_1, R_2, Pre)$ *is valid.*

The difficulty in the application of flow contracts arises from the fact that flow contracts refer to two invocations of a block $\mathtt{b}$ in different contexts.

Example. *Consider*

```
if (l>0) { l++; if (l%2 == 1) {l--;} }
```

again, with blocks $b_1 =$ `if (l%2==1) {l--;}` *and* $b_2 =$ `if (l>0) {l++;` b_1`}`. *Let* $\mathcal{C}_{b_1,\bar{\mathtt{x}}} = \mathcal{C}_{b_2,\bar{\mathtt{x}}} = (true, \mathtt{l}, \mathtt{l})$ *be flow contracts for* b_1 *and* b_2. *To prove* $\mathcal{C}_{b_2,\bar{\mathtt{x}}}$ *by classical self-composition,*

$$\mathtt{l} = \mathtt{l}' \rightarrow (\langle \mathtt{if\ (l>0)\ \{l++;}\ b_1\mathtt{\};\ if\ (l'>0)\ \{l'++;}\ b_1'\mathtt{\}}\rangle \mathtt{l} = \mathtt{l}')$$

has to be shown. (For presentational purposes, we ignore the heap in this example.) Symbolic execution of the program, as far as possible, yields:

$$\dfrac{\dfrac{\mathtt{l} = \mathtt{l}',\ \mathtt{l} > 0 \Longrightarrow \{\mathtt{l} := \mathtt{l}+1\}\ \langle b_1;\ \mathtt{if\ (l'>0)\ \{}\ \mathtt{l'++;}\ b_1'\ \mathtt{\}}\rangle \mathtt{l} = \mathtt{l}'}{\vdots\ \textit{symbolic execution}}\ (5.1)}{\Longrightarrow \mathtt{l} = \mathtt{l}' \rightarrow (\langle \mathtt{if\ (l>0)\ \{l++;}\ b_1\mathtt{\};\ if\ (l'>0)\ \{l'++;}\ b_1'\mathtt{\}}\rangle \mathtt{l} = \mathtt{l}')}$$

$$\text{apply-Equality + close}\ \dfrac{*}{\mathtt{l} = \mathtt{l}',\ \mathtt{l}' > 0 \Longrightarrow \mathtt{l} > 0,\ \{\mathtt{l}' := \mathtt{l}' + 1\}\ \langle b_1'\rangle \mathtt{l} = \mathtt{l}'}$$

$$\text{close}\ \dfrac{*}{\mathtt{l} = \mathtt{l}' \Longrightarrow \mathtt{l} > 0,\ \mathtt{l}' > 0,\ \mathtt{l} = \mathtt{l}'}$$

To close branch (5.1), $\mathcal{C}_{b_1,\bar{\mathtt{x}}}$ *needs to be used—but it is not obvious how this can be done, because* $\mathcal{C}_{b_1,\bar{\mathtt{x}}}$ *refers to the invocation of* b_1 *and the invocation of* b_1' *at the same time. A similar problem occurs if* $\mathcal{C}_{b_2,\bar{\mathtt{x}}}$ *is proved with the help of the optimizations discussed in Section 5.2.*

The main idea of the solution is a coordinated delay of the application of flow contracts. The solution is compatible with the optimizations from Section 5.2 and additionally allows the combination of flow contracts with functional contracts.

Let `b` be a block with the functional contract $\mathcal{F}_{\mathtt{b},\bar{\mathtt{x}}} = (Pre, Post, Rep)$ consisting of: (1) a formula Pre representing the precondition; (2) a formula $Post$ representing the postcondition; and (3) a term Rep representing the assignable clause for `b`. In functional verification, block contracts are applied by the rule "useBlockContract", introduced by Wacker [2012]. The rule is an adaption of

the rule "useMethodContract" from Weiß [2011] for blocks. For presentational purposes we consider a simplified version of the rule only:

$$\text{useBlockContract}\ \frac{\begin{array}{ll} \mathit{pre} & \Gamma \Longrightarrow \{u\}\mathit{Pre}, \Delta \\ \mathit{post} & \Gamma \Longrightarrow \{u; u_{anon}\}(\mathit{Post} \rightarrow [\pi\ \omega]\phi), \Delta \end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \mathtt{b};\ \omega]\phi, \Delta}$$

Here, u is an arbitrary update; $u_{anon} = (\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathit{Rep}, h), \bar{\mathtt{x}} := \bar{x}')$ is an anonymising update setting the locations of Rep (which might be modified by $\mathtt{b}$) and the local variables which might be modified to unknown values; h of type Heap and $\bar{x}'$ of appropriate types are fresh symbols. We require Pre to entail equations $\mathtt{heap}_{pre} = \mathtt{heap}$ and $\bar{\mathtt{x}}_{pre} = \bar{\mathtt{x}}$ which store the values of the program variables of the initial state in program variables $\mathtt{heap}_{pre}$ and $\bar{\mathtt{x}}_{pre}$ such that the initial values can be referred to in the post-condition. Additionally, we require that Pre and Post entail a formula which expresses that the heap is wellformed. For the sake of simplicity we do not handle exceptions here.

The plan is to use an extended version of the rule "useBlockContract" during symbolic execution—in many cases for the trivial functional contract $\mathcal{F}_{b,\bar{\mathtt{x}}} = (\mathit{true}, \mathit{true}, \mathit{allLocs})$—which adds some extra information to the sequent allowing a delayed application of information flow contracts. The extra information is encapsulated in a new two-state predicate $C_{\mathtt{b}}(\bar{x}, h, \bar{x}', h')$ with the intended meaning that $\mathtt{b}$ started in state $s_1 : \mathtt{heap} \mapsto h, \bar{\mathtt{x}} \mapsto \bar{x}$ terminates in state $s_2 : \mathtt{heap} \mapsto h', \bar{\mathtt{x}} \mapsto \bar{x}'$. This predicate can be integrated into the rule "useBlockContract" as follows:

$$\text{useBlockContract2}\ \frac{\begin{array}{ll} \mathit{pre} & \Gamma \Longrightarrow \{u\}\mathit{Pre}, \Delta \\ \mathit{post} & \Gamma, \{u\}C_{\mathtt{b}}(\bar{\mathtt{x}}, \mathtt{heap}, \bar{x}', h'), \{u; u_{anon}\}(\mathtt{heap} = h' \wedge \bar{\mathtt{x}} = \bar{x}') \\ & \qquad \Longrightarrow \{u; u_{anon}\}(\mathit{Post} \rightarrow [\pi\ \omega]\phi), \Delta \end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \mathtt{b};\ \omega]\phi, \Delta}$$

where h' and $\bar{x}'$ are fresh function symbols. By Lemma 12 below, the rule "useBlockContract2" is sound. The introduction of $C_{\mathtt{b}}(\bar{x}, h, \bar{x}', h')$ to the post branch allows us to store the initial and the final state of $\mathtt{b}$ for a delayed application of information flow contracts: Theorem 13 shows that two predicates $C_{\mathtt{b}}(\bar{x}_1, h_1, \bar{x}_2, h_2)$ and $C_{\mathtt{b}}(\bar{x}'_1, h'_1, \bar{x}'_2, h'_2)$ appearing on the antecedent of a sequent can be approximated by an instantiation of a flow contract $\mathcal{C}_{\mathtt{b},\bar{\mathtt{x}}} = (\mathit{Pre}, R_1, R_2)$ for $\mathtt{b}$ by

$$\begin{array}{l} \{\mathtt{heap} := h_1 \parallel \bar{\mathtt{x}} := \bar{x}_1\}\mathit{Pre} \wedge \{\mathtt{heap} := h'_1 \parallel \bar{\mathtt{x}} := \bar{x}'_1\}\mathit{Pre} \\ \rightarrow \big(\mathit{obsEq}(\bar{x}_1, h_1, \bar{x}'_1, h'_1, R_1) \rightarrow \mathit{obsEq}(\bar{x}_2, h_2, \bar{x}'_2, h'_2, R_2)\big) \end{array}$$

This approximation is applied by the rule "useFlowContract":

$$\text{useFlowContract}\quad \frac{\begin{array}{l}\Gamma, C_{\mathtt{b}}(\bar{x}_1, h_1, \bar{x}_2, h_2), C_{\mathtt{b}}(\bar{x}'_1, h'_1, \bar{x}'_2, h'_2),\\ \quad \{\mathtt{heap} := h_1 \parallel \bar{\mathtt{x}} := \bar{x}_1\}\mathit{Pre} \wedge \{\mathtt{heap} := h'_1 \parallel \bar{\mathtt{x}} := \bar{x}'_1\}\mathit{Pre}\\ \quad \rightarrow \big(\mathit{obsEq}(\bar{x}_1, h_1, \bar{x}'_1, h'_1, R_1) \rightarrow \mathit{obsEq}(\bar{x}_2, h_2, \bar{x}'_2, h'_2, R_2)\big)\\ \Longrightarrow \Delta\end{array}}{\Gamma, C_{\mathtt{b}}(\bar{x}_1, h_1, \bar{x}_2, h_2), C_{\mathtt{b}}(\bar{x}'_1, h'_1, \bar{x}'_2, h'_2) \Longrightarrow \Delta}$$

Example. *Let $\mathcal{F}_{b_1,\bar{\mathtt{x}}} = (\mathit{true}, \mathit{true}, \mathit{allLocs})$ be the trivial functional contract for b_1. Applied on the example from above, (5.1) can be simplified as shown in Figure 5.3. For presentational purposes all heap symbols have been removed from the example. Therefore, C_{b_1} takes only two parameters and obsEq only three. Adding the heap results in essentially the same proof but with more complex formulas.*

The proof uses the following abbreviations of rule names:

Abbreviation	*Full name*	*Abbreviation*	*Full name*
uBC2	useBlockContract2	eq	applyEquality
uFC	useFlowContract	if	conditional
obsEq	*replaces $\mathit{obsEq}(\cdot)$ by its definition (Corollary 1)*	simp	*combination of all update simplification rules*
$++$	plusPlus	close	close
eq + simp	*repeated application of the rules* eq *and* simp		

Firstly, the symbolic execution is continued by the rule "useBlockContract2" and (after several simplifications) by the rule "conditional". The conditional rule splits the proof into two branches. The right branch, which represents the case that the condition $\mathtt{l}' > 0$ evaluates to false, can be closed after further simplifications and the application of equalities. On the other branch, the remaining program is executed symbolically by the rule "plusPlus" and another application of "useBlockContract2", now on the block b'_1. After some further simplifications, we are in the position to apply the flow contract for b_1: the antecedent of the sequent contains the two predicates $C_{b_1}(\mathtt{l}+1, \ell)$ and $C_{b_1}(\mathtt{l}'+1, \ell')$ on which the rule "useFlowContract" can be applied. With the help of the guarantees from the flow contract for b_1, the proof closes after some final simplifications.

Formally, $C_{\mathtt{b}}(\bar{x}, h, \bar{x}', h')$ is valid in Kripke structure $\mathcal{D}$ and state s if and only if

$$\{\bar{\mathtt{x}} := \bar{x} \parallel \mathtt{heap} := h\}\langle \mathtt{b} \rangle (\mathtt{heap} = h' \wedge \bar{\mathtt{x}} = \bar{x}')$$

is valid in $(\mathcal{D}, s)$. Note that the usage of the rule "useBlockContract2" during symbolic execution allows the application of arbitrary functional contracts in addition to flow contracts. This allows for taking advantage of the precision of

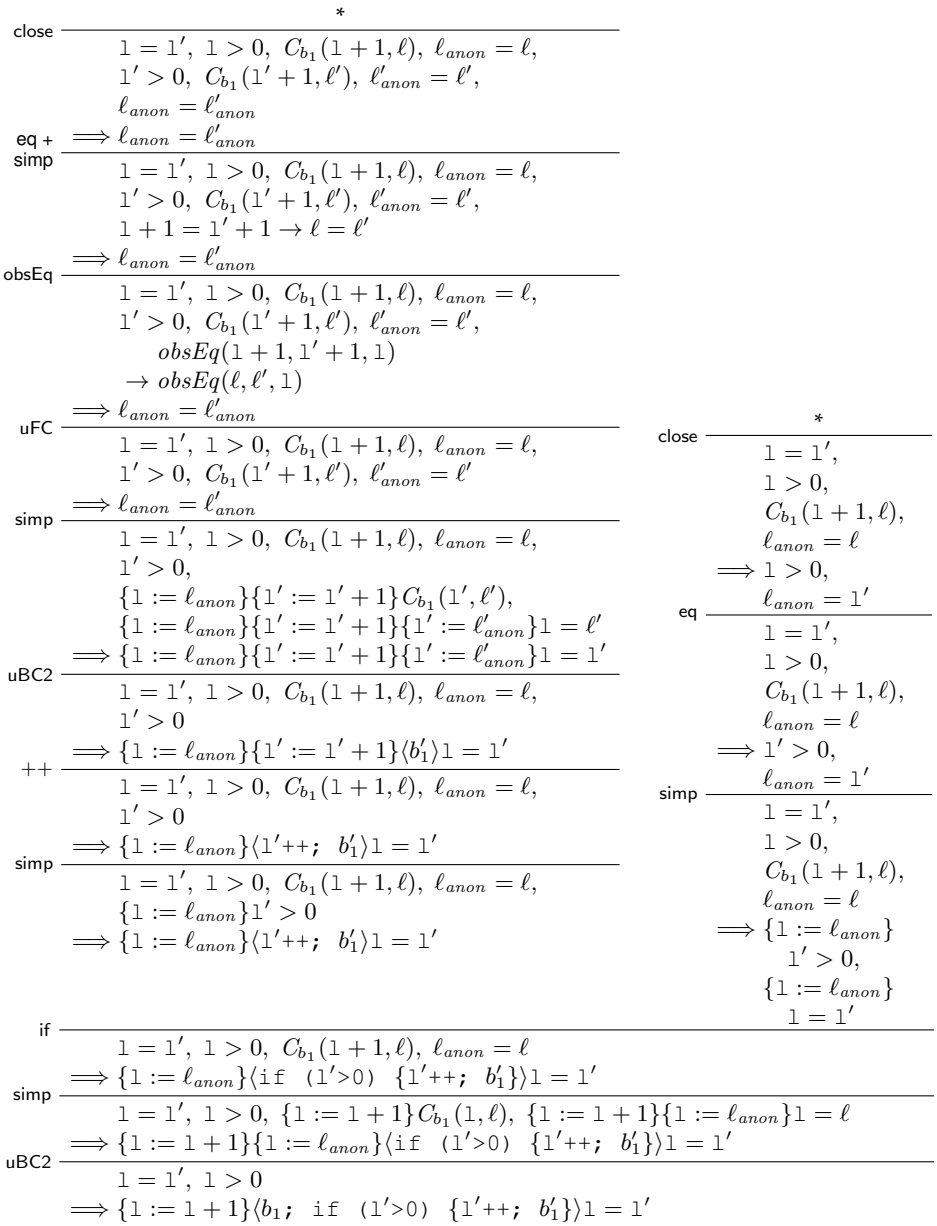


Figure 5.3: Proof tree to the example from page 61.

functional contracts within information flow proofs, if necessary. The default, however, is using the trivial functional contract $\mathcal{F}_{b,\bar{\mathtt{x}}} = (true, true, allLocs)$ as in the presented example.

The remainder of this section shows soundness of the above approach.

Lemma 12. *Let* $\mathtt{b}$ *be a block which fulfills the functional block contract* $\mathcal{F}_{\mathtt{b},\bar{\mathtt{x}}} = (Pre, Post, Rep)$*. Then the rule "useBlockContract2" is sound.*

Proof. Because the rule "useBlockContract" is sound (see Wacker [2012]) it suffices to show that the premiss *post* of "useBlockContract2" is valid in a Kripke structure $\mathcal{D}$ and a state s if and only if the premiss *post* of "useBlockContract" is valid in $(\mathcal{D}, s)$.

Let $\mathcal{D}$ be a Kripke structure and let s be a state. If the premiss *post* of "useBlockContract2" is valid in $(\mathcal{D}, s)$, then by simple propositional logic also the premiss *post* of "useBlockContract" is valid in $(\mathcal{D}, s)$. Thus, we assume that the premiss *post* of "useBlockContract" is valid in $(\mathcal{D}, s)$ and set out to show that the premiss *post* of "useBlockContract2" is true in $(\mathcal{D}, s)$. We assume $\bigwedge \Gamma \nrightarrow \bigvee \Delta$ and $\{u; u_{anon}\}Post$ are true in $(\mathcal{D}, s)$ with the aim to show that $\{u; u_{anon}\}[\pi\ \omega]\phi$ is also true in $(\mathcal{D}, s)$. Since the new constant symbols h' and $\bar{x}'$ do not occur in $\{u; u_{anon}\}Post$ we find a Kripke structure $\mathcal{D}'$ that differs from $\mathcal{D}$ only in the interpretation of these symbols such that in $(\mathcal{D}', s)$ both $\{u; u_{anon}\}Post$ and $\{u\}C_{\mathtt{b}}(\bar{\mathtt{x}}, \mathtt{heap}, \bar{x}', h') \wedge \{u; u_{anon}\}(\mathtt{heap} = h' \wedge \bar{\mathtt{x}} = \bar{x}')$ are true. This may be achieved by choosing the structure $\mathcal{D}'$ such that the state s_2 presented by $((h')^{\mathcal{D}',s}, (\bar{x}')^{\mathcal{D}',s})$ is the final state of $\mathtt{b}$ when started in the state s_1 presented by $((\{u\}\mathtt{heap})^{\mathcal{D},s}, (\{u\}\bar{\mathtt{x}})^{\mathcal{D},s})$. By validity of the premiss *post* of "useBlockContract2" we obtain that $\{u; u_{anon}\}[\pi\ \omega]\phi$ is true in $(\mathcal{D}', s)$. Since $\{u; u_{anon}\}[\pi\ \omega]\phi$ does likewise not contain the new symbols it is also true in the orignal Kripke structure $\mathcal{D}$ and state s. □

Theorem 13. *Let* $\mathtt{b}$ *be a block fulfilling the flow contract* $\mathcal{C}_{\mathtt{b},\bar{\mathtt{x}}} = (Pre, R_1, R_2)$*. Further, let* $\mathcal{D}$ *be a Kripke structure,* s *be a state and* β *be a variable assignment.*

$\mathcal{D}, s, \beta \vDash C_{\mathtt{b}}(\bar{x}_a, h_a, \bar{x}'_a, h'_a)$ *and* $\mathcal{D}, s, \beta \vDash C_{\mathtt{b}}(\bar{x}_b, h_b, \bar{x}'_b, h'_b)$ *imply*

$$\begin{aligned} \mathcal{D}, s, \beta \vDash \quad & \{\mathtt{heap} := h_a \mid\mid \bar{\mathtt{x}} := \bar{x}_a\}Pre \wedge \{\mathtt{heap} := h_b \mid\mid \bar{\mathtt{x}} := \bar{x}_b\}Pre \\ & \rightarrow \big(obsEq(\bar{x}_a, h_a, \bar{x}'_a, h'_a, R_1) \rightarrow obsEq(\bar{x}_b, h_b, \bar{x}'_b, h'_b, R_2)\big) \end{aligned} \tag{5.2}$$

Proof. We assume that the left-hand side of (5.2) is true in $(\mathcal{D}, s, \beta)$, that is, $\mathcal{D}, s, \beta \vDash \{\mathtt{heap} := h_a \mid\mid \bar{\mathtt{x}} := \bar{x}_a\}Pre$ and $\mathcal{D}, s, \beta \vDash \{\mathtt{heap} := h_b \mid\mid \bar{\mathtt{x}} := \bar{x}_b\}Pre$. By assumption $\mathcal{D}, s, \beta \vDash C_{\mathtt{b}}(\bar{x}_a, h_a, \bar{x}'_a, h'_a)$ and $\mathcal{D}, s, \beta \vDash C_{\mathtt{b}}(\bar{x}_b, h_b, \bar{x}'_b, h'_b)$ are true, which by definition says $\mathcal{D}, s, \beta \vDash \{\bar{\mathtt{x}} := \bar{x}_a \mid\mid \mathtt{heap} := h_a\}\langle\mathtt{b}\rangle(\mathtt{heap} = h'_a \wedge \bar{\mathtt{x}} = \bar{x}'_a)$ and $\mathcal{D}, s, \beta \vDash \{\bar{\mathtt{x}} := \bar{x}_b \mid\mid \mathtt{heap} := h_b\}\langle\mathtt{b}\rangle(\mathtt{heap} = h'_b \wedge \bar{\mathtt{x}} = \bar{x}'_b)$. The

assumption that the flow contract $\mathcal{C}_{\mathtt{b},\bar{\mathtt{x}}} = (Pre, R_1, R_2)$ is fulfilled implies via Definition 11 and Theorem 9 that $\mathcal{D}, s, \beta \vDash \Psi_{\mathtt{b},\bar{\mathtt{x}},R_1,R_2,Pre}$ is true. By Definition, $\Psi_{\mathtt{b},\bar{\mathtt{x}},R_1,R_2,Pre}$ equals

$$\begin{aligned}\forall h_1, h_1', h_2, h_2'.\forall \bar{x}_1, \bar{x}_1', \bar{x}_2, \bar{x}_2'.\\ (\circledast in\ s_1 \circledast)(Pre \wedge \langle \mathtt{b} \rangle (\circledast save\ s_2 \circledast)) \wedge (\circledast in\ s_1' \circledast)(Pre \wedge \langle \mathtt{b} \rangle (\circledast save\ s_2' \circledast)) \\ \rightarrow \big(obsEq(\bar{x}_1, h_1, \bar{x}_1', h_1', R_1) \rightarrow obsEq(\bar{x}_2, h_2, \bar{x}_2', h_2', R_2)\big)\end{aligned} \tag{5.3}$$

with abbreviations

$$\begin{aligned}(\circledast in\ s_i \circledast) &\equiv \{\mathtt{heap} := h_i \mid\mid \bar{\mathtt{x}} := \bar{x}_i\} \quad (\circledast save\ s_2 \circledast) \equiv (\mathtt{heap} = h_2 \wedge \bar{\mathtt{x}} = \bar{x}_2)\\ (\circledast in\ s_i' \circledast) &\equiv \{\mathtt{heap} := h_i' \mid\mid \bar{\mathtt{x}} := \bar{x}_i'\} \quad (\circledast save\ s_2' \circledast) \equiv (\mathtt{heap} = h_2' \wedge \bar{\mathtt{x}} = \bar{x}_2')\ .\end{aligned}$$

Let β' be defined as $\beta'(y) = ((y)[\bar{x}_1/\bar{x}_a, h_1/h_a, \bar{x}_2/\bar{x}_b, h_2/h_b])^{\mathcal{D},s,\beta}$ for all variables y. Hence, we have

$$\bar{x}_1^{\mathcal{D},s,\beta'} = \bar{x}_a^{\mathcal{D},s,\beta}, h_1^{\mathcal{D},s,\beta'} = h_a^{\mathcal{D},s,\beta}, \bar{x}_2^{\mathcal{D},s,\beta'} = \bar{x}_b^{\mathcal{D},s,\beta} \text{ and } h_2^{\mathcal{D},s,\beta'} = h_b^{\mathcal{D},s,\beta}. \tag{5.4}$$

Because Equation (5.3) is valid in $(\mathcal{D}, s, \beta)$,

$$\begin{aligned}(\circledast in\ s_1 \circledast)(Pre \wedge \langle \mathtt{b} \rangle (\circledast save\ s_2 \circledast)) \wedge (\circledast in\ s_1' \circledast)(Pre \wedge \langle \mathtt{b} \rangle (\circledast save\ s_2' \circledast))\\ \rightarrow \big(obsEq(\bar{x}_1, h_1, \bar{x}_1', h_1', R_1) \rightarrow obsEq(\bar{x}_2, h_2, \bar{x}_2', h_2', R_2)\big)\end{aligned} \tag{5.5}$$

is valid in $(\mathcal{D}, s, \beta')$. Inspection of (5.5) under consideration of (5.4) shows that in the present situation $obsEq(\bar{x}_a, h_a, \bar{x}_a', h_a', R_1) \rightarrow obsEq(\bar{x}_b, h_b, \bar{x}_b', h_b', R_2)$ is true in $(\mathcal{D}, s, \beta)$, as desired. □

5.4 Discussion

Popular approaches to check for noninterference of programs are approximate methods like security type systems (a prominent example in this field is the JIF-System by Myers [1999a]), the analyses of the dependence graph of a program for graph-theoretical reachability properties (Hammer et al. [2006]), specialized approximate information flow calculi based on Hoare like logics (Amtoft et al. [2006]) and the usage of abstraction and ghost code for explicit tracking of dependencies (Pan [2005], Bubel et al. [2008], van Delft [2011]). These approaches are efficient, but do not have the precision of self-composition nor do they allow for as fine-grained specifications as they are possible with the help of observation expressions (Section 3.1). Nanevski et al. [2011] formalize information flow properties in a higher-order logic and use Coq for the verification of those properties. This approach seems to be extremely expressive, but comes with the price of more and more complex interactions with the proof system.

Almost all approaches mentioned so far check for unconditional information flow. There are only few approaches which study conditional information flow and in particular information flow contracts. One of the first contributions on conditional information flow was by Amtoft and Banerjee [2007]. They developed a Hoare logic for compositional *intra*procedural analyses of conditional information flow. This approach was the basis for a contribution on software contracts for conditional information flow for SPARK Ada by Amtoft et al. [2008]. The latter approach works on a relatively simple while-language including method calls. The handling of arrays was added in a later contribution (Amtoft et al. [2010]). Object-orientation is not supported. One advantage of the presented approach is that information flow and functional contracts can be combined easily. This results in highest possible precision whereas Amtoft et al. [2008] introduce fixed over-approximations.

Finally self-composition (Barthe et al. [2004], Darvas et al. [2005]) is a popular approach to state noninterference and use off-the-shelf software verification systems to check for it, as also presented in this chapter. The approach has been applied to full-fledged programming languages like Java.

To the best of the authors knowledge there are only very few contributions aiming at an improvement of the efficiency of the self-composition approach. A recent approach by Phan [2013] uses bounded symbolic execution (symbolic execution without inductive invariants) and a formulation of (non-conditional) noninterference based on symbolic traces which is quite near in spirit to the one of Theorem 9 (which goes back to Scheben and Schmitt [2012]). Phan found that with this formulation it is sufficient to symbolically execute a program only once. Independently of Phan [2013], the author of this thesis found that the same holds if the wp-calculus or Dynamic Logic is used, as presented in Section 5.2. Therefore, the presented approach is not restricted to bounded programs. Additionally, it can be used to check for conditional noninterference and with more fine-grained specifications. Barthe et al. [2011] build product programs to increase the level of automation in relational reasoning, which can also be used for information flow verification, but their focus is mainly on increasing the degree of automation and less on increasing efficiency.

Compositional/modular self-composition reasoning is also studied rarely: A contribution by Naumann [2006] duplicates each variable, field, parameter and method body in the Java source code and uses standard JML method contracts to state noninterference with the help of the duplicates. The contracts are verified with the help of ESC/Java2. This approach has the drawback that there is no obvious translation of JML annotations from the non-duplicated source to the duplicated source: an object invariant

```
invariant (\sum Object o;; 1) < 10;
```

for instance might evaluate differently in the duplicated code than in the non-duplicated one. The paper mentions vaguely how modularity on the method level could be achieved, but thorough investigation is left for future work. Another contribution by Dufay et al. [2005] introduces new JML-keywords which directly define relations between the program variables of two self-composed executions. In particular two keywords to distinguish the variables of the two runs are defined. The approach uses ghost code to store the return value and the values of parameters of the first run in order to use those values during the application of noninterference contracts in the second run. As the authors mention themselves, the approach is limited in case arrays are involved in method invocations. The author of this thesis does not see how even more complex data structures or equivalently complex heap manipulations can be tracked with ghost code. The proposed usage of ghost code seems to be a serious limitation of the approach. Resolving such limitations is mentioned as an aim of future work. The approach from Section 5.3 overcomes such limitations: it does not use additional ghost code and is not limited by its usage. Finally, a recent approach on modular relational verification by Hawblitzel et al. [2013], which can be used for information flow verification as well, targets at the same direction as Section 5.3. Hawblitzel et al. [2013] use mutual summaries, instead of information flow contracts, in order to prove relational properties modularly. Their approach, however, is not compatible with the efficient self-composition style reasoning presented in Section 5.2.

6 Object Orientation

This chapter is a revised version of Beckert et al. [2014]. The author of this thesis contributed on Sections 2, 4, 5.1 and 5.2 of the paper only to a smaller extent. Therefore, the results of these sections are repeated (in Section 6.1 below) only as far as they are necessary for the presentation of the subsequent results.

The former chapters handle sources and sinks of object type in the same way as sources and sinks of primitive type, that is, the definition of noninterference (Definition 3) requires their values to be identical. As a consequence, programs with low variables of object type will usually be classified insecure: one cannot expect that the Java Virtual Machine generates the same new objects in two runs starting in states with possibly differently many created "high" objects. On the other hand, references in Java are opaque, that is, they can be compared by the == function only (see Lindholm and Yellin [1999]), which allows to relax the notion of noninterference as already pointed out by Hedin and Sands [2005], Hansen and Probst [2006], Barthe et al. [2013], Banerjee and Naumann [2002], Beringer and Hofmann [2007] and Naumann [2006]. The classical notion of secure information flow can be replaced by object-sensitive secure information flow which uses a different notion of low-equivalence of states: for all states s_1, s_2, s_1', s_2' such that the observable values of s_1 and s_2 are related by a partial isomorphism π and such that the program α started in s_1 terminates in s_1' and α started in s_2 terminates in s_2' the observable values of s_1' and s_2' are related by a partial isomorphism extending π. This chapter investigates into the concept of object-sensitive secure information flow itself with the aim to find a feasible formalization of the property in Java DL. This formalization promises to be useful in self-composition approaches in general.

Object-sensitive secure information flow can be formalized in Java DL in the style of Theorem 9. The formalization is appealing, because it is semantically precise and, in contrast to other formalizations based on self-composition (see Section 6.5), it does not require changes or additions to the investigated program. Additionally, it can be easily realized on top of software verification systems like KeY. A naive encoding of the isomorphism property, however, increases the burden on the analysis considerably. Beckert et al. [2014] present in Section 5.2 an alternative but equivalent formulation of object-sensitive secure information flow: in this formulation the partial isomorphism π in the

```
final class C {
  static C x, y;      // low variables
  static boolean h;   // high variable
  static void m1() { if (h) {x = new C(); y = new C();}
                     else   {y = new C(); x = new C();} } }
```

Figure 6.1: Secure object creation, Beckert et al. [2014]

prestate is restricted to be the identity. This simplifies the formalization of object-sensitive secure information flow in Java DL (and hence its verification) already a lot.

This chapter shows that additionally restricting the partial isomorphism in the poststate to newly created objects leads to a marginally stronger, but more intuitive property. This property is a sufficient criterion for object-sensitive secure information flow (Theorem 26) and, in contrast to object-sensitive secure information flow, it is compositional (Theorem 29) which is considered an indispensable prerequisite for modular verification of information flow properties (see Section 5.3). The sufficient criterion can be formalized in a way which reduces the burden on analysis tools significantly (Section 6.3). The main difference between the original property and the sufficient criterion is that the criterion admits the attacker the ability to distinguish between newly created objects and objects which already existed in the prestate.

6.1 Information Flow in Java

Figure 6.1 reproduces a typical example of a program for which flow(`m1()`, $\langle$`x`, `y`$\rangle$, $\langle$`x`, `y`$\rangle$) does not hold, but which is considered secure in object-sensitive information flow. In this example it is impossible to learn the value of `h` by observation of `x` and `y` if object references can be compared by the == function only. The abstraction that references in Java may be treated opaque might be broken by native methods such as `Object::hashCode()`, as demonstrated by Hedin and Sands [2006]. This potential leakage, however, can be dealt with by assigning a high security level to the output of native methods.

The attacker model in object-sensitive secure information flow is identical to the one from Section 3.1, with the additional assumption that attackers cannot learn more than object identity from object references. In particular, the order in which objects have been generated cannot be learned. Attackers may, on the other hand, retrieve the runtime type $type(e^s)$ of object valued expressions and, for array references, their length $len(e^s)$.

6.1.1 Isomorphisms

Isomorphisms for typed structures, as treated for instance in Mitchell [1990], are defined between structures $(\mathcal{D}_1, s_1)$ and $(\mathcal{D}_2, s_2)$.

Definition 14. *An isomorphism ρ from a Σ structure $(\mathcal{D}_1, s_1)$ onto the Σ structure $(\mathcal{D}_2, s_2)$ is a bijection from the universe D_1 of $(\mathcal{D}_1, s_1)$ onto the universe D_2 of $(\mathcal{D}_2, s_2)$ satisfying*

1. *for all types T and all $d \in D_1$:* $\quad d \in T^{\mathcal{D}_1, s_1} \Leftrightarrow \rho(d) \in T^{\mathcal{D}_2, s_2}$
2. *for any n-ary function symbol $f \in \Sigma$ and all n-tuples $\bar{d} \in D_1^n$:* $\rho(f^{\mathcal{D}_1, s_1}(\bar{d})) = f^{\mathcal{D}_2, s_2}(\rho(\bar{d}))$
3. *for any n-ary predicate symbol $p \in \Sigma$ and all n-tuples $\bar{d} \in D_1^n$:* $(\mathcal{D}_1, s_1) \models p(\bar{d}) \Leftrightarrow (\mathcal{D}_2, s_2) \models p(\rho(\bar{d}))$

For $\bar{d} = (d_1, \ldots, d_n)$ the notion $\rho(\bar{d})$ is a shorthand for $(\rho(d_1), \ldots, \rho(d_n))$.

We will consider isomorphisms from $(\mathcal{D}, s_1)$ onto $(\mathcal{D}, s_2)$ for an arbitrary, but fixed $\mathcal{D}$ only. Therefore, in the following $\mathcal{D}$ will usually be omitted.

Bijections between objects of typed structures enjoy the following property:

Lemma 15. *Let $(\mathcal{D}, s)$ be a structure and let π be a bijection from X onto Y for finite subsets $X, Y \subseteq Obj^{\mathcal{D}}$ such that*

1. *If $null \in X$ then $\pi(null) = null$ and $null \in Y$ implies $null \in X$.*
2. *π preserves the exact types of its arguments.*
3. *π preserves the length of array objects.*

Then there is an isomorphism $\rho : s \to \rho(s)$ extending the bijection π.

Partial isomorphisms with respect to observation expressions R, as they are needed for the definition of object-oriented secure information flow, are defined with the help of the objects observable by R.

Definition 16. *$Obj(R^s)$ denotes the set of objects observable by observation expression R in state s, that is,*

$$Obj(R^s) = \{o \in Obj^{\mathcal{D}} \mid \exists i.(o = R^s[i])\} \cup \bigcup_{i \in \{j \mid R^s[j] \in Seq^{\mathcal{D}}\}} Obj(R^s[i])\,.$$

Definition 17 (Partial isomorphism w. r. t. R)**.** *Let R be an observation expression and s_1, s_2 be two states.*

A partial isomorphism with respect to R from s_1 onto s_2 is a bijection $\pi : Obj(R^{s_1}) \to Obj(R^{s_2})$ such that

(a) the requirements of Lemma 15 hold and

(b) $\pi_{Seq}(R^{s_1}) = R^{s_2}$ *where* π_{Seq} *is defined on sequences as*

$$\pi_{Seq}(\langle e_1, \ldots, e_k \rangle) = \langle e'_1, \ldots, e'_k \rangle$$

where $e'_i = \pi(e_i)$ *if* $e_i \in Obj^{\mathcal{D}}$, $e'_i = \pi_{Seq}(e_i)$ *if* $e_i \in Seq^{\mathcal{D}}$ *and* $e'_i = e_i$ *else.*

To simplify notation, every partial isomorphism π *is extended to all primitive values* w *by* $\pi(w) = w$*. If* π *is a partial isomorphism from* s_1 *onto* s_2*, written* $s_2 = \pi(s_1)$*, then* s_1 *and* s_2 *are called isomorphic.*

If all program variables p are observable by R, that is, if for all program variables p there exists an index i such that $p = R[i]$, then every isomorphism extending a partial isomorphism π with respect to R according to Lemma 15 is a total isomorphism from s_1 onto s_2, because $\pi(p^{s_1}) = p^{s_2}$ by requirement (b). On the other hand, not every partial isomorphism can be extended to a total isomorphism. If q is a program variable such that q does not appear as a subterm in R, then $\pi(q^{s_1}) = q^{s_2}$ is not required.

To clarify the role of condition (b) in Definition 17 consider the following example. Let x be a program variable of type C and f a field in C, say of type integer, such that $R = \langle x, f(x) \rangle$. Let s_1, s_2 be states. In this case condition (b) implies $\pi((f(x))^{s_1}) = (f(x))^{s_2} = f^{s_2}(x^{s_2}) = f^{s_2}(\pi(x^{s_1}))$. This amounts to the usual requirements of isomorphisms on mathematical structures.

6.1.2 Formalization of Opaqueness of References

Object-sensitive noninterference relies on opaqueness of references. This section formalizes opaqueness of references in terms of isomorphisms between structures. If references are opaque, then the behavior of Java programs only depend on the values of references up to comparison by ==. Therefore, a program α started in isomorphic states also terminates in isomorphic states (if α terminates).

Postulate 18. *Let* s_1*,* s_2*,* s'_1 *be states. Let* α *be a program which started in* s_1 *terminates in* s_2*, and let* $\rho : s_1 \to s'_1$ *be an isomorphism.*

Then there exists an isomorphism $\rho' : s_2 \to s'_2$ *that coincides with* ρ *on all objects existing in state* s_1 *(that is,* $\rho(o) = \rho'(o)$ *for all* $o \in Obj^{\mathcal{D}}$ *with* $created^{s_1}(o) = tt$*) such that* α *started in* $\rho(s_1)$ *terminates in* $\rho'(s_2)$*.*[1]

[1] The notation $\rho(s)$ is defined below Definition 14.

The isomorphism ρ' may be different form ρ, because α may generate new objects and there is no reason why a new element o' generated in the run of α starting in $\rho(s_1)$ should be the ρ-image of the new element o generated in the run starting in s_1.

6.1.3 Basic Object-Sensitive Noninterference

To formalize object-sensitive noninterference, it is helpful first to introduce a notion for the object-sensitive indistinguishability of states.

Definition 19 (Object-Sensitive Agreement of States)**.** *Let R be an observation expression.*

In an object-sensitive setting two states s, s' agree on R, denoted by $\text{agree}_{os}(R, s, s')$, if and only if there exists a partial isomorphism $\pi : Obj(R^s) \to Obj(R^{s'})$ with respect to R. The partial isomorphism π is uniquely determined by R, s and s'. The notation $\text{agree}_{os}(R, s, s', \pi)$ indicates that $\text{agree}_{os}(R, s, s')$ is true and π is the mapping thus defined.

Note that by Definition 17 $\text{agree}_{os}(R, s, s')$ entails $R^s[i] = R^{s'}[i]$ for all subterms $R[i]$ of primitive type. Object-sensitive noninterference is formalized with the help of isomorphisms on structures as follows.

Definition 20 (Object-Sensitive Noninterference)**.** *Let α be a program and R_1 and R_2 be two observation expressions.*

In an object-sensitive setting a program α allows information to flow only from R_1 to R_2 when started in s_1, denoted by $\text{flow}_{os}(s_1, \alpha, R_1, R_2)$, if and only if for all object creation orders o, o' and all states s_1', s_2, s_2' such that α started in s_1 with order o terminates in s_2 and α started in s_1' with order o' terminates in s_2' the following applies

if $\text{agree}_{os}(R_1, s_1, s_1', \pi_1)$ *for some* π_1
then $\text{agree}_{os}(R_2, s_2, s_2', \pi_2)$ *for some* π_2 *that is compatible with* π_1

where π_2 is said to be compatible with π_1 if
$\pi_2(o) = \pi_1(o)$ *for all* $o \in Obj(R_1^{s_1}) \cap Obj(R_2^{s_2})$ *with* $created^{s_1}(o) = tt$.

$\text{flow}_{os}(\alpha, R_1, R_2)$ *denotes the case that* $\text{flow}_{os}(s_1, \alpha, R_1, R_2)$ *holds for all states* s_1.

Note that the object creation order in the two runs of α might differ which models that the two runs might be executed in different environments, for instance on different Java Virtual Machines. As a consequence, α started two times in s_1

might result in different (but isomorphic) final states. Though Chapter 3 does not address this issue, this fact applies there, too.

For presentational purposes, Definition 20 does not allow for conditional information flow statements, but can—as well as all subsequent results—be extended to conditional information flow along the lines of Chapters 3 to 5.

The following extension of method `m1()` from Figure 6.1 illustrates Definition 20:

```
final class C {
  static C x, y;      // low variables
  static boolean h;   // high variable
  C next;

  static void m2() {
    if (h) { x = new C(); y = new C(); x.next = y; }
    else   { y = new C(); x = new C(); x.next = x; }
  }
}
```

Whether `m2()` leaks information or not depends on the examined observation expression. For $R = \langle C.x, C.y \rangle$ the observation will always consist of two freshly created, distinct object references. If $\text{agree}_{\text{os}}(R, s_1, s_1', \pi_1)$, the partial isomorphism π_2 defined as an extension of π_1 by $\pi_2(x^{s_2}) = x^{s_2'}$ and $\pi_2(y^{s_2}) = y^{s_2'}$ ensures $\text{agree}_{\text{os}}(R, s_2, s_2', \pi_2)$ and, therefore, $\text{flow}_{\text{os}}(\texttt{m2()}, R, R)$.

On the other hand, if $R' = \langle C.x, C.y, C.x.next \rangle$ is chosen, π_2 is no longer a partial isomorphism as $\pi_2(next^{s_2}(x^{s_2})) = next^{s_2'}(x^{s_2'})$ would need to hold. But if $h^{s_1} = \texttt{true}$ and $h^{s_1'} = \texttt{false}$, the resulting heap structures are not isomorphic: $\pi_2(next^{s_2}(x^{s_2})) = \pi_2(y^{s_2})$ and $next^{s_2'}(x^{s_2'}) = x^{s_2'} = \pi_2(x^{s_2})$ which cannot be equal as π_2 is an injection. The attacker can learn the value of h by comparing x and $x.next$. $\text{flow}_{\text{os}}(\texttt{m2()}, R', R')$ does not hold.

The information flow property of Definition 20 is compositional in the following sense.

Lemma 21 (Compositionality of flow_{os})**.** *Let α_1, α_2 be programs and $\alpha_1;\alpha_2$ their sequential composition. Further let R_1, R_2, R_3 be observation expressions and s_1, s_2 be states.*

If $\text{flow}_{\text{os}}(s_1, \alpha_1, R_1, R_2)$, $\text{flow}_{\text{os}}(s_2, \alpha_2, R_2, R_3)$ as well as the condition $Obj(R_1^{s_1}) \cap Obj(R_3^{s_3}) \subseteq Obj(R_2^{s_2})$ hold for all s_1, s_2, s_3 such that α_1 started in s_1 terminates in s_2 and α_2 started in s_2 terminates in s_3, then $\text{flow}_{\text{os}}(\alpha_1;\alpha_2, R_1, R_3)$ holds.

The following lemma on agree_{os} is needed for later reference.

Lemma 22. *If* $agree_{os}(R, s, s', \pi)$ *and if* $\rho : s \to \rho(s)$ *and* $\rho' : s' \to \rho'(s')$ *are isomorphisms then*

(1) $agree_{os}(R, s, \rho'(s'), \rho' \circ \pi)$ *and*

(2) $agree_{os}(R, \rho(s), s', \pi \circ \rho^{-1})$.

6.1.4 Optimized Object-Sensitive Noninterference

The basic notion of object-sensitive noninterference can be optimized. To this end, this section introduces a simpler, but semantically equivalent definition of object-sensitive noninterference. The definition restricts the partial isomorphism in the prestate to be the identity. This simplifies the formulation of verification conditions considerably (see Theorem 30 below), also making them easier to verify.

Definition 23 (Optimized Object-Sensitive Noninterference)**.** *Let* α *be a program and* R_1 *and* R_2 *be two observation expressions.*

In an object-sensitive setting a program α *allows information to flow only from* R_1 *to* R_2 *when started in* s_1*, denoted by* $flow^*_{\mathrm{os}}(s_1, \alpha, R_1, R_2)$*, if and only if for all states* s_1'*,* s_2*,* s_2' *such that* α *started in* s_1 *terminates in* s_2 *and* α *started in* s_1' *terminates in* s_2' *(for arbitrary object creation orders) the following applies:*

if $agree_{os}(R_1, s_1, s_1', id)$
then $agree_{os}(R_2, s_2, s_2', \pi_2)$ *for some* π_2 *compatible with* id.

$\mathrm{agree}_{\mathrm{os}}(R_1, s_1, s_1', id)$ implies in particular $Obj(R_1^{s_1}) = Obj(R_1^{s_1'})$ since $\pi_1 = id$ is a bijection from $Obj(R_1^{s_1})$ onto $Obj(R_1^{s_1'})$.

Theorem 24. *For all programs* α*, any two observation expressions* R_1 *and* R_2*, and any state* s_1 $\quad flow^*_{\mathrm{os}}(s_1, \alpha, R_1, R_2) \quad \Leftrightarrow \quad flow_{\mathrm{os}}(s_1, \alpha, R_1, R_2)$.

Postulate 18 (on page 70) is essential for the proof of Theorem 24.

6.2 An Efficient Compositional Criterion

Though Definition 23 already simplifies the formulation of verification conditions and consequently checking for $\mathrm{flow}_{\mathrm{os}}$, it can still be improved. This section presents a slightly stronger information flow property, $\mathrm{flow}^{**}_{\mathrm{os}}$, which is simpler to check. $\mathrm{flow}^{**}_{\mathrm{os}}$ is a criterion for $\mathrm{flow}_{\mathrm{os}}$, that is, a sufficient but not a necessary condition. Roughly speaking, the main difference between $\mathrm{flow}_{\mathrm{os}}$

and $\text{flow}^{**}_{\text{os}}$ is that $\text{flow}^{**}_{\text{os}}$ admits the attacker the ability to distinguish between newly created objects and objects which already existed in the prestate. This property of $\text{flow}^{**}_{\text{os}}$ is responsible for its compositionality (Theorem 29), which is an indispensable prerequisite for modular verification of information flow properties. In $\text{flow}^{**}_{\text{os}}$, the partial isomorphisms differs from the identity on new objects only. This reduces the effort to verify $\text{flow}^{**}_{\text{os}}$ considerably if only few or no new objects are created. Also, there is no obligation that one isomorphism is an extension of another.

A disadvantage of $\text{flow}^{**}_{\text{os}}$ is that it requires an additional observation expression N_2 which exactly names the new elements of the set of objects observable in the poststate. Also, it has to be proven that N_2 exactly names the new elements. However, N_2 is usually an explicit subexpression of R_2 and can be named easily. Proving that N_2 exactly names the new elements is in the majority of cases a simple task as well.

Definition 25 (Strong Object-Sensitive Noninterference)**.** *Let N_2 be an observation expression such that all terms in N_2 are of object type. Further, let α be a program, R_1, R_2 observation expressions, and s_1 a state.*

The predicate $\text{flow}^{**}_{\text{os}}(s_1, \alpha, R_1, R_2, N_2)$ is true if and only if the following applies for all states s_1', s_2, s_2' such that α started in s_1 terminates in s_2 and α started in s_1' terminates in s_2' (for arbitrary object creation orders):

$$
\begin{array}{ll}
\textit{if} & \textit{agree}_{os}(R_1, s_1, s_1', id) \\
\textit{then} & \textit{all objects in } \textit{Obj}(N_2^{s_2}) \textit{ and } \textit{Obj}(N_2^{s_2'}) \textit{ are new and} \\
& \textit{agree}_{os}(N_2, s_2, s_2', \pi) \textit{ for a partial isomorphism } \pi \textit{ and} \\
& \textit{if} \quad \textit{agree}_{os}(N_2, s_2, s_2', id) \quad \textit{then} \quad \textit{agree}_{os}(R_2, s_2, s_2', id)
\end{array}
$$

As before, $\text{flow}^{**}_{\text{os}}(\alpha, R_1, R_2, N_2)$ denotes the case that $\text{flow}^{**}_{\text{os}}(s_1, \alpha, R_1, R_2, N_2)$ holds for all states s_1.

Theorem 26. *Let N_2 be an observation expression such that all expressions in N_2 are of object type. Further, let α be a program, R_1, R_2 observation expressions, and s_1 a state.*

1. $\textit{flow}^{**}_{\text{os}}(s_1, \alpha, R_1, R_2, N_2) \Rightarrow \textit{flow}_{\text{os}}(s_1, \alpha, R_1, R_2)$.
2. $$
\begin{array}{ll}
\textit{If} & \textit{for all states } s_2 \textit{ such that } \alpha \textit{ started in } s_1 \textit{ terminates in } s_2 \\
& \textit{Obj}(N_2^{s_2}) = \{o \in \textit{Obj}(R_2^{s_2}) \mid \textit{created}^{s_1}(o) = \textit{ff}\} \textit{ and} \\
& \{o \in \textit{Obj}(R_2^{s_2}) \mid \textit{created}^{s_1}(o) = tt\} \subseteq \textit{Obj}(R_1^{s_1}) \\
\textit{then} & \textit{flow}_{\text{os}}(s_1, \alpha, R_1, R_2) \Rightarrow \textit{flow}^{**}_{\text{os}}(s_1, \alpha, R_1, R_2, N_2)\ .
\end{array}
$$

The proof of Theorem 26 uses the following auxiliary lemma. The lemma states that there are always object creation orders such that in two runs of a program

α, which are started in R equivalent states, the same new objects are chosen for those objects which are observable by R.

Lemma 27. *Let α be a program such that α started in s_1 terminates in s_2, α started in s_1' terminates in s_2', $\text{agree}_{os}(R, s_1, s_1', id)$ and $\text{agree}_{os}(N, s_2, s_2', \pi)$ hold true for observation expressions R and N. In addition let all objects in $Obj(N^{s_2})$ and $Obj(N^{s_2'})$ be new.*

Then there are isomorphisms $\rho : s_2 \to \rho(s_2)$ and $\rho' : s_2' \to \rho'(s_2')$ and object creation orders such that (1) α started in s_1 terminates in $\rho(s_2)$, (2) α started in s_1' terminates in $\rho'(s_2')$, (3) $\text{agree}_{os}(N, \rho(s_2), \rho'(s_2'), id)$, and (4) $\rho(o) = o$ and $\rho'(o) = o$ for all o existing in state s_1 or s_1'.

Proof of Lemma 27. Let ρ_{00} be a type preserving and array-length preserving injective function from $Obj(N^{s_2})$ to $Obj^{\mathcal{D}}$, such that the image $\rho_{00}(Obj(N^{s_2}))$ only contains objects that were not already created in states s_1 or s_1'. Because the universe D of $\mathcal{D}$ contains an infinite reservoir of non-created objects, such a choice is possible. Let ρ_0 be the extension of ρ_{00} that is the identity on all (finitely many) objects existing in state s_1. Let $\rho_{00}' = \rho_{00} \circ \pi^{-1}$ and ρ_0' the extension of ρ_{00}' which is the identity on all objects existing in s_1'. By Lemma 15 there are isomorphisms $\rho : s_2 \to \rho(s_2)$ and $\rho' : s_2' \to \rho'(s_2')$ extending ρ_0 and ρ_0', respectively. By Lemma 22, $\text{agree}_{\text{os}}(N, s_2, s_2', \pi)$ implies $\text{agree}_{\text{os}}(N, \rho(s_2), \rho'(s_2'), \rho' \circ \pi \circ \rho^{-1})$. By construction of ρ, ρ' the isomorphism $\rho' \circ \pi \circ \rho^{-1}$ is the identity function on $Obj(N^2)$, thus $\text{agree}_{\text{os}}(N, \rho(s_2), \rho'(s_2'), id)$ holds. $\rho(o) = o$ and $\rho'(o) = o$ for all o existing in state s_1 or s_1' follows from the construction, too. Further, the isomorphisms ρ and ρ' are chosen such that still (1) every newly created object is different from $null$; (2) the dynamic types of new objects correspond to the type allocated by the Java program; and (3) no newly created object is already created in s_1 or s_1'. Therefore, there are object creation orders such that α started in s_1 terminates in $\rho(s_2)$ and α started in s_1' terminates in $\rho'(s_2')$ (compare Section 2.1). □

Proof of Theorem 26.

Part 1: We assume $\text{flow}_{\text{os}}^{**}(s_1, \alpha, R_1, R_2, N_2)$ and show $\text{flow}_{\text{os}}^{*}(s_1, \alpha, R_1, R_2)$. To this end, let s_1', s_2, s_2' be states such that α started in s_1 terminates in s_2, α started in s_1' terminates in s_2' and $\text{agree}_{\text{os}}(R_1, s_1, s_1', id)$. It has to be shown that $\text{agree}_{\text{os}}(R_2, s_2, s_2', \pi)$ holds, where the uniquely determined partial isomorphism π is compatible with id.

By assumption we obtain $\text{agree}_{\text{os}}(N_2, s_2, s_2', \sigma)$ for some partial isomorphism σ and we know that all objects in $Obj(N_2^{s_2})$ and $Obj(N_2^{s_2'})$ are new. By Lemma 27, there are isomorphisms $\rho : s_2 \to \rho(s_2)$, $\rho' : s_2' \to \rho'(s_2')$ and object creation orders such that α started in s_1 terminates in $\rho(s_2)$, α started in s_1' terminates in

$\rho'(s_2')$, and $\text{agree}_{\text{os}}(N_2, \rho(s_2), \rho'(s_2'), id)$ holds. Another appeal to the definition of $\text{flow}^{**}_{\text{os}}(s_1, \alpha, R_1, R_2, N_2)$, now for the object creation orders from Lemma 27, yields $\text{agree}_{\text{os}}(R_2, \rho(s_2), \rho'(s_2'), id)$. By Lemma 22, the latter implies $\text{agree}_{\text{os}}(R_2, s_2, s_2', (\rho')^{-1} \circ \rho)$. Further, $(\rho')^{-1} \circ \rho(o) = o$ holds by Lemma 27 for all $o \in Obj(R_1^{s_1}) \cap Obj(R_2^{s_2})$. Thus $(\rho')^{-1} \circ \rho$ is compatible with id and the claim is proved.

Part 2: For the reverse implication we assume $\text{flow}^{*}_{\text{os}}(s_1, \alpha, R_1, R_2)$ and aim at proving $\text{flow}^{**}_{\text{os}}(s_1, \alpha, R_1, R_2, N_2)$. Let s_1', s_2, s_2' be states such that α started in s_1 terminates in s_2, α started in s_1' terminates in s_2', and $\text{agree}_{\text{os}}(R_1, s_1, s_1', id)$. From $\text{flow}^{*}_{\text{os}}(s_1, \alpha, R_1, R_2)$ we obtain $\text{agree}_{\text{os}}(R_2, s_2, s_2', \pi)$ where π is compatible with id. By case assumption, $Obj(N_2^{s_2}) = \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = \mathit{ff}\}$ holds. Therefore, π is a partial isomorphism from $Obj(N_2^{s_2})$ onto $Obj(N_2^{s_2'})$. This yields $\text{agree}_{\text{os}}(N_2, s_2, s_2', \pi)$.

In the following, we assume $\text{agree}_{\text{os}}(N_2, s_2, s_2', id)$ with the intention to show $\text{agree}_{\text{os}}(R_2, s_2, s_2', id)$. First of all, $\text{agree}_{\text{os}}(R_2, s_2, s_2', \pi)$ implies $\pi_{Seq}(R_2^{s_2}) = R_2^{s_2'}$, where π_{Seq} is defined as $\pi_{Seq}(\langle e_1, \ldots, e_k\rangle) = \langle e_1', \ldots, e_k'\rangle$ with

$$e_i' = \begin{cases} \pi(e_i) & \text{if } e_i \in Obj^{\mathcal{D}} \\ \pi_{Seq}(e_i) & \text{if } e_i \in Seq^{\mathcal{D}} \\ e_i & \text{else} \end{cases} .$$

It remains to be shown that $\pi(e_i) = e_i$ for $e_i \in Obj(R_2^{s_2})$. We distinguish two cases: (1) $created^{s_1}(e_i) = tt$ and (2) $created^{s_1}(e_i) = \mathit{ff}$.

In case (1) we obtain $e_i \in Obj(R_1^{s_1})$ by case assumption $\{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$. Hence $\pi(e_i) = e_i$ since π is compatible with id. In case (2) we use assumptions $\text{agree}_{\text{os}}(N_2, s_2, s_2', id)$ and $Obj(N_2^{s_2}) = \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = \mathit{ff}\}$, and also arrive at $\pi(e_i) = e_i$. □

The remainder of this section shows compositionality of $\text{flow}^{**}_{\text{os}}$. The key property of $\text{flow}^{**}_{\text{os}}$ compared to flow_{os} which is responsible for its compositionality is that an attacker can observe in the poststate of a program run at most objects which are newly created or which already have been observed in the prestate. More precisely, Theorem 29 below shows the compositionality of

(1) $\text{flow}_{\text{os}}(s_1, \alpha, R_1, R_2)$ in combination with

(2) $\text{agree}_{\text{os}}(R_1, s_1, s_1')$ implies $\{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$.

Both properties are implied by $\text{flow}^{**}_{\text{os}}$, as the next lemma shows.

Lemma 28. *Let N_2 be an observation expression such that all terms in N_2 are of object type. Further, let α be a program, R_1, R_2 observation expressions, and s_1 a state. Let s_1', s_2, s_2' be states such that α started in s_1 terminates in s_2 and α started in s_1' terminates in s_2'.*

Then $\mathit{flow}^{**}_{\mathrm{os}}(s_1, \alpha, R_1, R_2, N_2)$ implies

(1) $\mathit{flow}_{\mathrm{os}}(s_1, \alpha, R_1, R_2)$ and additionally that

(2) $\mathit{agree}_{\mathit{os}}(R_1, s_1, s_1')$ implies $\{o \in \mathit{Obj}(R_2^{s_2}) \mid \mathit{created}^{s_1}(o) = \mathit{tt}\} \subseteq \mathit{Obj}(R_1^{s_1})$.

Proof. By Theorem 26 and Theorem 24, $\mathrm{flow}_{\mathrm{os}}(s_1, \alpha, R_1, R_2)$ follows directly from $\mathrm{flow}^{**}_{\mathrm{os}}(s_1, \alpha, R_1, R_2, N_2)$.

For the second part, we may assume $\mathrm{agree}_{\mathrm{os}}(R_1, s_1, s_1', \pi_1)$. By Lemma 15, there is an isomorphism $\rho_0 : s_1' \to \rho_0(s_1')$ extending $(\pi_1)^{-1}$. Thus we may conclude $\mathrm{agree}_{\mathrm{os}}(R_1, s_1, \rho_0(s_1'), \rho_0 \circ \pi_1)$ using Lemma 22. Since ρ_0 extends $(\pi_1)^{-1}$ we have $\mathrm{agree}_{\mathrm{os}}(R_1, s_1, \rho_0(s_1'), id)$.

We prove $\{o \in \mathit{Obj}(R_2^{s_2}) \mid \mathit{created}^{s_1}(o) = \mathit{tt}\} \subseteq \mathit{Obj}(R_1^{s_1})$ by contradiction. Assume $\mathrm{flow}^{**}(s_1, \alpha, R_1, R_2, N_2)$, $\mathrm{agree}(R_1, s_1, \rho_0(s_1'), id)$ and $o \in \mathit{Obj}(R_2^{s_2})$ such that $\mathit{created}^{s_1}(o) = \mathit{tt}$ and $o \notin \mathit{Obj}(R_1^{s_1})$. Because $o \in \mathit{Obj}(R_2^{s_2})$, there is an index i such that $R_2[i]^{s_2} = o$.

By $\mathrm{flow}^{**}(s_1, \alpha, R_1, R_2, N_2)$ and Postulate 18, we get $\mathrm{agree}(N_2, s_2, \rho_0'(s_2'), \pi_2)$ for a partial isomorphism π_2 and an isomorphism ρ_0' which equals ρ_0 for all objects created in $\rho_0(s_1')$. Additionally, we know that all objects in $\mathit{Obj}(N_2^{s_2})$ and $\mathit{Obj}(N_2^{\rho_0'(s_2')})$ are new. By Lemma 27, there are isomorphisms $\rho : s_2 \to \rho(s_2)$ and $\rho' : \rho_0'(s_2') \to \rho'(\rho_0'(s_2'))$ and object creation orders such that α started in s_1 terminates in $\rho(s_2)$, α started in $\rho_0(s_1')$ terminates in $\rho'(\rho_0'(s_2'))$, and additionally $\mathrm{agree}_{\mathrm{os}}(N_2, \rho(s_2), \rho'(\rho_0'(s_2')), id)$ holds. Another appeal to the definition of $\mathrm{flow}^{**}_{\mathrm{os}}(s_1, \alpha, R_1, R_2, N_2)$, now for the object creation orders from Lemma 27, yields $\mathrm{agree}_{\mathrm{os}}(R_2, \rho(s_2), \rho'(\rho_0'(s_2')), id)$. By Lemma 22, the latter implies that $\mathrm{agree}_{\mathrm{os}}(R_2, s_2, s_2', (\rho_0')^{-1} \circ (\rho')^{-1} \circ \rho)$ holds.

On the other hand, o is neither in the domain of π_1 of $\mathrm{agree}(R_1, s_1, s_1', \pi_1)$ nor of id of $\mathrm{agree}_{\mathrm{os}}(N_2, \rho(s_2), \rho'(\rho_0'(s_2')), id)$ and thus there is no reason why $R_2[i]^{s_2'} = (\rho_0')^{-1} \circ (\rho')^{-1} \circ \rho(o)$ indeed should hold. We will use this intuition to derive a contradiction to $\mathrm{agree}_{\mathrm{os}}(R_2, s_2, s_2', (\rho_0')^{-1} \circ (\rho')^{-1} \circ \rho)$.

To this end, we need to assume that $R_2[i]^{s_2'} = (\rho_0')^{-1} \circ (\rho')^{-1} \circ \rho(o)$ "accidentally" holds true (which is possible). We construct an isomorphism ρ_1' such that $\mathrm{agree}_{\mathrm{os}}(R_2, s_2, s_2', (\rho_1')^{-1} \circ (\rho')^{-1} \circ \rho)$ still needs to hold, but for which $R_2[i]^{s_2'} = (\rho_1')^{-1} \circ (\rho')^{-1} \circ \rho(o)$ definitely is false (which then is a contradiction).

Let o' be a so far "uninvolved" object, that is, let o' be neither created in s_2 nor in $\rho(s_2)$, s_2', $\rho_0'(s_2')$, or $\rho'(\rho_0'(s_2'))$. Further, let ρ_1 be an isomorphism which is defined as follows:

$$\rho_1(x) = \begin{cases} o' & \text{if } x = \rho_0^{-1}(o) \\ \rho_0^{-1}(o) & \text{if } x = o' \\ \rho_0(x) & \text{else} \end{cases}$$

Because neither o nor o' are in the domain of π_1 and therefore $\rho_0^{-1}(o)$ and $\rho_0^{-1}(o')$ are not in $Obj(R_1^{s_1'})$, the isomorphism ρ_1 is still an extension of π_1^{-1}.

By flow$^{**}(s_1, \alpha, R_1, R_2, N_2)$ and Postulate 18, we get agree$(N_2, s_2, \rho_1'(s_2'), \pi_2)$ for an isomorphism ρ_1' which equals ρ_1 on all objects created in $\rho_1(s_1')$. Further, all objects in $Obj(N_2^{s_2})$ and $Obj(N_2^{\rho_1'(s_2')})$ are new. An inspection of the proof of Lemma 27 shows that $\rho : s_2 \to \rho(s_2)$ and $\rho' : \rho_1'(s_2') \to \rho'(\rho_1'(s_2'))$ are still isomorphisms such that α started in s_1 terminates in $\rho(s_2)$, α started in $\rho_1(s_1')$ terminates in $\rho'(\rho_1'(s_2'))$, and agree$_{\text{os}}(N_2, \rho(s_2), \rho'(\rho_1'(s_2')), id)$ for suitable object creation orders. Another appeal to flow$_{\text{os}}^{**}(s_1, \alpha, R_1, R_2, N_2)$, now for the object creation orders from Lemma 27, yields agree$_{\text{os}}(R_2, \rho(s_2), \rho'(\rho_1'(s_2')), id)$. By Lemma 22, the latter implies agree$_{\text{os}}(R_2, s_2, s_2', (\rho_1')^{-1} \circ (\rho')^{-1} \circ \rho)$.

By the construction of ρ_1', ρ' and ρ we have $R_2[i]^{s_2'} = (\rho_1')^{-1} \circ (\rho')^{-1} \circ \rho(o) = o'$. This is a contradiction to $R_2[i]^{s_2'} = (\rho_0')^{-1} \circ (\rho')^{-1} \circ \rho(o)$ from above, because $o' \neq (\rho_0')^{-1}(o)$ by definition. □

Theorem 29 (Compositionality of flow**)**.** *Let s_1, s_2, s_3 be states such that α_1 started in s_1 terminates in s_2 and α_2 started in s_2 terminates in s_3. If*

1. *flow$(s_1, \alpha_1, R_1, R_2)$,*
2. *flow$(s_2, \alpha_2, R_2, R_3)$,*
3. *for all states s_1' we have agree$(R_1, s_1, s_1') \Rightarrow \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$ and*
4. *for all states s_2' we have agree$(R_2, s_2, s_2') \Rightarrow \{o \in Obj(R_3^{s_3}) \mid created^{s_2}(o) = tt\} \subseteq Obj(R_2^{s_2})$*

then
flow$(s_1, \alpha_1; \alpha_2, R_1, R_3)$ and for all states s_1' we have
agree$(R_1, s_1, s_1') \Rightarrow \{o \in Obj(R_3^{s_3}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$.

Proof. Let s_1', s_2', s_3' be states such that α_1 started in s_1' terminates in s_2' and α_2 started in s_2' terminates in s_3'. We may assume agree(R_1, s_1, s_1'), otherwise the

lemma is trivially true. By $\text{flow}(s_1, \alpha_1, R_1, R_2)$ we derive $\text{agree}(R_2, s_2, s_2')$ and thus we have by assumption

$$\{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1}) \text{ and} \tag{6.1}$$
$$\{o \in Obj(R_3^{s_3}) \mid created^{s_2}(o) = tt\} \subseteq Obj(R_2^{s_2}) \tag{6.2}$$

We show $\text{flow}(s_1, \alpha_1; \alpha_2, R_1, R_3)$ with the help of Lemma 21. Thus it is sufficient to show (1) $Obj(R_1^{s_1}) \cap Obj(R_3^{s_3}) \subseteq Obj(R_2^{s_2})$ and (2) $\{o \in Obj(R_3^{s_3}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$.

By (6.1) and (6.2) we get $\{o \in Obj(R_3^{s_3}) \mid created^{s_1}(o) = tt\} \subseteq \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$ and thus $Obj(R_1^{s_1}) \cap Obj(R_3^{s_3}) = \{o \in Obj(R_3^{s_3}) \mid created^{s_1}(o) = tt\} \subseteq \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_2^{s_2})$. □

6.3 Formalisation in JavaDL

The overall goal is to prove information flow properties $\text{flow}_{\text{os}}(\alpha, R_1, R_2)$ for particular programs α and particular observation expressions R_i. To this end, this section presents a formulation of strong object-sensitive noninterference in Java DL in the lines of Theorem 9. The construction of the formula $\phi_{\alpha,R_1,R_2,N_2}$ expressing $\text{flow}^{**}_{\text{os}}(\alpha, R_1, R_2, N_2)$ is explained step by step.

$\text{flow}^{**}_{\text{os}}(\alpha, R_1, R_2, N_2)$ quantifies over states. A state s is a mapping of the (finitely many) program variables $\bar{\mathtt{x}}$ and `heap` to values. By Lemma 8, updates of the form $\{\mathtt{heap} := h \parallel \bar{\mathtt{x}} := \bar{x}\}$ can be used to refer to multiple states within one structure $(\mathcal{D}, s)$. Thus, quantification over all states can be achieved as in Theorem 9 by quantification over all values $\bar{x}$ and h of $\bar{\mathtt{x}}$ and `heap`:

$$\forall h.\forall \bar{x}.\{\mathtt{heap} := h \parallel \bar{\mathtt{x}} := \bar{x}\}\phi$$

$\text{flow}^{**}_{\text{os}}(\alpha, R_1, R_2, N_2)$ involves four states, the two pre-states s_1, s_1' and the post-states s_2, s_2'. Correspondingly, $\phi_{\alpha,R_1,R_2,N_2}$ contains four pairs of universally quantified variables $(h_1, \bar{x}_1)$, $(h_1', \bar{x}_1')$, $(h_2, \bar{x}_2)$, $(h_2', \bar{x}_2')$ representing the states s_1, s_1', s_2, s_2'. This leads to the following schematic form of $\phi_{\alpha,R_1,R_2,N_2}$ which directly follows the definition of $\text{flow}^{**}_{\text{os}}(\alpha, R_1, R_2, N_2)$:

$$\begin{aligned}
\phi_{\alpha,R_1,R_2,N_2} \equiv \forall h_1, h_1', h_2, h_2'.\forall \bar{x}_1, \bar{x}_1', \bar{x}_2, \bar{x}_2'.\\
(\quad (\circledast in\ s_1 \circledast)\langle\alpha\rangle(\circledast save\ s_2 \circledast)\\
\wedge (\circledast in\ s_1' \circledast)\langle\alpha\rangle(\circledast save\ s_2' \circledast)\\
\wedge (\circledast in\ s_1 \circledast)R_1 \doteq (\circledast in\ s_1' \circledast)R_1\\
\rightarrow (\quad newIso\\
\wedge (\quad (\circledast in\ s_2 \circledast)N_2 \doteq (\circledast in\ s_2' \circledast)N_2\\
\rightarrow (\circledast in\ s_2 \circledast)R_2 \doteq (\circledast in\ s_2' \circledast)R_2)))
\end{aligned}$$

To maintain readability, $\phi_{\alpha,R_1,R_2,N_2}$ uses the same abbreviations $(\!*\ldots*\!)$ as Theorem 9:

$$\begin{array}{llll}(\!*in\ s_i*\!) \equiv & \{\mathtt{heap} := h_i \mid\mid \bar{\mathtt{x}} := \bar{x}_i\} & (\!*save\ s_2*\!) \equiv & (\mathtt{heap} = h_2 \wedge \bar{\mathtt{x}} = \bar{x}_2)\\ (\!*in\ s_i'*\!) \equiv & \{\mathtt{heap} := h_i' \mid\mid \bar{\mathtt{x}} := \bar{x}_i'\} & (\!*save\ s_2'*\!) \equiv & (\mathtt{heap} = h_2' \wedge \bar{\mathtt{x}} = \bar{x}_2')\end{array}$$

The abbreviation $newIso$ is defined as

$$\begin{aligned} newIso \equiv \quad & (\!*in\ s_1*\!)objectsNew((\!*in\ s_2*\!)N_2)\\ & \wedge (\!*in\ s_1'*\!)objectsNew((\!*in\ s_2'*\!)N_2)\\ & \wedge Agree_{type}((\!*in\ s_2*\!)N_2, (\!*in\ s_2'*\!)N_2)\\ & \wedge Agree_{obj}((\!*in\ s_2*\!)N_2, (\!*in\ s_2'*\!)N_2)\end{aligned}$$

where the predicates $objectsNew(Seq\ X)$, $Agree_{type}(Seq\ X, Seq\ X')$ as well as $Agree_{obj}(Seq\ X, Seq\ X')$ are defined recursively as follows:

$$objectsNew(Seq\ X) \equiv \forall i.(0 \leq i < X.\mathbf{len} \rightarrow created(X[i]) \doteq FALSE)$$

$$\begin{aligned} & Agree_{type}(Seq\ X, Seq\ X') \equiv\\ & \quad X.\mathbf{len} \doteq X'.\mathbf{len}\\ & \wedge \forall i.(0 \leq i < X.\mathbf{len} \rightarrow\\ & \qquad X[i] \sqsubseteq Obj\\ & \qquad \wedge \bigwedge_{A \text{ in } \alpha} (exactInstance_A(X[i]) \leftrightarrow exactInstance_A(X'[i])))\end{aligned}$$

$$\begin{aligned} & Agree_{obj}(Seq\ X, Seq\ X') \equiv\\ & \quad \forall i.\forall j.(0 \leq i < X.\mathbf{len} \wedge 0 \leq j < X.\mathbf{len} \rightarrow (X[i] \doteq X[j] \leftrightarrow X'[i] \doteq X'[j]))\end{aligned}$$

As before, $R[i]$ denotes $seqGet_{Any}(R, i)$ and $t \sqsubseteq A$ denotes $instance_A(t)$.

It remains to show that $\phi_{\alpha,R_1,R_2,N_2}$ indeed expresses $\text{flow}^{**}_{\text{os}}(\alpha, R_1, R_2, N_2)$.

Theorem 30. *Let α be a program, let R_1, R_2, N_2 be observation expressions.*

$$flow^{**}_{\text{os}}(\alpha, R_1, R_2, N_2) \quad \Leftrightarrow \quad s_1 \models \phi_{\alpha,R_1,R_2,N_2}.$$

Proof.

"$\Leftarrow$": Assume $s_1 \models \phi_{\alpha,R_1,R_2,N_2}$. In order to prove $\text{flow}^{**}_{\text{os}}(\alpha, R_1, R_2, N_2)$, let s_1, s_1', s_2, s_2' be states such that α started in s_1 terminates in s_2, α started in s_1' terminates in s_2', and $\text{agree}_{\text{os}}(R_1, s_1, s_1', id)$. We need to show that all objects in $Obj(N_2^{s_2})$ and $Obj(N_2^{s_2'})$ are new, $\text{agree}_{\text{os}}(N_2, s_2, s_2', \pi)$ for a partial isomorphism π and if $\text{agree}_{\text{os}}(N_2, s_2, s_2', id)$ then $\text{agree}_{\text{os}}(R_2, s_2, s_2', id)$.

As in Theorem 9, we chose the variable assignment β such that $\beta(h_1) = \texttt{heap}^{s_1}$, $\beta(h_1') = \texttt{heap}^{s_1'}$, $\beta(h_2) = \texttt{heap}^{s_2}$, $\beta(h_2') = eap^{s_2'}$, $\beta(\bar{x}_1) = \bar{\texttt{x}}^{s_1}$, $\beta(\bar{x}_1') = \bar{\texttt{x}}^{s_1'}$, $\beta(\bar{x}_2) = \bar{\texttt{x}}^{s_2}$, and $\beta(\bar{x}_2') = \bar{\texttt{x}}^{s_2'}$. Then $\mathcal{D}, s, \beta \models (\!*in\ s_1*\!)\langle\alpha\rangle(\!*save\ s_2*\!)$ holds by Lemma 8 and by the definition of $\langle\alpha\rangle$ (for an arbitrary state s), because α started in s_1 terminates in s_2. Similarly, $\mathcal{D}, s, \beta \models (\!*in\ s_1'*\!)\langle\alpha\rangle(\!*save\ s_2'*\!)$ holds, because α started in s_1' terminates in s_2'. By Corollary 1, $\text{agree}_{\text{os}}(R_1, s_1, s_1', id)$ implies $\mathcal{D}, s, \beta \models (\!*in\ s_1*\!)R_1 \doteq (\!*in\ s_1'*\!)R_1$. As a consequence

$$\begin{aligned}\mathcal{D}, s, \beta \models \quad & newIso \\ \wedge\ (\quad & (\!*in\ s_2*\!)N_2 \doteq (\!*in\ s_2'*\!)N_2 \\ \rightarrow\ & (\!*in\ s_2*\!)R_2 \doteq (\!*in\ s_2'*\!)R_2)\end{aligned}$$

holds by assumption $s_1 \models \phi_{\alpha,R_1,R_2,N_2}$.

Another application of Corollary 1 on

$$\mathcal{D}, s, \beta \models (\!*in\ s_2*\!)N_2 \doteq (\!*in\ s_2'*\!)N_2 \rightarrow (\!*in\ s_2*\!)R_2 \doteq (\!*in\ s_2'*\!)R_2)$$

yields that $\text{agree}_{\text{os}}(N_2, s_2, s_2', id)$ implies $\text{agree}_{\text{os}}(R_2, s_2, s_2', id)$.

It remains to be shown that all objects in $Obj(N_2^{s_2})$ and $Obj(N_2^{s_2'})$ are new and $\text{agree}_{\text{os}}(N_2, s_2, s_2', \pi)$ for a partial isomorphism π. We know $\mathcal{D}, s, \beta \models newIso$ which is equivalent to:

$$\begin{aligned}\mathcal{D}, s, \beta \models \quad & (\!*in\ s_1*\!)objectsNew((\!*in\ s_2*\!)N_2) \\ \wedge\ & (\!*in\ s_1'*\!)objectsNew((\!*in\ s_2'*\!)N_2) \\ \wedge\ & Agree_{type}((\!*in\ s_2*\!)N_2, (\!*in\ s_2'*\!)N_2) \\ \wedge\ & Agree_{obj}((\!*in\ s_2*\!)N_2, (\!*in\ s_2'*\!)N_2)\end{aligned}$$

$\mathcal{D}, s, \beta \models (\!*in\ s_1*\!)objectsNew((\!*in\ s_2*\!)N_2)$ implies $Obj(N_2^{s_2})$, whereas $\mathcal{D}, s, \beta \models (\!*in\ s_1'*\!)objectsNew((\!*in\ s_2'*\!)N_2)$ implies $Obj(N_2^{s_2'})$. Finally, the combination of

$$\mathcal{D}, s, \beta \models Agree_{type}((\!*in\ s_2*\!)N_2, (\!*in\ s_2'*\!)N_2)$$

and

$$\mathcal{D}, s, \beta \models Agree_{obj}((\!*in\ s_2*\!)N_2, (\!*in\ s_2'*\!)N_2)$$

implies that s_2 and s_2' are isomorphic according to Definition 17 (Partial isomorphism w. r. t. R).

In total $\text{flow}_{\text{os}}^{**}(\alpha, R_1, R_2, N_2)$ has been shown.

"$\Rightarrow$": Assume $\text{flow}_{\text{os}}^{**}(\alpha, R_1, R_2, N_2)$. We need to show $\mathcal{D}, s, \beta \models \phi_{\alpha,R_1,R_2,N_2}$. To this end, let β be an arbitrary variable assignment and let s_1, s_1', s_2, s_2' be states such that $\beta(h_1) = \texttt{heap}^{s_1}$, $\beta(h_1') = \texttt{heap}^{s_1'}$, $\beta(h_2) = \texttt{heap}^{s_2}$, $\beta(h_2') =$

$eap^{s_2'}$, $\beta(\bar{x}_1) = \bar{\mathrm{x}}^{s_1}$, $\beta(\bar{x}_1') = \bar{\mathrm{x}}^{s_1'}$, $\beta(\bar{x}_2) = \bar{\mathrm{x}}^{s_2}$, and $\beta(\bar{x}_2') = \bar{\mathrm{x}}^{s_2'}$. We may assume (for an arbitrary state s) $\mathcal{D}, s, \beta \models (\!*in\, s_1*\!)\langle\alpha\rangle(\!*save\, s_2*\!)$, $\mathcal{D}, s, \beta \models (\!*in\, s_1'*\!)\langle\alpha\rangle(\!*save\, s_2'*\!)$ and $\mathcal{D}, s, \beta \models (\!*in\, s_1*\!)R_1 \doteq (\!*in\, s_1'*\!)R_1$ and need to show that

$$\begin{aligned}\mathcal{D}, s, \beta \models \quad & newIso \\ \wedge\, (\quad & (\!*in\, s_2*\!)N_2 \doteq (\!*in\, s_2'*\!)N_2 \\ \rightarrow\, & (\!*in\, s_2*\!)R_2 \doteq (\!*in\, s_2'*\!)R_2)\end{aligned}$$

holds.

$\mathcal{D}, s, \beta \models (\!*in\, s_1*\!)\langle\alpha\rangle(\!*save\, s_2*\!)$, $\mathcal{D}, s, \beta \models (\!*in\, s_1'*\!)\langle\alpha\rangle(\!*save\, s_2'*\!)$ and $\mathcal{D}, s, \beta \models (\!*in\, s_1*\!)R_1 \doteq (\!*in\, s_1'*\!)R_1$ imply that α started in s_1 terminates in s_2, α started in s_1' terminates in s_2', and $\mathrm{agree}_{\mathrm{os}}(R_1, s_1, s_1', id)$. Thus, $\mathrm{flow}^{**}_{\mathrm{os}}(\alpha, R_1, R_2, N_2)$ implies that all objects in $Obj(N_2^{s_2})$ and $Obj(N_2^{s_2'})$ are new, $\mathrm{agree}_{\mathrm{os}}(N_2, s_2, s_2', \pi)$ for a partial isomorphism π and if $\mathrm{agree}_{\mathrm{os}}(N_2, s_2, s_2', id)$ then $\mathrm{agree}_{\mathrm{os}}(R_2, s_2, s_2', id)$. Analog to the first part of the proof,

$$\mathcal{D}, s, \beta \models (\!*in\, s_2*\!)N_2 \doteq (\!*in\, s_2'*\!)N_2 \rightarrow (\!*in\, s_2*\!)R_2 \doteq (\!*in\, s_2'*\!)R_2)$$

holds because $\mathrm{agree}_{\mathrm{os}}(N_2, s_2, s_2', id)$ implies $\mathrm{agree}_{\mathrm{os}}(R_2, s_2, s_2', id)$. Similarly,

$$\mathcal{D}, s, \beta \models newIso$$

because all objects in $Obj(N_2^{s_2})$ and $Obj(N_2^{s_2'})$ are new and $\mathrm{agree}_{\mathrm{os}}(N_2, s_2, s_2', \pi)$ for a partial isomorphism π. In total we have shown $\mathcal{D}, s, \beta \models \phi_{\alpha, R_1, R_2, N_2}$. □

The next section shows how strong object-sensitive noninterference is specified in the JML extension from Chapter 4.

6.4 JML Extension

The information flow behavior of methods is specified in JML with the help of determines clauses, as presented in Chapter 4.3. This section introduces the new keyword `\new_objects` which can be used in determines clauses to name the observable newly created objects. The syntax of determines clauses is defined by the Extended Backus–Naur Form (EBNF) of Listing 6.1. Here, `expression` is a usual JML expression.

The semantics of the determines clause is defined with the help of conditional object-sensitive non-interference, the obvious extension of Definition 25 with conditions along the lines of Definition 3: Let R_{post} be defined as the concatenation of the expressions behind the **`determines`** keyword and the expressions behind the **`\erases`** keywords. Let R_{pre} be defined as the concatenation

```
determines_clause =
  "determines", ( expressions | "\nothing" ),
  "\by",        ( expressions | "\itself" | "\nothing" ),
  {   [ "\declassifies", expressions ]
    | [ "\erases",       expressions ]
    | [ "\new_objects",  expressions ] };
expressions = expression, { ",", expression };
```

Listing 6.1: EBNF of `determines` clauses in the object-sensitive context.

of the expressions behind the **\by** keyword and the expressions behind the **\declassifies** keywords and let N_2 be defined as the concatenation of the expressions behind the `\new_objects` keywords. In this context **\nothing** is identified with the empty sequence and **\itself** is identified with R_{post}. Let further ϕ_{pre} be the precondition of the contract defined as usual by requires clauses and class invariants. A method `m` fulfills a determines clause if and only if $\text{flow}_{\text{os}}(\texttt{m}, R_{pre}, R_{post}, N_2, \phi_{pre})$ is valid.

The syntax and semantics of block contracts and loop invariants of Chapter 4.3 are extended accordingly.

Example 1. *Consider the following example originating from Naumann [2006]. In this example a fresh array with 10 fresh* `Node` *objects is generated. Additionally, the* `val` *attribute of each* `Node` *object is assigned the value of the low parameter* `x`*. It has to be shown that the resulting array does not depend on the secret parameter* `secret`*.*

```
1  public class Naumann {
2      Node[] m_result;
3
4      /*@ determines  m_result,
5                      (\seq_def int i; 0; m_result.length; m_result[i]),
6                      (\seq_def int i; 0; m_result.length;
7                         m_result[i].val)
8                 \by  x;
9        @*/
10     //@ helper
11     void  pair_m(int x, int secret) {
12         /*@ normal_behavior
13             ensures     m_result != null && m_result.length == 10;
14             ensures     \typeof(m_result) == \type(Node[]);
15             determines      m_result \by \nothing
16               \new_objects  m_result;
17           @*/
```

```
18          { m_result = new Node[10]; }
19          int i = 0;
20          /*@ loop_invariant 0 <= i && i <= m_result.length;
21              loop_invariant     m_result != null
22                              && \typeof(m_result) == \type(Node[]);
23              assignable  m_result[*];
24              decreases   m_result.length - i;
25              determines      i, x, m_result,
26                              (\seq_def int j; 0; i; m_result[j]),
27                              (\seq_def int j; 0; i; m_result[j].val)
28                \by           \itself
29                \new_objects  m_result[i-1];
30            @*/
31          while (i < 10) {
32              m_result[i] = new Node();
33              m_result[i].val = x;
34              i++;
35          }
36      }
37
38      class Node {
39          public int val;
40      }
41
42  }
```

The security requirement is expressed by a method contract. The contract states that the attribute `m_result` *as well as its content depends at most on the value of the parameter* `x` *(and therefore not on the value of* `secret`*). Though the method generates new objects, the method contract does not use the keyword* `\new_objects`*. This is possible, because the verification problem is split into two parts: the generation of the new array object and the loop. In the first part as well as in each loop iteration only one new object is created. This makes it easy to find an isomorphism for the newly created objects of each part. Further, by the compositionality result of Section 6.2, we may assume at the end of each part that identical objects have been created. Because this holds in particular after the execution of the loop, the method contract does not need to list the new objects again, but can check for object identity instead. Intuitively, each part checks for a compatible extension of the isomorphism of the previous part and therefore we do not need to check for the existence of an isomorphism for the composition of those parts again. This fact simplifies the verification of object-sensitive noninterference considerably.*

The block contract for the array creation, line 18, ensures (beside some basic functional guarantees) that the final value of `m_result` *does not depend on anything and that*

the final value is a newly created object. Similarly, the loop invariant states that the values of control variable `i`*, of the parameter* `x`*, of* `m_result` *and of the content of* `m_result` *up to position* `i` *depend in each loop iteration only on themselves. Further, in each loop iteration the only new observable object is* `m_result[i-1]` *(where the expression* `m_result[i-1]` *is evaluated in the poststate). Beside these information flow guarantees, the loop invariant ensures termination of the loop and that at most the content of* `m_result` *is modified.*

6.5 Discussion

The discussion below focuses on object-sensitive secure information flow. A broader discussion on the specification and verification of secure information flow can be found in Sections 4.5 and 5.4.

The work closest to this chapter is Amtoft et al. [2006]. The authors build on *region logic*, a kind of Hoare logic with concepts from separation logic, which is comparable to Java DL. They use the same basic definition of object-sensitive secure information flow. Instead of providing verification conditions which can be discharged with a standard calculus, as presented in this chapter, they introduce a specialized, more efficient calculus in the lines of Chapter 7 to show object-sensitive secure information flow. This specialized calculus uses approximate rules which avoid explicit modeling of isomorphisms, but comes with the price of imprecision. The discerning points of the presented work are: (1) a further investigation of the security property, allowing the restriction of isomorphisms as far as possible and thus making the explicit, non approximate modeling of isomorphisms feasible with a minimum of additional user interaction; (2) verification conditions that are discharged with an existing tool; and (3) a more flexible specification methodology.

Contributions (1) and (3) also distinguish this work from the approaches mentioned in Section 5.4, including JIF, which already presented an approximate treatment of object-sensitive secure information flow for Java in Myers [1999a]. JIF is a practical approach to the analysis of secure information flow which covers a broad range of language features, but it has not been formally proven to enforce noninterference. Similar to JIF, Barthe et al. [2013] and Banerjee and Naumann [2002] use type systems for the verification of object-oriented secure information flow. They treat a smaller set of language features, but prove that their type systems indeed enforce noninterference. A closely related approach is Beringer and Hofmann [2007]. Here, only the information flow analysis is based on type systems; the verification task is separated from the analysis and based on program logics. Still, points (1) and (3) as well as the overall precision

are discerning points of this chapter. The approach by Barthe et al. [2013], already mentioned above, and the approaches by Hansen and Probst [2006] and Hedin and Sands [2005] target Java Bytecode in contrast to source code, as the other approaches do. Hedin and Sands [2005] use a type system approach, too, whereas Hansen and Probst [2006] use abstract interpretation in combination with classical static analysis.

To the best of the authors knowledge, the only approach which models isomorphisms explicitly is the self-composition approach in Naumann [2006]. The drawback of that approach is that the specifier needs to track the isomorphism manually with the help of additional ghost code annotations. This increases the burden on the specifier, whereas the presented approach detects the isomorphism automatically.

7 An Approximate Information Flow Calculus

Many times programs contain simple parts which *can* be verified with self-composition style reasoning, but which can be verified more efficiently with approximate approaches, even though the optimizations of Sections 5.2 and 5.3 increase the efficiently of self-composition style reasoning considerably. This section presents an approximate information flow calculus along the lines of Amtoft et al. [2006] which can handle simple examples efficiently and—in contrast to Amtoft et al. [2006]—additionally provides the possibility to resort to self-composition style reasoning, if higher precision is needed at any time during a proof.

As in Amtoft et al. [2006], the calculus is based on a special "agree" predicate $\ltimes : Any$. Formulas and sequents are interpreted in two states instead of one: $\ltimes(t)$ is valid in states s_1 and s_2 if t^{s_1} equals t^{s_2}. Information flow statements $\text{flow}(\alpha, R_1, R_2, \phi)$ can be expressed with the help of $\ltimes$ by

$$(\phi \wedge \ltimes(R_1)) \rightarrow [\alpha]\ltimes(R_2)\,, \tag{7.1}$$

where ϕ is a "functional precondition", that is, a formula which neither contains the $\ltimes$ operator nor modalities. Intuitively, formula (7.1) reads as follows: if ϕ is valid in two states s_1 and s_1' and if s_1 and s_1' agree on the value of R_1, then any two states reached from s_1 and s_1' by execution of α, respectively, agree on the value of R_2.

The information flow statement

$$\text{flow}(\texttt{l1=l2; if (false) l2=h}, \langle \texttt{l1}, \texttt{l2}\rangle, \langle \texttt{l1}, \texttt{l2}\rangle, \mathit{true}) \tag{7.2}$$

for instance holds if and only if the sequent

$$\ltimes(\texttt{l1}), \ltimes(\texttt{l2}) \Longrightarrow [\texttt{l1=l2; if (false) l2=h}](\ltimes(\texttt{l1}) \wedge \ltimes(\texttt{l2})) \tag{7.3}$$

is universally valid. Figure 7.1 shows an example derivation in the approximate calculus which proves that (7.3) is indeed universally valid. The rule names in Figure 7.1 are abbreviated according to Table 7.1. We will go shortly trough the

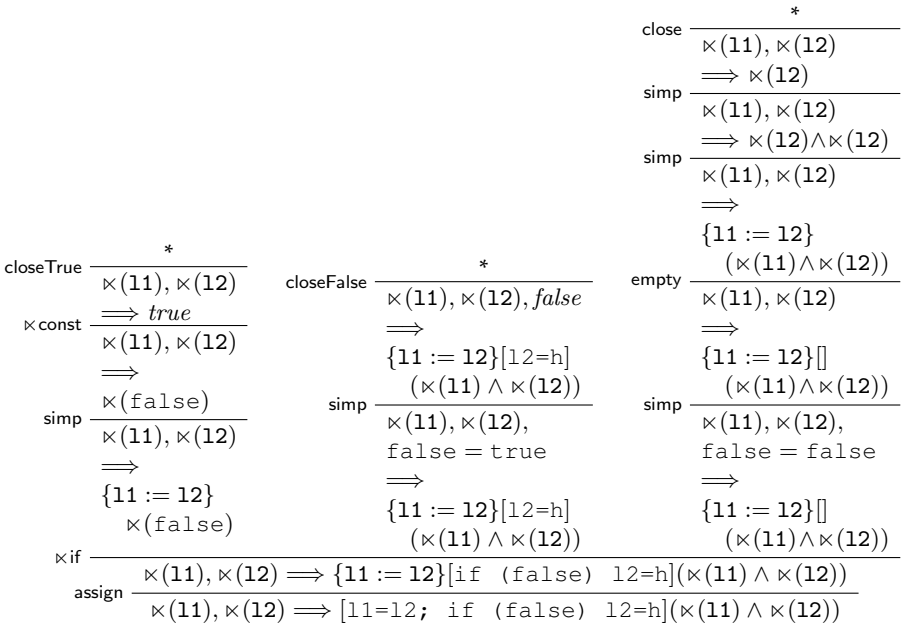


Figure 7.1: Two-state calculus: example derivation.

Abbreviation	Full name	Abbreviation	Full name
assign	assignLocal	⋉if	⋉conditional
empty	emptyModality	⋉const	⋉constant
closeTrue	closeTrue	closeFalse	closeFalse
simp	combination of all update simplification rules	close	close

Table 7.1: Two-state calculus: abbreviations of rule names.

proof tree and explain the intuition behind each derivation step to get a feeling for the calculus.

The first derivation is a symbolic execution step which transforms the assignment `l1=l2` as usual into the update $\{\texttt{l1} := \texttt{l2}\}$. The second step executes the `if` statement symbolicly. In contrast to the normal if rule, the $\ltimes$if rule has three premisses:

(1) In premiss one it has to be shown that the two states, in which the `if` statement is executed, evaluate the guard in the same way. This way it is ensured that any two runs which agree on `l1` and `l2` in the prestate will execute the same branch of the `if` statement.

(2) In premiss two it has to be shown that the postcondition holds if the runs take the `then` branch, whereas

(3) in premiss three it has to be shown that the postcondition holds if the runs take the `else` branch.

Because `false` is a constant, $\ltimes(\texttt{false})$ is universally valid and Branch 1 of the proof tree closes almost immediately. The then branch of the `if` statement is never executed, because its guard is `false`. Therefore, Branch 2 of the proof tree closes easily, too. Branch 3 remains. Here, the rule "simp" removes the tautology $\texttt{false} = \texttt{false}$ from the sequent. Thereafter, the empty modality is removed by the rule "empty" and the update $\{\texttt{l1} := \texttt{l2}\}$ is applied to the post condition $\ltimes(\texttt{l1}) \land \ltimes(\texttt{l2})$, again by the rule "simp". The resulting sequent is obviously universally valid and thus Branch 3 closes within two additional steps, too.

The remainder of the chapter is structured as follows. The next section introduces the extension to the Java DL syntax from Section 2.1, whereas Section 7.2 introduces its one-state and two-state semantics. Finally, Section 7.3 presents the calculus followed by its soundness proof in Section 7.4. The chapter closes with a short discussion.

7.1 Java DL Syntax Extension

Slightly different from the intuition given in the introduction, the syntax of Java DL from Section 2.1 is not augmented by a special predicate, but by a new modal operator $\ltimes$. Apart from that, terms and formulas are defined recursively as before, whereby nesting of $\ltimes$ is explicitly allowed.

The notation $\ltimes(R)$ from the introduction, where R is an observation expression or a term (and not a formula), is a shorthand for the formula $\forall x.\ltimes(x = R)$, where x is a variable of the same type as R which does not occur free in R. As

Lemma 32 below shows, the shorthand $\ltimes(R)$ has the intended meaning, that is, $\ltimes(R)$ is valid in states s_1 and s_2 if and only if R^{s_1} equals R^{s_2}.

$$(\phi \wedge \ltimes(R_1)) \rightarrow [\alpha](\psi \wedge \ltimes(R_2)) \tag{7.4}$$

and Formula (7.1) are syntactically correct formulas in the extended Java DL syntax using the shorthands $\ltimes(R_i)$ for the formulas $\forall x.\ltimes(x = R_i)$. The next section explains their formal semantics.

7.2 Two-State Semantics

Formulas ϕ containing the $\ltimes$ operator are interpreted in pairs of states. The two-state semantics of a formula is defined on the basis of a restricted two-state evaluation which interprets ϕ mainly in the first state.

Definition 31 (Restricted Two-State Evaluation)**.** *A formula ϕ is true in restricted two-state evaluation in two-state structure $(\mathcal{D}, s_1, s_2)$ and variable assignment β, donated by $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi$, if and only if:*

$$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \Leftrightarrow \begin{cases} \mathcal{D}, s_1, \beta \vDash \phi & \text{if the top-level operator of } \phi \text{ is a predicate} \\ \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1 \text{ if and only if } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi_1 & \text{if } \phi = \ltimes\phi_1 \\ \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_2 & \text{if } \phi = \phi_1 \wedge \phi_2 \\ \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1 \text{ or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_2 & \text{if } \phi = \phi_1 \vee \phi_2 \\ \text{not } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1 & \text{if } \phi = \neg\phi_1 \\ \mathcal{D}, s_1, s_2, \beta^{x/d} \vDash_{2\downarrow 1} \phi_1 \text{ for all } d \in D & \text{if } \phi = \forall x.\phi_1 \\ \text{There is an element } d \in D \text{ such that } \mathcal{D}, s_1, s_2, \beta^{x/d} \vDash_{2\downarrow 1} \phi_1 & \text{if } \phi = \exists x.\phi_1 \\ \mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2\downarrow 1} \phi_1 \text{ where } s_i^u = val_{\mathcal{D}, s_i, \beta}(u)(s_i) & \text{if } \phi = \{u\}\phi_1 \\ \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi_1 \text{ for all } s_1^\alpha, s_2^\alpha \text{ such that } \alpha \text{ started in } s_i \text{ terminates (normally) in } s_i^\alpha & \text{if } \phi = [\alpha]\phi_1 \\ \text{There are } s_1^\alpha, s_2^\alpha \text{ such that } \alpha \text{ started in } s_i \text{ terminates (normally) in } s_i^\alpha \text{ and } \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi_1 & \text{if } \phi = \langle\alpha\rangle\phi_1 \end{cases}$$

$\mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \phi$ denotes the case that $\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi$ does not hold.

For conciseness of notation, in the following, $\mathcal{D}, s_1, s_2, \beta \models \phi$ will sometimes be abbreviated by $s_1, s_2 \models \phi$. In these cases $\mathcal{D}$ and β are arbitrary but fixed. Similarly, t^s will be used to abbreviate $t^{\mathcal{D},s,\beta}$. In other cases, the variable assignment β is not important for the evaluation of formulas or terms (for instance if closed formulas are evaluated). In these cases β will be omitted.

The next lemma shows that the abbreviation $\ltimes(t)$ (where t is a term and not a formula) has the semantics motivated by the introduction.

Lemma 32. *Let $(\mathcal{D}, s_1, s_2)$ be a two-state structure, let β be a variable assignment, let t be a term of type T and let x be a variable of type T which does not occur free in t.*

$\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \ltimes(t)$ (which abbreviates $\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \forall x.\ltimes(x = t)$) holds if and only if $t^{\mathcal{D},s_1,\beta} = t^{\mathcal{D},s_2,\beta}$.

Proof. By Definition 31, $\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \forall x.\ltimes(x = t)$ holds if and only if $\mathcal{D}, s_1, s_2, \beta^{x/d} \models_{2\downarrow 1} \ltimes(x = t)$ for all $d \in T^{\mathcal{D}}$. Again by Definition 31, the latter is equivalent to $\mathcal{D}, s_1, \beta^{x/d} \models x = t$ iff $\mathcal{D}, s_2, \beta^{x/d} \models x = t$ for all $d \in T^{\mathcal{D}}$. By the usual Java DL semantics and because x does not occur free in t this holds if and only if $d = t^{\mathcal{D},s_1,\beta}$ iff $d = t^{\mathcal{D},s_2,\beta}$ for all $d \in T^{\mathcal{D}}$. Because there is a $d \in T^{\mathcal{D}}$ such that $d = t^{\mathcal{D},s_1,\beta}$, this is equivalent to $t^{\mathcal{D},s_1,\beta} = t^{\mathcal{D},s_2,\beta}$. □

To illustrate Definition 31, let us consider Formula (7.1) again. Formula (7.1) is valid in states s_1 and s_2 with respect to the restricted two-state evaluation, written $s_1, s_2 \models_{2\downarrow 1} (\phi \wedge \ltimes(R_1)) \rightarrow [\alpha]\ltimes(R_2)$, if and only if

	$s_1, s_2 \models_{2\downarrow 1} \phi$ (which is equivalent to $s_1 \models \phi$, because ϕ is a "functional precondition", see Lemma 49.2 below)
and	$R_1^{s_1} = R_1^{s_2}$
imply	for all states s_1^α, s_2^α such that α started in s_1 terminates in s_1^α and α started in s_2 terminates in s_2^α the equation $R_2^{s_1^\alpha} = R_2^{s_2^\alpha}$ is true.

This is not yet compatible with the semantics of flow(α, R_1, R_2, ϕ) as given in Definition 3, because flow(α, R_1, R_2, ϕ) has the additional precondition that beside $s_1 \models \phi$ also $s_2 \models \phi$ holds. Therefore, the two-state semantics is defined on the basis of the restricted two-state evaluation as follows.

Definition 33 (Two-State Evaluation)**.** *A formula ϕ evaluates to tt in two-state evaluation in two-state structure $(\mathcal{D}, s_1, s_2)$ and variable assignment β, denoted by $\mathcal{D}, s_1, s_2, \beta \models_2 \phi$, if and only if*

$$(\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi \text{ or } \mathcal{D}, s_2, s_1, \beta \models_{2\downarrow 1} \phi) \text{ and } \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \phi$$

holds. $\mathcal{D}, s_1, s_2, \beta \not\models_2 \phi$ denotes the case that $\mathcal{D}, s_1, s_2, \beta \models_2 \phi$ does not hold.

Definition 34 (Two-State Model)**.** *A two-state structure $(\mathcal{D}, s_1, s_2)$ is called a two-state model of a formula ϕ, denoted by $\mathcal{D}, s_1, s_2 \vDash_2 \phi$, if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_2 \phi$ for all variable assignments β. $\mathcal{D}, s_1, s_2 \nvDash_2 \phi$ denotes the case that $\mathcal{D}, s_1, s_2 \vDash_2 \phi$ does not hold. A two-state structure $(\mathcal{D}, s_1, s_2)$ is called a two-state model for a set of formulas M, denoted by $\mathcal{D}, s_1, s_2 \vDash_2 M$, if and only if $\mathcal{D}, s_1, s_2 \vDash_2 \phi$ for all $\phi \in M$.*

Definition 35 (Universal Validity in Two-State Semantics)**.** *A formula ϕ is universally valid in two-state semantics, denoted by $\vDash_2 \phi$, if and only if $\mathcal{D}, s_1, s_2 \vDash_2 \phi$ for all two-state structures $(\mathcal{D}, s_1, s_2)$.*

Note that $\vDash_2$ has several non-classical properties, for instance, $\vDash_2$ is *not* closed under conjunction: let $\mathtt{x}$ be a program variable of type Boolean and let $s_1 : \mathtt{x} \mapsto \mathtt{true}$ and $s_2 : \mathtt{x} \mapsto \mathtt{false}$ be two states. Then $s_1, s_2 \vDash_2 \mathtt{x} = \mathtt{true} \rightarrow \ltimes(\mathtt{x})$ and $s_1, s_2 \vDash_2 \mathtt{x} = \mathtt{false} \rightarrow \ltimes(\mathtt{x})$ hold, but $s_1, s_2 \vDash_2 (\mathtt{x} = \mathtt{true} \rightarrow \ltimes(\mathtt{x})) \wedge (\mathtt{x} = \mathtt{false} \rightarrow \ltimes(\mathtt{x}))$ does not. Indeed, the exemplary derivation of $\ltimes(\mathtt{x})$ in Section 7.3 shows that it appears to be unlikely that there is a semantics for $\vDash_2$ such that $\ltimes$ is interpreted as desired and at the same time $\vDash_2$ has all properties of classical logic.

That the semantics of $\vDash_2$ is chosen reasonably is shown by the next lemma.

Lemma 36. *Let ϕ be a closed formula which neither contains the $\ltimes$ operator nor modalities ("a functional precondition"), let α be a program and let R_1, R_2 be observation expressions.*

$\phi \wedge \ltimes(R_1) \rightarrow [\alpha]\ltimes(R_2)$ is universally valid in two-state semantics if and only if flow(α, R_1, R_2, ϕ) holds.

Proof. Let $(\mathcal{D}, s_1, s_2)$ be an arbitrary two-state structure. By Definition 34 (Two-State Model) and Definition 33 (Two-State Evaluation),

$$\mathcal{D}, s_1, s_2 \vDash_2 (\phi \wedge \ltimes(R_1)) \rightarrow [\alpha]\ltimes(R_2)$$

holds if and only if

$$\begin{aligned} (\quad & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} (\phi \wedge \ltimes(R_1)) \rightarrow [\alpha]\ltimes(R_2) \\ \text{or } & \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} (\phi \wedge \ltimes(R_1)) \rightarrow [\alpha]\ltimes(R_2)\,) \\ \text{and } & \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} (\phi \wedge \ltimes(R_1)) \rightarrow [\alpha]\ltimes(R_2) \end{aligned} \tag{7.5}$$

holds for all variable assignments β. Because the observation expressions R_1 and R_2 are closed formulas and because ϕ is closed by the premiss of the lemma, β can be omitted. By Definition 31 (Restricted Two-State Evaluation) the last conjunct $\mathcal{D}, s_1, s_1 \vDash_{2\downarrow 1} (\phi \wedge \ltimes(R_1)) \rightarrow [\alpha]\ltimes(R_2)$ is equivalent to

> $\mathcal{D}, s_1, s_1 \nvDash_{2\downarrow 1} \phi$ or $R_1^{\mathcal{D}, s_1} \neq R_1^{\mathcal{D}, s_1}$ or for all states s_1^α such that α started in s_1 terminates in s_1^α the equation $R_2^{\mathcal{D}, s_1^\alpha} = R_2^{\mathcal{D}, s_1^\alpha}$ holds.

This is always true, because $R_2^{\mathcal{D},s_1^\alpha}$ always equals itself. Hence, (7.5) is equivalent to

$$\begin{aligned} &\mathcal{D}, s_1, s_2 \vDash_{2\downarrow 1} (\phi \wedge \ltimes(R_1)) \rightarrow [\alpha]\ltimes(R_2) \\ \text{or } &\mathcal{D}, s_2, s_1 \vDash_{2\downarrow 1} (\phi \wedge \ltimes(R_1)) \rightarrow [\alpha]\ltimes(R_2) \ . \end{aligned} \tag{7.6}$$

Because $\mathcal{D}, s_1, s_2 \nvDash_{2\downarrow 1} \phi$ is equivalent to $\mathcal{D}, s_1 \nvDash \phi$ and $\mathcal{D}, s_2, s_1 \nvDash_{2\downarrow 1} \phi$ is equivalent to $\mathcal{D}, s_2 \nvDash \phi$, see Lemma 49.2, equation (7.6) can be rearranged by Definition 31 (Restricted Two-State Evaluation) to

$$\begin{aligned} &\mathcal{D}, s_1 \nvDash \phi \text{ or } R_1^{\mathcal{D},s_1} \neq R_1^{\mathcal{D},s_2} \text{ or for all states } s_1^\alpha, s_2^\alpha \text{ such that } \alpha \\ &\quad \text{terminates in } s_1^\alpha, s_2^\alpha \text{ if started in } s_1, s_2, \text{ respectively, the equation} \\ &\quad R_2^{\mathcal{D},s_1^\alpha} = R_2^{\mathcal{D},s_2^\alpha} \text{ holds} \\ \text{or } &\mathcal{D}, s_2 \nvDash \phi \text{ or } R_1^{\mathcal{D},s_1} \neq R_1^{\mathcal{D},s_2} \text{ or for all states } s_1^\alpha, s_2^\alpha \text{ such that } \alpha \\ &\quad \text{terminates in } s_1^\alpha, s_2^\alpha \text{ if started in } s_1, s_2, \text{ respectively, the equation} \\ &\quad R_2^{\mathcal{D},s_1^\alpha} = R_2^{\mathcal{D},s_2^\alpha} \text{ holds} \end{aligned} \tag{7.7}$$

which is equivalent to

$$\begin{aligned} &\mathcal{D}, s_1 \nvDash \phi \\ \text{or } &\mathcal{D}, s_2 \nvDash \phi \\ \text{or } &R_1^{\mathcal{D},s_1} \neq R_1^{\mathcal{D},s_2} \\ \text{or for all states } &s_1^\alpha, s_2^\alpha \text{ such that } \alpha \text{ terminates in } s_1^\alpha, s_2^\alpha \text{ if started in} \\ &s_1, s_2, \text{ respectively, } R_2^{\mathcal{D},s_1^\alpha} = R_2^{\mathcal{D},s_2^\alpha} \text{ holds.} \end{aligned} \tag{7.8}$$

Equation (7.8) matches exactly the semantics of flow(α, R_1, R_2, ϕ). □

The requirement $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \phi$ of Definition 33 is necessary to avoid inconsistency of the semantics, see Lemma 44 (Consistency of the Two-State Semantics), but has no impact on the interpretation of formulas expressing information flow.

Sequents $\Gamma \Longrightarrow \Delta$ are interpreted in two-state semantics by their meaning formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$.

Definition 37 (Validity of Sequents in Two-State Semantics)**.** *A sequent* $\Gamma \Longrightarrow \Delta$ *is valid in two-state structure* $(\mathcal{D}, s_1, s_2)$ *and variable assignment* β *if and only if* $\mathcal{D}, s_1, s_2, \beta \vDash_2 \bigwedge \Gamma \rightarrow \bigvee \Delta$.

Intuitively, the interpretation of sequents $\Gamma \Longrightarrow \Delta$ in two states s_1 and s_2 is as usual in the sense that the left hand side is connected by "and" and the right hand side is connected by "or": if $s_1, s_2 \vDash_{2\downarrow 1} \Gamma$ *and* $s_2, s_1 \vDash_{2\downarrow 1} \Gamma$ hold, then there exists $\delta \in \Delta$ such that $s_1, s_2 \vDash_{2\downarrow 1} \delta$ *or* $s_2, s_1 \vDash_{2\downarrow 1} \delta$ holds. As in Lemma 36, the

additional requirement $s_1, s_1 \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow \bigvee \Delta$ has no impact on the evaluation of the sequents that we are interested in.

However, formulas ϕ can be interpreted naturally in one state instead of two—even if they contain $\ltimes$—by giving them the semantics of $s_1, s_1 \vDash_2 \phi$. If $\ltimes$ is interpreted rigidly as *tt* in the (one-state) semantics from Section 2.1, then $s_1, s_1 \vDash_2 \phi$ and $s_1 \vDash \phi$ coincide (Lemma 39).

Definition 38. *In the (one-state) semantics from Section 2.1 $\ltimes$ is interpreted rigidly as tt, that is, all structures $(\mathcal{D}, s)$ are required to fulfill $\mathcal{D}, s \vDash \ltimes\phi$ for all formulas ϕ.*

Lemma 39. *$\mathcal{D}, s_1, s_1, \beta \vDash_2 \phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \phi$ holds.*

Proof. By Definition 33 (Two-State Evaluation) $\mathcal{D}, s_1, s_1, \beta \vDash_2 \phi$ if and only if $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \phi$. The lemma follows by structural induction on the structure of Java DL formulas.

Base case 1. By Definition 31 (Restricted Two-State Evaluation), $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \phi$ for any formula ϕ with a predicate symbol as top level symbol.

Base case 2. By Definition 31, $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \ltimes\phi$ holds if and only if

$$\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \phi \text{ iff } \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \phi\,.$$

Therefore, $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \ltimes\phi$ holds in all states s_1 and for all formulas ϕ.

Step case. The claim is quite obvious. The cases $\phi = \phi_1 \wedge \phi_2$ and $\phi = [\alpha]\phi_1$ are proved exemplary.

Case $\phi = \phi_1 \wedge \phi_2$. $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \phi_1 \wedge \phi_2$ holds if and only if $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \phi_1$ and $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \phi_2$ hold. By the induction hypothesis this is equivalent to $\mathcal{D}, s_1, \beta \vDash \phi_1$ and $\mathcal{D}, s_1, \beta \vDash \phi_2$, which again is equivalent to $\mathcal{D}, s_1, \beta \vDash \phi_1 \wedge \phi_2$.

Case $\phi = [\alpha]\phi_1$. $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} [\alpha]\phi_1$ holds if and only if $\mathcal{D}, s_1^\alpha, s_1^\alpha, \beta \vDash_{2\downarrow 1} \phi_1$ for all s_1^α such that α started in s_1 terminates in s_1^α. By the induction hypothesis this is equivalent to $\mathcal{D}, s_1^\alpha, \beta \vDash \phi_1$ for all s_1^α such that α started in s_1 terminates in s_1^α, which again is equivalent to $\mathcal{D}, s_1, \beta \vDash [\alpha]\phi_1$. □

Satisfyability, logical consequence and logical equivalence are defined in two-state semantics as usual:

Definition 40 (Satisfyability in Two-State Semantics)**.** *A formula ϕ is satisfyable in two-state semantics if and only if there exist a two-state structure $(\mathcal{D}, s_1, s_2)$ and a variable assignment β such that $\mathcal{D}, s_1, s_2, \beta \vDash_2 \phi$ holds.*

Definition 41 (Logical Consequence in Two-State Semantics)**.** *Let ϕ be a formula and let M be a set of formulas. $M \models_2 \phi$ if and only if for all two-state structures $(\mathcal{D}, s_1, s_2)$ and all variable assignments β the validity of $\mathcal{D}, s_1, s_2, \beta \models_2 M$ implies the validity of $\mathcal{D}, s_1, s_2, \beta \models_2 \phi$.*

Definition 42 (Logical Equivalence in Two-State Semantics)**.** *Two formulas ϕ and ψ are logical equivalent in two-state semantics, denoted by $\phi \equiv_2 \psi$ if and only if $\phi \leftrightarrow \psi$ is universal valid in two-state semantics.*

The two-state logic defined above enjoys the desirable property of consistency, as the following two lemmas show.

Lemma 43. *Let ϕ be a formula, let $(\mathcal{D}, s_1, s_2)$ be a two-state structure and let β be a variable assignment.*

$\mathcal{D}, s_1, s_2, \beta \models_2 \neg\phi$ implies $\mathcal{D}, s_1, s_2, \beta \not\models_2 \phi$.

Proof. Let ψ be an arbitrary formula. By Definition 33 (Two-State Evaluation), $\mathcal{D}, s_1, s_2, \beta \models_2 \psi$ if and only if

$$(\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \psi \text{ or } \mathcal{D}, s_2, s_1, \beta \models_{2\downarrow 1} \psi) \text{ and } \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \psi\,.$$

Therefore, $\mathcal{D}, s_1, s_2, \beta \models_2 \psi$ implies $\mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \psi$ which is equivalent to $\mathcal{D}, s_1, \beta \models \psi$ by Lemma 39.

Assume $\mathcal{D}, s_1, s_2, \beta \models_2 \neg\phi$ and $\mathcal{D}, s_1, s_2, \beta \models_2 \phi$. Then, by the thoughts above, also $\mathcal{D}, s_1, \beta \models \neg\phi$ and $\mathcal{D}, s_1, \beta \models \phi$ hold. This is a contradiction to the consistency of Java DL. □

Theorem 44 (Consistency of the Two-State Semantics)**.** *Let M be a set of formulas which has a two-state model and let ϕ be a formula. Then either $M \not\models_2 \phi$ or $M \not\models_2 \neg\phi$.*

Proof. By the assumption that M has a model, there exists a two-state structure $(\mathcal{D}, s_1, s_2)$ such that $\mathcal{D}, s_1, s_2 \models_2 M$. Assume $M \models_2 \neg\phi$. By Definition 41 (Logical Consequence in Two-State Semantics), $M \models_2 \neg\phi$ if and only if every two-state model of M is a two-state model of $\neg\phi$. Therefore, $\mathcal{D}, s_1, s_2 \models_2 \neg\phi$ needs to hold. By Lemma 43 this implies $\mathcal{D}, s_1, s_2 \not\models_2 \phi$ and hence $M \not\models_2 \phi$. □

The converse implication of Lemma 43, however, does not hold: there are a formula ϕ and states s_1, s_2 such that neither $s_1, s_2 \models_2 \phi$ nor $s_1, s_2 \models_2 \neg\phi$ holds. Let, for instance, s_1 and s_2 be states such that $x^{s_1} = \texttt{true}$ and $x^{s_2} = \texttt{false}$ for some program variable x of type Boolean. Then neither $s_1, s_2 \models_2 \ltimes(x)$ nor $s_1, s_2 \models_2 \neg\ltimes(x)$ is true: in the first case neither $s_1, s_2 \models_{2\downarrow 1} \ltimes(x)$ nor $s_2, s_1 \models_{2\downarrow 1} \ltimes(x)$ hold, in the second case $s_1, s_1 \models_{2\downarrow 1} \neg\ltimes(x)$ does not hold.

As usual, universal validity and satisfiability have the following properties.

Lemma 45.

1. *A formula ϕ is universal valid if and only if $\emptyset \models_2 \phi$ holds.*

2. *If a formula ϕ is satisfyable, then $\neg\phi$ is not universally valid, that is $\emptyset \models_2 \neg\phi$ does not hold.*

Proof.

1. Any two-state structure $(\mathcal{D}, s_1, s_2)$ is a two-state model for the empty set of formulas. Therefore, by Definition 41 (Logical Consequence in Two-State Semantics), $\emptyset \models_2 \phi$ if and only if $\mathcal{D}, s_1, s_2 \models_2 \phi$ for all structure $(\mathcal{D}, s_1, s_2)$. By Definition 35 (Universal Validity in Two-State Semantics), the latter holds if and only if ϕ is universally valid.

2. By Definition 40 (Satisfyability in Two-State Semantics), ϕ is satisfyable if and only if there exists a two-state structure $(\mathcal{D}, s_1, s_2)$ and a variable assignment β such that $\mathcal{D}, s_1, s_2, \beta \models_2 \phi$. By Lemma 43, $\mathcal{D}, s_1, s_2, \beta \models_2 \phi$ implies $\mathcal{D}, s_1, s_2, \beta \not\models_2 \neg\phi$. Therefore, $\mathcal{D}, s_1, s_2, \beta \models_2 \neg\phi$ does not hold for all two-state structures $(\mathcal{D}, s_1, s_2)$ and all variable assignments β. Hence, by Definition 41 (Logical Consequence in Two-State Semantics), $\emptyset \models_2 \neg\phi$ does not hold.

□

Unfortunately, logical equivalence is *no* congruence if interpreted in two-state semantics. It is not even transitive. However, logical equivalence interpreted in restricted two-state evaluation is a congruence. This will be useful in subsequent proofs:

Definition 46 (Logical Equivalence in Restricted Two-State Evaluation)**.** *Two formulas ϕ and ψ are logical equivalent in restricted two-state evaluation, denoted by $\phi \equiv_{2\downarrow 1} \psi$ if and only if $\mathcal{D}, s_1, s_2 \models_{2\downarrow 1} \phi \leftrightarrow \psi$ for all two-state structures $(\mathcal{D}, s_1, s_2)$.*

Lemma 47 (Auxiliary Lemma)**.** *Let ϕ and ψ be formulas.*

$\phi \equiv_{2\downarrow 1} \psi$ if and only if either $\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi$ and $\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \psi$ or $\mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \phi$ and $\mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \psi$ for all two-state structures $(\mathcal{D}, s_1, s_2)$ and variable assignments β.

Proof. $\phi \equiv_{2\downarrow 1} \psi$

$(\text{Definition 46}) \Leftrightarrow$ for all structures $(\mathcal{D}, s_1, s_2)$ and variable assignments β:
$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \leftrightarrow \psi$

$(\text{Definition of } \leftrightarrow) \Leftrightarrow$ for all structures $(\mathcal{D}, s_1, s_2)$ and variable assignments β:
$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)$

$(\text{Definition 31}) \Leftrightarrow$ for all structures $(\mathcal{D}, s_1, s_2)$ and variable assignments β:
$(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi$ and $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi)$
or $(\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \phi$ and $\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \psi)$

□

Lemma 48 ($\equiv_{2\downarrow 1}$ is a Congruence)**.** *Let ϕ_1, ϕ_2, ψ_1 and ψ_2 be formulas.*

$\equiv_{2\downarrow 1}$ is a congruence, that is, $\equiv_{2\downarrow 1}$ is an equivalence relation such that $\phi_1 \equiv_{2\downarrow 1} \phi_2$ and $\psi_1 \equiv_{2\downarrow 1} \psi_2$ imply

1. $\neg\phi_1 \equiv_{2\downarrow 1} \neg\phi_2$
2. $(\phi_1 \ op \ \psi_1) \equiv_{2\downarrow 1} (\phi_2 \ op \ \psi_2)$ *for* $op \in \{\wedge, \vee\}$
3. $Qx.\phi_1 \equiv_{2\downarrow 1} Qx.\phi_2$ *for* $x \in \mathcal{V}, Q \in \{\forall, \exists\}$
4. $\{u\}\phi_1 \equiv_{2\downarrow 1} \{u\}\phi_2$
5. $[[\alpha]]\phi_1 \equiv_{2\downarrow 1} [[\alpha]]\phi_2$ *for* $[[\alpha]] = [\alpha]$ *or* $[[\alpha]] = \langle\alpha\rangle$ *and* α *a program*
6. $\ltimes(\phi_1) \equiv_{2\downarrow 1} \ltimes(\phi_2)$.

Proof. Firstly, $\equiv_{2\downarrow 1}$ is an equivalence relation:

- Reflexivity: $\phi_1 \equiv_{2\downarrow 1} \phi_1 \quad \Leftrightarrow \quad \vDash_{2\downarrow 1} \phi_1 \leftrightarrow \phi_1 \quad \Leftrightarrow \quad \vDash_{2\downarrow 1} true$
- Symmetry: $\phi_1 \equiv_{2\downarrow 1} \phi_2 \quad \Leftrightarrow \quad \vDash_{2\downarrow 1} \phi_1 \leftrightarrow \phi_2$
  $\Leftrightarrow \quad \vDash_{2\downarrow 1} \phi_2 \leftrightarrow \phi_1 \quad \Leftrightarrow \quad \phi_2 \equiv_{2\downarrow 1} \phi_1$
- Transitivity: By Lemma 47, $\phi_1 \equiv_{2\downarrow 1} \phi_2$ and $\phi_2 \equiv_{2\downarrow 1} \psi_1$ implies

$$\begin{aligned}
&\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta: \\
&\quad \big(\quad (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_2) \\
&\qquad \text{or } (\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \phi_2)\big) \\
&\text{and } \big(\quad (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_2 \text{ and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi_1) \\
&\qquad \text{or } (\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \phi_2 \text{ and } \mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \psi_1)\big)
\end{aligned} \tag{7.9}$$

which in turn is equivalent to

$$\begin{aligned}
&\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:}\\
&\quad(\quad \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi_1\\
&\qquad\text{and } \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi_2\\
&\qquad\text{and } \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \psi_1)\\
&\text{or}\ (\quad \mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \phi_1\\
&\qquad\text{and } \mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \phi_2\\
&\qquad\text{and } \mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \psi_1)
\end{aligned} \tag{7.10}$$

by distribution of "and" over "or" and subsequent simplification. Equation (7.10) implies

$$\begin{aligned}
&\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:}\\
&\quad(\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \psi_1)\\
&\text{or } (\mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \psi_1)
\end{aligned} \tag{7.11}$$

by dropping $\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi_2$ and $\mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \phi_2$, respectively. By Definition 31 (Restricted Two-State Evaluation), Equation 7.11 is equivalent to $\models_{2\downarrow 1} (\phi_1 \wedge \psi_1) \vee (\neg\phi_1 \wedge \neg\psi_1)$ which in turn is equivalent to $\models_{2\downarrow 1} \phi_1 \leftrightarrow \psi_1$ by the definition of $\leftrightarrow$. $\models_{2\downarrow 1} \phi_1 \leftrightarrow \psi_1$ finally is equivalent to $\phi_1 \equiv_{2\downarrow 1} \psi_1$ by Definition 46 (Logical Equivalence in Restricted Two-State Evaluation).

Secondly, $\phi_1 \equiv_{2\downarrow 1} \phi_2$ and $\psi_1 \equiv_{2\downarrow 1} \psi_2$ imply 1. to 6.:

1. $$\begin{aligned}
&\phi_1 \equiv_{2\downarrow 1} \phi_2\\
\Leftrightarrow\ &\text{(Definition 46)} && \models_{2\downarrow 1} \phi_1 \leftrightarrow \phi_2\\
\Leftrightarrow\ &\text{(Definition of } \leftrightarrow\text{)} && \models_{2\downarrow 1} (\neg\phi_1 \wedge \neg\phi_2) \vee (\phi_1 \wedge \phi_2)\\
\Leftrightarrow\ &\left(\begin{array}{l}\text{Commutativity of } \vee,\\ \text{Definition of } \leftrightarrow\end{array}\right) && \models_{2\downarrow 1} \neg\phi_1 \leftrightarrow \neg\phi_2\\
\Leftrightarrow\ &\text{(Definition 46)} && \neg\phi_1 \equiv_{2\downarrow 1} \neg\phi_2
\end{aligned}$$

2. Similar to Lemma 47, $\phi_1 \equiv_{2\downarrow 1} \phi_2$ and $\psi_1 \equiv_{2\downarrow 1} \psi_2$ implies

$$\begin{aligned}
&\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:}\\
&\quad(\quad \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi_1 \wedge \phi_2\\
&\qquad\text{or } \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \neg\phi_1 \wedge \neg\phi_2)\\
&\text{and}\ (\quad \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \psi_1 \wedge \psi_2\\
&\qquad\text{or } \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \neg\psi_1 \wedge \neg\psi_2)\,.
\end{aligned} \tag{7.12}$$

Further, by Lemma 47, $(\phi_1\ op\ \psi_1) \equiv_{2|1} (\phi_2\ op\ \psi_2)$ is equivalent to

$$\begin{aligned}
&\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:}\\
&\quad (\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi_1\ op\ \psi_1\\
&\qquad \text{and } \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi_2\ op\ \psi_2)\\
&\text{or } (\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg(\phi_1\ op\ \psi_1)\\
&\qquad \text{and } \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg(\phi_2\ op\ \psi_2))\,.
\end{aligned} \tag{7.13}$$

It remains to be shown that (7.12) implies (7.13) for $op \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$. To this end, let $(\mathcal{D}, s_1, s_2)$ be a two-state structure and let β be a variable assignment.

- Case $op = \wedge$:
  - Case $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi_1 \wedge \phi_2$ and $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \psi_1 \wedge \psi_2$:

    In this case the first part of Equation 7.13,

    $$\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi_1 \wedge \psi_1 \text{and } \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi_2 \wedge \psi_2\ ,$$

    is fulfilled (Definition 31 and commutativity of "and").
  - Case $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi_1 \wedge \phi_2$ and $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg\psi_1 \wedge \neg\psi_2$:

    In this case the second part of Equation 7.13,

    $$\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg(\phi_1 \wedge \psi_1) \text{and } \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg(\phi_2 \wedge \psi_2))\,,$$

    is fulfilled, because $\mathcal{D}, s_1, s_2, \beta \nvDash_{2|1} \psi_1$ and $\mathcal{D}, s_1, s_2, \beta \nvDash_{2|1} \psi_2$ hold by Definition 31 (Restricted Two-State Evaluation)
  - Case $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg\phi_1 \wedge \neg\phi_2$ and ($\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \psi_1 \wedge \psi_2$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg\psi_1 \wedge \neg\psi_2$):

    In this case the second part of Equation 7.13,

    $$\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg(\phi_1 \wedge \psi_1) \text{and } \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg(\phi_2 \wedge \psi_2))\,,$$

    is fulfilled, because $\mathcal{D}, s_1, s_2, \beta \nvDash_{2|1} \phi_1$ and $\mathcal{D}, s_1, s_2, \beta \nvDash_{2|1} \phi_2$ hold by Definition 31 (Restricted Two-State Evaluation).
- Case $op = \vee$:
  - Case $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi_1 \wedge \phi_2$ and ($\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \psi_1 \wedge \psi_2$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg\psi_1 \wedge \neg\psi_2$):

    In this case the first part of Equation 7.13,

    $$\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi_1 \vee \psi_1 \text{and } \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi_2 \vee \psi_2\ ,$$

is fulfilled, because $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1$ and $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_2$ hold by Definition 31 (Restricted Two-State Evaluation).

– Case $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\phi_1 \wedge \neg\phi_2$ and $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi_1 \wedge \psi_2$:

In this case the first part of Equation 7.13,

$$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1 \vee \psi_1 \text{and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_2 \vee \psi_2 \ ,$$

is fulfilled, because $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi_1$ and $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi_2$ hold by Definition 31 (Restricted Two-State Evaluation).

– Case $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\phi_1 \wedge \neg\phi_2$ and $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\psi_1 \wedge \neg\psi_2$:

In this case the second part of Equation 7.13,

$$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg(\phi_1 \vee \psi_1) \text{and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg(\phi_2 \vee \psi_2)) \ ,$$

is fulfilled (Definition 31 and commutativity of "and").

3. By Lemma 47, $\phi_1 \equiv_{2\downarrow 1} \phi_2$ is equivalent to

$$\begin{aligned} &\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:} \\ &\quad (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_2) \\ &\text{or } (\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \phi_2) \end{aligned} \tag{7.14}$$

whereas $Qx.\phi_1 \equiv_{2\downarrow 1} Qx.\phi_2$ for $x \in Var, Q \in \{\forall, \exists\}$ is equivalent to

$$\begin{aligned} &\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:} \\ &\quad (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} Qx.\phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} Qx.\phi_2) \\ &\text{or } (\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} Qx.\phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} Qx.\phi_2) \ . \end{aligned} \tag{7.15}$$

- Case $Q = \forall$: Equation 7.15 holds if and only if

$$\begin{aligned} &\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and all variable assignments } \beta\text{:} \\ &\quad (\text{for all } d \in D\text{:} \qquad \mathcal{D}, s_1, s_2, \beta^{x/d} \vDash_{2\downarrow 1} \phi_1 \\ &\qquad\qquad \text{and } \mathcal{D}, s_1, s_2, \beta^{x/d} \vDash_{2\downarrow 1} \phi_2) \\ &\text{or } (\text{there exist } d_1, d_2 \in D\text{:} \qquad \mathcal{D}, s_1, s_2, \beta^{x/d_1} \nvDash_{2\downarrow 1} \phi_1 \\ &\qquad\qquad \text{and } \mathcal{D}, s_1, s_2, \beta^{x/d_2} \nvDash_{2\downarrow 1} \phi_2) \ . \end{aligned} \tag{7.16}$$

Assume that $\mathcal{D}, s_1, s_2, \beta^{x/d} \vDash_{2\downarrow 1} \phi_1$ and $\mathcal{D}, s_1, s_2, \beta^{x/d} \vDash_{2\downarrow 1} \phi_2$ do not hold for all $d \in D$. Otherwise we are done. Then there exists a $d \in D$ such that $\mathcal{D}, s_1, s_2, \beta^{x/d} \nvDash_{2\downarrow 1} \phi_1$ or $\mathcal{D}, s_1, s_2, \beta^{x/d} \nvDash_{2\downarrow 1} \phi_2$. By (7.14) we have that there exists a $d \in D$ such that $\mathcal{D}, s_1, s_2, \beta^{x/d} \nvDash_{2\downarrow 1} \phi_1$ and $\mathcal{D}, s_1, s_2, \beta^{x/d} \nvDash_{2\downarrow 1} \phi_2$ and hence (7.16) holds.

- Case $Q = \exists$: Equation 7.15 holds if and only if

$$\begin{aligned}&\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and all variable assignments } \beta\text{:}\\ &\quad(\text{there exist } d_1, d_2 \in D\text{:} \qquad \mathcal{D}, s_1, s_2, \beta^{x/d_1} \models_{2\downarrow 1} \phi_1\\ &\qquad\qquad\qquad\qquad \text{and } \mathcal{D}, s_1, s_2, \beta^{x/d_2} \models_{2\downarrow 1} \phi_2) \qquad (7.17)\\ &\quad\text{or } (\text{for all } d \in D\text{:} \qquad \mathcal{D}, s_1, s_2, \beta^{x/d} \not\models_{2\downarrow 1} \phi_1\\ &\qquad\qquad\qquad\qquad \text{and } \mathcal{D}, s_1, s_2, \beta^{x/d} \not\models_{2\downarrow 1} \phi_2)\end{aligned}$$

  Assume that $\mathcal{D}, s_1, s_2, \beta^{x/d} \not\models_{2\downarrow 1} \phi_1$ and $\mathcal{D}, s_1, s_2, \beta^{x/d} \not\models_{2\downarrow 1} \phi_2$ do not hold for all $d \in D$. Otherwise we are done. Then there exists a $d \in D$ such that $\mathcal{D}, s_1, s_2, \beta^{x/d} \models_{2\downarrow 1} \phi_1$ or $\mathcal{D}, s_1, s_2, \beta^{x/d} \models_{2\downarrow 1} \phi_2$. By (7.14) we have that there exists a $d \in D$ such that $\mathcal{D}, s_1, s_2, \beta^{x/d} \models_{2\downarrow 1} \phi_1$ and $\mathcal{D}, s_1, s_2, \beta^{x/d} \models_{2\downarrow 1} \phi_2$ and hence (7.17) holds.

4. By Lemma 47, $\phi_1 \equiv_{2\downarrow 1} \phi_2$ is equivalent to

$$\begin{aligned}&\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:}\\ &\quad(\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi_2) \qquad (7.18)\\ &\text{or } (\mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \phi_2)\end{aligned}$$

   whereas $\{u\}\phi_1 \equiv_{2\downarrow 1} \{u\}\phi_2$ is equivalent to

$$\begin{aligned}&\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:}\\ &\quad(\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \{u\}\phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \{u\}\phi_2) \qquad (7.19)\\ &\text{or } (\mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \{u\}\phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \{u\}\phi_2)\ .\end{aligned}$$

   By Definition 31, equation (7.19) equals

$$\begin{aligned}&\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:}\\ &\quad(\mathcal{D}, s_1^u, s_2^u, \beta \models_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1^u, s_2^u, \beta \models_{2\downarrow 1} \phi_2) \qquad (7.20)\\ &\text{or } (\mathcal{D}, s_1^u, s_2^u, \beta \not\models_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1^u, s_2^u, \beta \not\models_{2\downarrow 1} \phi_2)\end{aligned}$$

   where s_1^u and s_2^u result from s_1 and s_2 by application of $\{u\}$, respectively. Equation (7.18) implies (7.20).

5. By Lemma 47, $\phi_1 \equiv_{2\downarrow 1} \phi_2$ is equivalent to

$$\begin{aligned}&\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:}\\ &\quad(\mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \phi_2) \qquad (7.21)\\ &\text{or } (\mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \not\models_{2\downarrow 1} \phi_2)\end{aligned}$$

whereas $[[\alpha]]\phi_1 \equiv_{2\downarrow 1} [[\alpha]]\phi_2$ for $[[\alpha]] = [\alpha]$ or $[[\alpha]] = \langle\alpha\rangle$ is equivalent to

for all structures $(\mathcal{D}, s_1, s_2)$ and variable assignments β:

$$\begin{aligned}&(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} [[\alpha]]\phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} [[\alpha]]\phi_2) \qquad (7.22)\\ \text{or } &(\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} [[\alpha]]\phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} [[\alpha]]\phi_2)\ .\end{aligned}$$

- Case $[[\alpha]] = [\alpha]$: Equation 7.22 holds if and only if

  for all structures $(\mathcal{D}, s_1, s_2)$ and all variable assignments β:

  $$\begin{aligned}&(\text{for all states } s_1^\alpha,\ s_2^\alpha \text{ such that } \alpha \text{ started in } s_1 \text{ terminates in } s_1^\alpha \text{ and } \alpha \text{ started in } s_2 \text{ terminates in } s_2^\alpha \text{ we have}\\ &\qquad \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi_2) \qquad (7.23)\\ \text{or } &(\text{there are states } s_1^\alpha,\ s_2^\alpha \text{ such that } \alpha \text{ started in } s_1 \text{ terminates in } s_1^\alpha \text{ and } \alpha \text{ started in } s_2 \text{ terminates in } s_2^\alpha \text{ and such that}\\ &\qquad \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \nvDash_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \nvDash_{2\downarrow 1} \phi_2)\ .\end{aligned}$$

  Assume that it is not the case that for all states s_1^α, s_2^α such that α started in s_1 terminates in s_1^α and α started in s_2 terminates in s_2^α we have $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi_1$ and $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi_2$. (Otherwise we are done.) Then there are states s_1^α, s_2^α such that α started in s_1 terminates in s_1^α and α started in s_2 terminates in s_2^α and such that $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \nvDash_{2\downarrow 1} \phi_1$ or $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \nvDash_{2\downarrow 1} \phi_2$. By (7.21) we get that there are states s_1^α, s_2^α such that α started in s_1 terminates in s_1^α and α started in s_2 terminates in s_2^α and such that $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \nvDash_{2\downarrow 1} \phi_1$ and $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \nvDash_{2\downarrow 1} \phi_2$. Thus 7.23 holds.

- Case $[[\alpha]] = \langle\alpha\rangle$: Equation 7.22 holds if and only if

  for all structures $(\mathcal{D}, s_1, s_2)$ and all variable assignments β:

  $$\begin{aligned}&(\text{there are states } s_1^\alpha,\ s_2^\alpha \text{ such that } \alpha \text{ started in } s_1 \text{ terminates in } s_1^\alpha \text{ and } \alpha \text{ started in } s_2 \text{ terminates in } s_2^\alpha \text{ and such that}\\ &\qquad \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi_2) \qquad (7.24)\\ \text{or } &(\text{for all states } s_1^\alpha,\ s_2^\alpha \text{ such that } \alpha \text{ started in } s_1 \text{ terminates in } s_1^\alpha \text{ and } \alpha \text{ started in } s_2 \text{ terminates in } s_2^\alpha \text{ we have}\\ &\qquad \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \nvDash_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \nvDash_{2\downarrow 1} \phi_2)\ .\end{aligned}$$

  Assume that it is not the case that for all states s_1^α, s_2^α such that α started in s_1 terminates in s_1^α and α started in s_2 terminates in s_2^α we have $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \nvDash_{2\downarrow 1} \phi_1$ and $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \nvDash_{2\downarrow 1} \phi_2$. (Otherwise

we are done.) Then there are states s_1^α, s_2^α such that α started in s_1 terminates in s_1^α and α started in s_2 terminates in s_2^α and such that $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \models_{2|1} \phi_1$ or $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \models_{2|1} \phi_2$. By (7.21) we get that there are states s_1^α, s_2^α such that α started in s_1 terminates in s_1^α and α started in s_2 terminates in s_2^α and such that $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \models_{2|1} \phi_1$ and $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \models_{2|1} \phi_2$. Thus 7.24 holds.

6. By Lemma 47, $\phi_1 \equiv_{2|1} \phi_2$ is equivalent to

$$\begin{aligned} &\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:} \\ &\quad (\mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_2) \\ &\text{or } (\mathcal{D}, s_1, s_2, \beta \not\models_{2|1} \phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \not\models_{2|1} \phi_2) \end{aligned} \tag{7.25}$$

whereas $\ltimes(\phi_1) \equiv_{2|1} \ltimes(\phi_2)$ is equivalent to

$$\begin{aligned} &\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:} \\ &\quad (\mathcal{D}, s_1, s_2, \beta \models_{2|1} \ltimes(\phi_1) \text{ and } \mathcal{D}, s_1, s_2, \beta \models_{2|1} \ltimes(\phi_2)) \\ &\text{or } (\mathcal{D}, s_1, s_2, \beta \not\models_{2|1} \ltimes(\phi_1) \text{ and } \mathcal{D}, s_1, s_2, \beta \not\models_{2|1} \ltimes(\phi_2))\ . \end{aligned} \tag{7.26}$$

By Definition 31, equation (7.26) equals

$$\begin{aligned} &\text{for all structures } (\mathcal{D}, s_1, s_2) \text{ and variable assignments } \beta\text{:} \\ &\quad (\quad \mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_1 \text{ iff } \mathcal{D}, s_2, s_1, \beta \models_{2|1} \phi_1 \\ &\qquad \text{and } \mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_2 \text{ iff } \mathcal{D}, s_2, s_1, \beta \models_{2|1} \phi_2) \\ &\text{or } (\quad \mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_1 \text{ iff } \mathcal{D}, s_2, s_1, \beta \not\models_{2|1} \phi_1 \\ &\qquad \text{and } \mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_2 \text{ iff } \mathcal{D}, s_2, s_1, \beta \not\models_{2|1} \phi_2)\ . \end{aligned} \tag{7.27}$$

Assume that

$$\begin{aligned} \text{not } (\quad &\mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_1 \text{ iff } \mathcal{D}, s_2, s_1, \beta \models_{2|1} \phi_1 \\ &\text{and } \mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_2 \text{ iff } \mathcal{D}, s_2, s_1, \beta \models_{2|1} \phi_2) \end{aligned}$$

holds. Otherwise we are done. Then we have $\mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_1$ iff $\mathcal{D}, s_2, s_1, \beta \not\models_{2|1} \phi_1$ or $\mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_2$ iff $\mathcal{D}, s_2, s_1, \beta \not\models_{2|1} \phi_2$. By equation (7.25) we get $\mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_1$ iff $\mathcal{D}, s_2, s_1, \beta \not\models_{2|1} \phi_1$ and $\mathcal{D}, s_1, s_2, \beta \models_{2|1} \phi_2$ iff $\mathcal{D}, s_2, s_1, \beta \not\models_{2|1} \phi_2$. Thus, (7.27) holds.

□

A positive consequence of Lemma 39 and Definition 35 (Universal Validity in Two-State Semantics) is that functional and information flow properties can be verified together: Formula (7.4) is a combination of the scheme for information flow properties with the usual scheme $\phi \rightarrow [\alpha]\psi$ for functional properties,

where ϕ and ψ neither contain the $\ltimes$ operator nor modalities. (7.4) is valid in two-state semantics, written $s_1, s_2 \vDash_2 (\phi \wedge \ltimes(R_1)) \rightarrow [\alpha](\psi \wedge \ltimes(R_2))$, if and only if

$$\begin{array}{rl} & s_1 \vDash \phi \\ \text{and} & s_2 \vDash \phi \\ \text{and} & R_1^{s_1} = R_1^{s_2} \\ \text{imply} & \text{for all states } s_1^\alpha,\ s_2^\alpha \text{ such that } \alpha \text{ terminates in } s_1^\alpha,\ s_2^\alpha \text{ if started in} \\ & s_1,\ s_2, \text{ respectively, } (s_1^\alpha \vDash \psi \text{ or } s_2^\alpha \vDash \psi) \text{ and } R_2^{s_1^\alpha} = R_2^{s_2^\alpha} \text{ hold} \end{array}$$

and

$$\begin{array}{rl} & s_1 \vDash \phi \\ \text{implies} & \text{for all states } s_1^\alpha \text{ such that } \alpha \text{ terminates in } s_1^\alpha \text{ if started in } s_1 \text{ the} \\ & \text{equation } s_1^\alpha \vDash \psi \text{ holds.} \end{array}$$

The latter agrees with the one-state semantics of $\phi \rightarrow [\alpha]\psi$. Hence, if (7.4) is universally valid in two-state semantics, then $\phi \rightarrow [\alpha]\psi$ is universally valid in one-state semantics and (7.1) is universally valid in two-state semantics.

In general, the following connections between the universal validity in one-state semantics and the universal validity in two-state semantics exist.

Lemma 49.

1. *If a formula ϕ is universally valid in two-state semantics, then it is universally valid in one-state semantics, too.*
2. *If a formula ϕ neither contains the $\ltimes$ operator nor modalities, then $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \phi$ holds.*
3. *Let u, u' be updates and let α, α' be programs such that for all states s the program α started in s^u terminates if and only if α' started in $s^{u'}$ terminates (where s^u is the state resulting from s by application of update u).*

   If the formulas ϕ and ψ neither contain the $\ltimes$ operator nor modalities, then

   (a) $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} [\alpha]\phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash [\alpha]\phi$ or $\mathcal{D}, s_2, \beta \vDash [\alpha]false$ hold;

   (b) $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \langle\alpha\rangle\phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \langle\alpha\rangle\phi$ and $\mathcal{D}, s_2, \beta \vDash \langle\alpha\rangle true$ hold;

   (c) $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \{u\}[\alpha]\phi \rightarrow \{u'\}[\alpha']\phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \{u\}[\alpha]\phi \rightarrow \{u'\}[\alpha']\phi$ or $\mathcal{D}, s_2, \beta \vDash \{u'\}[\alpha']false$;

   (d) $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \{u\}\langle\alpha\rangle\phi \rightarrow \{u'\}\langle\alpha'\rangle\phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \{u\}\langle\alpha\rangle\phi \rightarrow \{u'\}\langle\alpha'\rangle\phi$ or $\mathcal{D}, s_2, \beta \vDash \{u\}[\alpha]false$;

(e) $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} (\psi \wedge \{u\}[\alpha]\phi) \rightarrow \{u'\}[\alpha']\phi$ *holds if and only if* $\mathcal{D}, s_1, \beta \vDash (\psi \wedge \{u\}[\alpha]\phi) \rightarrow \{u'\}[\alpha']\phi$ *or* $\mathcal{D}, s_2, \beta \vDash \{u'\}[\alpha']false$*; and*

(f) $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} (\psi \wedge \{u\}\langle\alpha\rangle\phi) \rightarrow \{u'\}\langle\alpha'\rangle\phi$ *holds if and only if* $\mathcal{D}, s_1, \beta \vDash (\psi \wedge \{u\}\langle\alpha\rangle\phi) \rightarrow \{u'\}\langle\alpha'\rangle\phi$ *or* $\mathcal{D}, s_2, \beta \vDash \{u\}[\alpha]false$.

4. *If a formula ϕ does not contain the $\ltimes$ operator and if each modality in ϕ is prefixed by an update of the form $\{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}$, where* heap *and $\bar{x}$ completely describe a state and where h and $\bar{x}$ are (arbitrary) variables, then $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \phi$ holds.*

5. *If a formula ϕ, which neither contains the $\ltimes$ operator nor modalities, is universally valid in one-state semantics, then it is universally valid in two-state semantics, too.*

6. *If a formula ϕ, which does not contain the $\ltimes$ operator and in which each modality is prefixed by an update of the form $\{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}$ (where* heap *and $\bar{x}$ completely describe a state and where h and $\bar{x}$ are variables), is universally valid in one-state semantics, then it is universally valid in two-state semantics, too.*

Proof.

1. The first part of the lemma is a direct consequence of Lemma 39 and Definition 35. Let $\mathcal{D}, s_1, s_2, \beta \vDash_2 \phi$ hold for all two-state structures $(\mathcal{D}, s_1, s_2)$ and all variable assignments β. Then in particular $\mathcal{D}, s_1, s_1, \beta \vDash_2 \phi$ holds for all two-state structures $(\mathcal{D}, s_1, s_1)$ and all variable assignments β. By Lemma 39 this is equivalent to: $\mathcal{D}, s_1, \beta \vDash \phi$ for all structures $(\mathcal{D}, s_1)$ and variable assignments β.

2. The proof parallels the one of Lemma 39 in great parts. The lemma follows by Definition 31 by induction on the structure of Java DL formulas.

   Base case. Let ϕ be a formula with a predicate symbol as top level symbol. By Definition 31, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \phi$ holds. Because ϕ does not contain $\ltimes$, this is the only base case.

   Step cases.

   Case $\phi = \phi_1 \wedge \phi_2$. $\mathcal{D}, s_1, \beta \vDash \phi_1 \wedge \phi_2$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \phi_1$ and $\mathcal{D}, s_1, \beta \vDash \phi_2$ hold. By the induction hypothesis this is equivalent to $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1$ and $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_2$, which again is equivalent to $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1 \wedge \phi_2$.

   Case $\phi = \phi_1 \vee \phi_2$. $\mathcal{D}, s_1, \beta \vDash \phi_1 \vee \phi_2$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \phi_1$ or $\mathcal{D}, s_1, \beta \vDash \phi_2$ hold. By the induction hypothesis this is equivalent to $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_2$, which again is equivalent to $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi_1 \vee \phi_2$.

Case $\phi = \neg\phi_1$. $\mathcal{D}, s_1, \beta \vDash \neg\phi_1$ holds if and only if $\mathcal{D}, s_1, \beta \nvDash \phi_1$ holds. By the induction hypothesis this is equivalent to $\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \phi_1$, which again is equivalent to $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\phi_1$.

Case $\phi = \forall x.\phi_1$. $\mathcal{D}, s_1, \beta \vDash \forall x.\phi_1$ holds if and only if $\mathcal{D}, s_1, \beta^{x/d} \vDash \phi_1$ for all $d \in D$. By the induction hypothesis the latter equals $\mathcal{D}, s_1, s_2, \beta^{x/d} \vDash_{2\downarrow 1} \phi_1$ for all $d \in D$, which again is equivalent to $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \forall x.\phi_1$.

Case $\phi = \exists x.\phi_1$. $\mathcal{D}, s_1, \beta \vDash \forall x.\phi_1$ holds if and only if there exists a $d \in D$ such that $\mathcal{D}, s_1, \beta^{x/d} \vDash \phi_1$. By the induction hypothesis $\mathcal{D}, s_1, \beta^{x/d} \vDash \phi_1$ if and only if $\mathcal{D}, s_1, s_2, \beta^{x/d} \vDash_{2\downarrow 1} \phi_1$. Therefore, there exists a $d \in D$ such that $\mathcal{D}, s_1, s_2, \beta^{x/d} \vDash_{2\downarrow 1} \phi_1$, which again is equivalent to $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \exists x.\phi_1$.

Case $\phi = \{u\}\phi_1$. $\mathcal{D}, s_1, \beta \vDash \{u\}\phi_1$ holds if and only if $\mathcal{D}, s_1^u, \beta \vDash \phi_1$ where s_i^u results from s_i by application of $\{u\}$. By the induction hypothesis this is equivalent to $\mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2\downarrow 1} \phi_1$, which again is equivalent to $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\phi_1$.

Because ϕ does not contain modalities, the cases $\phi = [\alpha]\phi_1$ and $\phi = \langle\alpha\rangle\phi_1$ do not occur.

3. (a) By Definition 31, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} [\alpha]\phi$ if and only if $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi$ for all s_1^α, s_2^α such that α started in s_i terminates in s_i^α. By Part 2, $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi$ if and only if $\mathcal{D}, s_1^\alpha, \beta \vDash \phi$. Thus, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} [\alpha]\phi$ if and only if $\mathcal{D}, s_1^\alpha, \beta \vDash \phi$ for all s_1^α, s_2^α such that α started in s_i terminates in s_i^α. The latter is equivalent to $\mathcal{D}, s_1, \beta \vDash [\alpha]\phi$ or α started in s_2 does not terminate. This again is equivalent to $\mathcal{D}, s_1, \beta \vDash [\alpha]\phi$ or $\mathcal{D}, s_2, \beta \vDash [\alpha]\mathit{false}$.

   (b) By Definition 31, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \langle\alpha\rangle\phi$ holds if and only if there exist s_1^α, s_2^α such that α started in s_i terminates in s_i^α and $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi$ holds. By Part 2, $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} \phi$ if and only if $\mathcal{D}, s_1^\alpha, \beta \vDash \phi$. Therefore, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \langle\alpha\rangle\phi$ if and only if there exist s_1^α, s_2^α such that α started in s_i terminates in s_i^α and $\mathcal{D}, s_1^\alpha, \beta \vDash \phi$. The latter is equivalent to $\mathcal{D}, s_1, \beta \vDash \langle\alpha\rangle\phi$ and there exist s_2^α such that α started in s_2 terminates in s_2^α. This again is equivalent to $\mathcal{D}, s_1, \beta \vDash \langle\alpha\rangle\phi$ and $\mathcal{D}, s_2, \beta \vDash \langle\alpha\rangle\mathit{true}$.

   (c) By Definition 31, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}[\alpha]\phi \rightarrow \{u'\}[\alpha']\phi$ holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\langle\alpha\rangle\neg\phi$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u'\}[\alpha']\phi$. Again by Definition 31 and by Parts 3a and 3b, this is equivalent to

$$\begin{aligned}&(\mathcal{D}, s_1, \beta \vDash \{u\}\langle\alpha\rangle\neg\phi \text{ and } \mathcal{D}, s_2, \beta \vDash \{u\}\langle\alpha\rangle\mathit{true})\\ \text{or } &\mathcal{D}, s_1, \beta \vDash \{u'\}[\alpha']\phi \text{ or } \mathcal{D}, s_2, \beta \vDash \{u'\}[\alpha']\mathit{false}\,.\end{aligned} \tag{7.28}$$

By assumption we have $\mathcal{D}, s_2, \beta \vDash \{u\}\langle\alpha\rangle true$ if and only if $\mathcal{D}, s_2, \beta \vDash \{u'\}\langle\alpha'\rangle true$. Therefore, Equation (7.28) is equivalent to

$$\begin{aligned} &(\quad \mathcal{D}, s_1, \beta \vDash \{u\}\langle\alpha\rangle\neg\phi \\ &\quad \text{or } \mathcal{D}, s_1, \beta \vDash \{u'\}[\alpha']\phi \\ &\quad \text{or } \mathcal{D}, s_2, \beta \vDash \{u'\}[\alpha']false\ ) \\ \text{and } &(\quad \mathcal{D}, s_2, \beta \vDash \{u'\}\langle\alpha'\rangle true \\ &\quad \text{or } \mathcal{D}, s_1, \beta \vDash \{u'\}[\alpha']\phi \\ &\quad \text{or } \mathcal{D}, s_2, \beta \vDash \{u'\}[\alpha']false\ )\ . \end{aligned} \tag{7.29}$$

Because

$$\mathcal{D}, s_2, \beta \vDash \{u'\}\langle\alpha'\rangle true \text{ or } \mathcal{D}, s_2, \beta \vDash \{u'\}[\alpha']false$$

is always fulfilled, the second conjunct is always true. Thus, (7.29) is equivalent to

$$\mathcal{D}, s_1, \beta \vDash \{u\}\langle\alpha\rangle\neg\phi \text{ or } \mathcal{D}, s_1, \beta \vDash \{u'\}[\alpha']\phi \text{ or } \mathcal{D}, s_2, \beta \vDash \{u'\}[\alpha']false\ ,$$

which again is equivalent to

$$\mathcal{D}, s_1, \beta \vDash \{u\}[\alpha]\phi \rightarrow \{u'\}[\alpha']\phi \text{ or } \mathcal{D}, s_2, \beta \vDash \{u'\}[\alpha']false\ ,$$

as desired.

(d) By Definition 31, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\langle\alpha\rangle\phi \rightarrow \{u'\}\langle\alpha'\rangle\phi$ holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}[\alpha]\neg\phi$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u'\}\langle\alpha'\rangle\phi$. Again by Definition 31 and by Parts 3a and 3b, this is equivalent to

$$\begin{aligned} &\mathcal{D}, s_1, \beta \vDash \{u\}[\alpha]\neg\phi \text{ or } \mathcal{D}, s_2, \beta \vDash \{u\}[\alpha]false \\ &\text{or } (\mathcal{D}, s_1, \beta \vDash \{u'\}\langle\alpha'\rangle\phi \text{ and } \mathcal{D}, s_2, \beta \vDash \{u'\}\langle\alpha'\rangle true)\ . \end{aligned} \tag{7.30}$$

By assumption we have $\mathcal{D}, s_2, \beta \vDash \{u\}\langle\alpha\rangle true$ if and only if $\mathcal{D}, s_2, \beta \vDash \{u'\}\langle\alpha'\rangle true$. Therefore, Equation (7.30) is equivalent to

$$\begin{aligned} &(\quad \mathcal{D}, s_1, \beta \vDash \{u\}[\alpha]\neg\phi \\ &\quad \text{or } \mathcal{D}, s_2, \beta \vDash \{u\}[\alpha]false \\ &\quad \text{or } \mathcal{D}, s_1, \beta \vDash \{u'\}\langle\alpha'\rangle\phi\ ) \\ \text{and } &(\quad \mathcal{D}, s_1, \beta \vDash \{u\}[\alpha]\neg\phi \\ &\quad \text{or } \mathcal{D}, s_2, \beta \vDash \{u\}[\alpha]false \\ &\quad \text{or } \mathcal{D}, s_2, \beta \vDash \{u\}\langle\alpha\rangle true\ )\ . \end{aligned} \tag{7.31}$$

Because

$$\mathcal{D}, s_2, \beta \vDash \{u\}\langle\alpha\rangle true \text{ or } \mathcal{D}, s_2, \beta \vDash \{u\}[\alpha]false$$

is always fulfilled, the second conjunct is always true. Thus, (7.31) is equivalent to

$$\mathcal{D}, s_1, \beta \vDash \{u\}[\alpha]\neg\phi \text{ or } \mathcal{D}, s_2, \beta \vDash \{u\}[\alpha]\mathit{false} \text{ or } \mathcal{D}, s_1, \beta \vDash \{u'\}\langle\alpha'\rangle\phi\,,$$

which again is equivalent to

$$\mathcal{D}, s_1, \beta \vDash \{u\}\langle\alpha\rangle\phi \to \{u'\}\langle\alpha'\rangle\phi \text{ or } \mathcal{D}, s_2, \beta \vDash \{u\}[\alpha]\mathit{false}\,,$$

as desired.

(e) By Definition 31, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} (\psi \wedge \{u\}[\alpha]\phi) \to \{u'\}[\alpha']\phi$ if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\psi$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}[\alpha]\phi \to \{u'\}[\alpha']\phi$. Similarly, $\mathcal{D}, s_1, \beta \vDash (\psi \wedge \{u\}[\alpha]\phi) \to \{u'\}[\alpha']\phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \neg\psi$ or $\mathcal{D}, s_1, \beta \vDash \{u\}[\alpha]\phi \to \{u'\}[\alpha']\phi$. By part 2 we have $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\psi$ if and only if $\mathcal{D}, s_1, \beta \vDash \neg\psi$. Additionally, part 3c yields $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}[\alpha]\phi \to \{u'\}[\alpha']\phi$ if and only if $\mathcal{D}, s_1, \beta \vDash \{u\}[\alpha]\phi \to \{u'\}[\alpha']\phi$ or $\mathcal{D}, s_2, \beta \vDash \{u'\}[\alpha']\mathit{false}$. Thus, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} (\psi \wedge \{u\}[\alpha]\phi) \to \{u'\}[\alpha']\phi$ if and only if $\mathcal{D}, s_1, \beta \vDash (\psi \wedge \{u\}[\alpha]\phi) \to \{u'\}[\alpha']\phi$ or $\mathcal{D}, s_2, \beta \vDash \{u'\}[\alpha']\mathit{false}$.

(f) By Definition 31, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} (\psi \wedge \{u\}\langle\alpha\rangle\phi) \to \{u'\}\langle\alpha'\rangle\phi$ if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\psi$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\langle\alpha\rangle\phi \to \{u'\}\langle\alpha'\rangle\phi$. Similarly, $\mathcal{D}, s_1, \beta \vDash (\psi \wedge \{u\}\langle\alpha\rangle\phi) \to \{u'\}\langle\alpha'\rangle\phi$ holds if and only if $\mathcal{D}, s_1, \beta \vDash \neg\psi$ or $\mathcal{D}, s_1, \beta \vDash \{u\}\langle\alpha\rangle\phi \to \{u'\}\langle\alpha'\rangle\phi$. By part 2 we have $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\psi$ if and only if $\mathcal{D}, s_1, \beta \vDash \neg\psi$. Additionally, part 3d yields $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\langle\alpha\rangle\phi \to \{u'\}\langle\alpha'\rangle\phi$ if and only if $\mathcal{D}, s_1, \beta \vDash \{u\}\langle\alpha\rangle\phi \to \{u'\}\langle\alpha'\rangle\phi$ or $\mathcal{D}, s_2, \beta \vDash \{u\}[\alpha]\mathit{false}$. Thus, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} (\psi \wedge \{u\}\langle\alpha\rangle\phi) \to \{u'\}\langle\alpha'\rangle\phi$ if and only if $\mathcal{D}, s_1, \beta \vDash (\psi \wedge \{u\}\langle\alpha\rangle\phi) \to \{u'\}\langle\alpha'\rangle\phi$ or $\mathcal{D}, s_2, \beta \vDash \{u\}[\alpha]\mathit{false}$.

4. We extend the induction proof of 2. by step cases for $\phi = \{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}[\alpha]\phi_1$ and $\phi = \{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}\langle\alpha\rangle\phi_1$, where h and $\bar{x}$ are variables.

   Case $\phi = \{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}[\alpha]\phi_1$. Let s denote the state which is completely determined by $\mathtt{heap}^s = h^{\mathcal{D},\beta}$ and $\bar{\mathtt{x}}^s = \bar{x}^{\mathcal{D},\beta}$. Then $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}[\alpha]\phi_1$ holds if and only if $\mathcal{D}, s, s, \beta \vDash_{2\downarrow 1} [\alpha]\phi_1$ holds. By Lemma 39, the latter holds if and only if $\mathcal{D}, s, \beta \vDash [\alpha]\phi_1$ holds. (Note, that for this step we do not need the induction hypothesis.)

   Case $\phi = \{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}\langle\alpha\rangle\phi_1$. Analog to case $\phi = \{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}[\alpha]\phi_1$ (indeed, it is exactly the same proof).

5. Let $\mathcal{D}, s, \beta \vDash \phi$ hold for all structures $(\mathcal{D}, s)$ and all variable assignments β. Further, let $(\mathcal{D}, s_1, s_2)$ be an arbitrary two-state structure and β a variable

assignment. Then in particular $\mathcal{D}, s_1, \beta \vDash \phi$ and $\mathcal{D}, s_2, \beta \vDash \phi$ hold. By part 2 this implies

$$\mathcal{D}, s_1, s_1, \beta \vDash_{2|1} \phi \text{ and } (\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2|1} \phi)\,.$$

By Definition 33 (Two-State Evaluation) the latter holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_2 \phi$.

6. Let $\mathcal{D}, s, \beta \vDash \phi$ hold for all structures $(\mathcal{D}, s)$ and all variable assignments β. Further, let $(\mathcal{D}, s_1, s_2)$ be an arbitrary two-state structure and β a variable assignment. Then in particular $\mathcal{D}, s_1, \beta \vDash \phi$ and $\mathcal{D}, s_2, \beta \vDash \phi$ hold. By part 4 this implies

$$\mathcal{D}, s_1, s_1, \beta \vDash_{2|1} \phi \text{ and } (\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \phi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2|1} \phi)\,.$$

By Definition 33 (Two-State Evaluation) the latter holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_2 \phi$.

□

Lemma 49 is the basis for the soundness proofs of a lot of rules from Section 7.3.

The next section presents a sound two-state calculus.

7.3 Two-State Calculus

As shown in the last section, information flow problems can be formulated naturally in Java DL using the $\ltimes$ predicate. A calculus to reason about them needs to be sound with respect to the two-state semantics of those formulas (Definition 33). Unfortunately, the normal Java DL calculus is not sound with respect to this semantics. For instance, let x be a program variable of type Boolean. One possibility to derive $\ltimes(x)$ would be using the rule andRight:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{x = \texttt{false} \Longrightarrow \texttt{true}}\,\text{closeTrue}}{x = \texttt{false} \Longrightarrow \ltimes(\texttt{false})}\,\ltimes\text{constant}}{x = \texttt{false} \Longrightarrow \ltimes(x)}\,\text{applyEq}}{\Longrightarrow \ltimes(x), x = \texttt{true}}\,\text{eqTrueRight} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{x = \texttt{true} \Longrightarrow \texttt{true}}\,\text{closeTrue}}{x = \texttt{true} \Longrightarrow \ltimes(\texttt{true})}\,\ltimes\text{constant}}{x = \texttt{true} \Longrightarrow \ltimes(x)}\,\text{applyEq}}{\Longrightarrow \ltimes(x), \neg x = \texttt{true}}\,\text{notRight}}{\dfrac{\Longrightarrow \ltimes(x), x = \texttt{true} \wedge \neg x = \texttt{true}}{\Longrightarrow \ltimes(x)}}\,\text{andRight}$$

Though some of the normal Java DL rules are not sound in two-state semantics, many others indeed are and thus can be used in the two-state calculus without modification. An example of such a rule is orRight:

$$\text{orRight}\ \frac{\Gamma \Longrightarrow \phi_1, \phi_2, \Delta}{\Gamma \Longrightarrow \phi_1 \vee \phi_2, \Delta}$$

Its soundness in two-state semantics follows directly by Definition 37. In general, a non-splitting rule with premiss ϕ and conclusion ψ is sound in two-state semantics, if, roughly speaking, ϕ and ψ do not contain the $\ltimes$ predicate and if ϕ logically implies ψ in one-state semantics. In deed the condition is a bit more involved, as Theorem 52 below shows.

Lemma 50 (Auxiliary Lemma)**.** *Let $(\mathcal{D}, s_1, s_2)$ be a two-state structure and let β be a variable assignment.*

$\mathcal{D}, s_1, s_2, \beta \vDash_2 \phi$ implies $\mathcal{D}, s_1, s_2, \beta \vDash_2 \psi$ if
$\mathcal{D}, s_1', s_2', \beta \vDash_{2\downarrow 1} \phi$ implies $\mathcal{D}, s_1', s_2', \beta \vDash_{2\downarrow 1} \psi$ for all s_1', s_2'.

Proof.

$$\begin{aligned}
& \mathcal{D}, s_1, s_2, \beta \vDash_2 \phi \\
(\text{Definition 33}) \Leftrightarrow\ & (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi) \\
& \text{and } \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \phi \\
(\text{Assumption}) \Rightarrow\ & (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi) \\
& \text{and } \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \psi \\
(\text{Definition 33}) \Leftrightarrow\ & \mathcal{D}, s_1, s_2, \beta \vDash_2 \psi
\end{aligned}$$

□

Lemma 51 (Auxiliary Lemma)**.** *Let*

$$\frac{\Gamma, A^{pre}_{\phi_1,\ldots,\phi_n} \Longrightarrow B^{pre}_{\phi_1,\ldots,\phi_n}, \Delta}{\Gamma, A^{con}_{\phi_1,\ldots,\phi_n} \Longrightarrow B^{con}_{\phi_1,\ldots,\phi_n}, \Delta}$$

be a non-splitting rule schema with formula schema variables $\phi_1, \ldots, \phi_n$ such that the premiss and conclusion of the rule schema are interpreted in the same structure (in other words: such that no fresh symbols are introduced). Let further $A^{pre}_{\phi_1,\ldots,\phi_n}$, $B^{pre}_{\phi_1,\ldots,\phi_n}$, $A^{con}_{\phi_1,\ldots,\phi_n}$ and $B^{con}_{\phi_1,\ldots,\phi_n}$ be of a form such that

- *$(A^{pre}_{\phi_1,\ldots,\phi_n} \to B^{pre}_{\phi_1,\ldots,\phi_n}) \to (A^{con}_{\phi_1,\ldots,\phi_n} \to B^{con}_{\phi_1,\ldots,\phi_n})$ fulfills the requirements of Lemma 49.2, Lemma 49.4, Lemma 49.3c, Lemma 49.3d, Lemma 49.3e or Lemma 49.3f;*
- *$\phi_1, \ldots, \phi_n$ do not occur in the scope of quantifiers; and*

- *if $A^{pre}_{\phi_1,\ldots,\phi_n}$, $B^{pre}_{\phi_1,\ldots,\phi_n}$, $A^{con}_{\phi_1,\ldots,\phi_n}$ and $B^{con}_{\phi_1,\ldots,\phi_n}$ are evaluated in state s, then all occurrences of ϕ_j in $A^{pre}_{\phi_1,\ldots,\phi_n}$, $B^{pre}_{\phi_1,\ldots,\phi_n}$, $A^{con}_{\phi_1,\ldots,\phi_n}$ and $B^{con}_{\phi_1,\ldots,\phi_n}$ are evaluated in the same state $e(\phi_j, s)$ according to the evaluation function of the one-state semantics.*

Let $A^{pre}_{\phi_1,\ldots,\phi_n} \to B^{pre}_{\phi_1,\ldots,\phi_n}$ logically imply $A^{con}_{\phi_1,\ldots,\phi_n} \to B^{con}_{\phi_1,\ldots,\phi_n}$ in one-state semantics, that is $\vDash (A^{pre}_{\phi_1,\ldots,\phi_n} \to B^{pre}_{\phi_1,\ldots,\phi_n}) \to (A^{con}_{\phi_1,\ldots,\phi_n} \to B^{con}_{\phi_1,\ldots,\phi_n})$.

Then also $\vDash_{2|1} (A^{pre}_{\phi_1,\ldots,\phi_n} \to B^{pre}_{\phi_1,\ldots,\phi_n}) \to (A^{con}_{\phi_1,\ldots,\phi_n} \to B^{con}_{\phi_1,\ldots,\phi_n})$ holds.

Proof. Let $P_{\phi_1,\ldots,\phi_n}$ be defined as $A^{pre}_{\phi_1,\ldots,\phi_n} \to B^{pre}_{\phi_1,\ldots,\phi_n}$ and let $C_{\phi_1,\ldots,\phi_n}$ be defined as $A^{con}_{\phi_1,\ldots,\phi_n} \to B^{con}_{\phi_1,\ldots,\phi_n}$. We show that $\vDash_{2|1} P_{\psi_1,\ldots,\psi_n} \to C_{\psi_1,\ldots,\psi_n}$ holds for all formulas $\psi_1, \ldots, \psi_n$ under the given conditions, where schema variable ϕ_j is instantiated by formula ψ_j. We fix an arbitrary structure $(\mathcal{D}, s_1, s_2)$ and a variable assignment β. Let $m : \{\psi_1, \ldots, \psi_n\} \to \{true, false\}$ be the function defined by

$$m(\psi_j) = \begin{cases} true & \text{if } \mathcal{D}, e(\phi_j, s_1), e(\phi_j, s_2), \beta \vDash_{2|1} \psi_j \\ false & \text{otherwise} \end{cases}.$$

Because the ϕ_j (and therefore the ψ_j) do not occur in the scope of quantifiers and because all occurrences of ϕ_j in $P_{\phi_1,\ldots,\phi_n}$ and $C_{\phi_1,\ldots,\phi_n}$ are evaluated in the same state $e(\phi_j, s)$ if $P_{\phi_1,\ldots,\phi_n}$ and $C_{\phi_1,\ldots,\phi_n}$ are evaluated in the same state s (according to the evaluation function of the one-state semantics), $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} P_{\psi_1,\ldots,\psi_n} \to C_{\psi_1,\ldots,\psi_n}$ holds if and only if

$$\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} P_{m(\psi_1),\ldots,m(\psi_n)} \to C_{m(\psi_1),\ldots,m(\psi_n)} \tag{7.32}$$

holds, where the schema variables ϕ_j are instantiated by $m(\psi_j)$ for all j. By the premiss of the lemma, Lemma 49.2, Lemma 49.4, Lemma 49.3c, Lemma 49.3d, Lemma 49.3e or Lemma 49.3f is applicable on $P_{m(\phi_1),\ldots,m(\phi_n)} \to C_{m(\phi_1),\ldots,m(\phi_n)}$. Therefore, Equation (7.32) holds if

$$\mathcal{D}, s_1, \beta \vDash P_{m(\psi_1),\ldots,m(\psi_n)} \to C_{m(\psi_1),\ldots,m(\psi_n)} \tag{7.33}$$

holds. (7.33) is true by the premiss of the lemma. □

Theorem 52. *Let*

$$\frac{\Gamma, A^{pre}_{\phi_1,\ldots,\phi_n} \Longrightarrow B^{pre}_{\phi_1,\ldots,\phi_n}, \Delta}{\Gamma, A^{con}_{\phi_1,\ldots,\phi_n} \Longrightarrow B^{con}_{\phi_1,\ldots,\phi_n}, \Delta}$$

be a non-splitting rule schema with formula schema variables $\phi_1, \ldots, \phi_n$ such that the premiss and conclusion of the rule schema are interpreted in the same structure (in other words: such that no fresh symbols are introduced).

If the requirements of Lemma 51 are met then the rule is sound in two-state semantics.

Proof. By Lemma 50, it is sufficient to show that $\vDash_{2\downarrow 1} ((\bigwedge_{\gamma\in\Gamma}\gamma \wedge A^{pre}_{\phi_1,\ldots,\phi_n}) \rightarrow (B^{pre}_{\phi_1,\ldots,\phi_n} \vee \bigvee_{\delta\in\Delta})) \rightarrow ((\bigwedge_{\gamma\in\Gamma}\gamma \wedge A^{con}_{\phi_1,\ldots,\phi_n}) \rightarrow (B^{con}_{\phi_1,\ldots,\phi_n} \vee \bigvee_{\delta\in\Delta}))$ holds.

Let $(\mathcal{D}, s_1, s_2)$ be a two-state structure and let β be a variable assignment. If $\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \bigwedge_{\gamma\in\Gamma}\gamma$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee_{\delta\in\Delta}$ holds, then we are finished. Thus assume the contrary. We have to show

$$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} (A^{pre}_{\phi_1,\ldots,\phi_n} \rightarrow B^{pre}_{\phi_1,\ldots,\phi_n}) \rightarrow (A^{con}_{\phi_1,\ldots,\phi_n} \rightarrow B^{con}_{\phi_1,\ldots,\phi_n}) \,. \tag{7.34}$$

Because we have $\vDash (A^{pre}_{\phi_1,\ldots,\phi_n} \rightarrow B^{pre}_{\phi_1,\ldots,\phi_n}) \rightarrow (A^{con}_{\phi_1,\ldots,\phi_n} \rightarrow B^{con}_{\phi_1,\ldots,\phi_n})$ by the premiss of the lemma, (7.34) follows from Lemma 51. □

The following sections present the rules of the two-state calculus. Their soundness is shown in Section 7.4.

7.3.1 First Order Logic Rules

Figure 7.2 shows the subset of the classical first order logic rules which are sound in two-state semantics and therefore included without modification in the two-state calculus. Merely the tree splitting rules—andRight, orLeft and impRight—are unsound in two-state semantics. They may only be used if a sequent does not contain the $\ltimes$-predicate and modalities (see Lemma 49), thus only in the variants shown in Figure 7.3. Though those variants of the rules are sound, they are applicable only in rare cases. Therefore, Figure 7.4 shows variations of the rules which are sound in two-state semantics even if $\ltimes$ predicates and modalities are present on the sequent. To get an intuition why the three rules $\ltimes$andRight, $\ltimes$orLeft and $\ltimes$impRight look as they do, consider the cut rule. In the usual Java DL calculus the cut rule has the following form:

$$\text{cut}\ \frac{\Gamma, \phi \Longrightarrow \Delta \quad \Gamma, \neg\phi \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

Interpreted in two-state semantics, the premisses cover only two of three possibilities: the first premiss intuitively requires ϕ to hold in both states; the second premiss requires $\neg\phi$ to hold in both states. The case that ϕ holds in the one state and $\neg\phi$ in the other is missing. This can be expressed by $\neg\ltimes\phi$. Adding the third possibility leads to the following cut rule for two-state semantics:

$$\ltimes\text{cut}\ \frac{\Gamma, \neg\ltimes\phi \Longrightarrow \Delta \quad \Gamma, \phi \Longrightarrow \Delta \quad \Gamma, \neg\phi \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\text{andLeft}\ \frac{\Gamma, \phi, \psi \Longrightarrow \Delta}{\Gamma, \phi \wedge \psi \Longrightarrow \Delta}$$

$$\text{orRight}\ \frac{\Gamma \Longrightarrow \phi, \psi, \Delta}{\Gamma \Longrightarrow \phi \vee \psi, \Delta}$$

$$\text{impRight}\ \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \rightarrow \psi, \Delta}$$

$$\text{notLeft}\ \frac{\Gamma \Longrightarrow \phi, \Delta}{\Gamma, \neg\phi \Longrightarrow \Delta}$$

$$\text{notRight}\ \frac{\Gamma, \phi \Longrightarrow \Delta}{\Gamma \Longrightarrow \neg\phi, \Delta}$$

$$\text{close}\ \frac{*}{\Gamma, \phi \Longrightarrow \phi, \Delta}$$

$$\text{closeFalse}\ \frac{*}{\Gamma, \mathit{false} \Longrightarrow \Delta}$$

$$\text{closeTrue}\ \frac{*}{\Gamma \Longrightarrow \mathit{true}, \Delta}$$

$$\text{allRight}\ \frac{\Gamma \Longrightarrow (\phi)[x/c], \Delta}{\Gamma \Longrightarrow \forall x.\phi, \Delta}$$

with c is a new constant which has the same type as x

$$\text{allLeft}\ \frac{\Gamma, \forall x.\phi, [x/t](\phi) \Longrightarrow \Delta}{\Gamma, \forall x.\phi \Longrightarrow \Delta}$$

with $t \in \mathrm{Trm}_{A'}$ ground, $A' \sqsubseteq A$, if x is of type A

$$\text{exLeft}\ \frac{\Gamma, (\phi)[x/c] \Longrightarrow \Delta}{\Gamma, \exists x.\phi \Longrightarrow \Delta}$$

with c is a new constant which has the same type as x

$$\text{exRight}\ \frac{\Gamma \Longrightarrow \exists x.\phi, [x/t](\phi), \Delta}{\Gamma \Longrightarrow \exists x.\phi, \Delta}$$

with $t \in \mathrm{Trm}_{A'}$ ground, $A' \sqsubseteq A$, if x is of type A

Figure 7.2: Two-state calculus: unmodified first-order rules from [Ahrendt et al., Chapter 2].

$$\mathsf{restrictedAndRight}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \wedge \psi, \Delta}$$

where Γ, Δ, ϕ and ψ neither contain the $\ltimes$ operator nor modalities

$$\mathsf{restrictedOrLeft}\ \frac{\Gamma, \phi \Longrightarrow \Delta \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \vee \psi \Longrightarrow \Delta}$$

where Γ, Δ, ϕ and ψ neither contain the $\ltimes$ operator nor modalities

$$\mathsf{restrictedImpLeft}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \rightarrow \psi \Longrightarrow \Delta}$$

where Γ, Δ, ϕ and ψ neither contain the $\ltimes$ operator nor modalities

Figure 7.3: Two-state calculus: restricted first-order rules. The rules are variations of rules from [Ahrendt et al., Chapter 2]

$$\ltimes\mathsf{andRight}\ \frac{\Gamma \Longrightarrow \ltimes\phi, \ltimes\psi, \Delta \quad \Gamma \Longrightarrow \phi, \Delta \quad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \wedge \psi, \Delta}$$

$$\ltimes\mathsf{orLeft}\ \frac{\Gamma \Longrightarrow \ltimes\phi, \ltimes\psi, \Delta \quad \Gamma, \phi \Longrightarrow \Delta \quad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \vee \psi \Longrightarrow \Delta}$$

$$\ltimes\mathsf{impLeft}\ \frac{\Gamma \Longrightarrow \ltimes\phi, \ltimes\psi, \Delta \quad \Gamma \Longrightarrow \phi, \Delta \quad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \rightarrow \psi \Longrightarrow \Delta}$$

Figure 7.4: Two-state calculus: $\ltimes$ variants of first-order rules. The rules are variations of rules from [Ahrendt et al., Chapter 2]

$$\text{emptyModality}\ \frac{\Gamma \Longrightarrow \{u\}\phi, \Delta}{\Gamma \Longrightarrow \{u\}[]\phi, \Delta}$$

$$\text{assignLocal}\ \frac{\Gamma \Longrightarrow \{u\}\{\mathtt{a} := t\}[\pi\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\pi\ \mathtt{a}\ =\ t;\ \omega]\phi, \Delta}$$

$$\text{assignField}\ \frac{\Gamma \Longrightarrow \{u\}\{\mathtt{heap} := \mathit{store}(\mathtt{heap}, o, f, t)\}[\pi\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\pi\ o.\mathtt{f}\ =\ t;\ \omega]\phi, \Delta}$$

$$\text{assignArray}\ \frac{\Gamma \Longrightarrow \{u\}\{\mathtt{heap} := \mathit{store}(\mathtt{heap}, a, \mathit{arr}(i), t)\}[\pi\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\pi\ a\mathtt{[}i\mathtt{]}\ =\ t;\ \omega]\phi, \Delta}$$

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \{u\}[\pi\ \mathtt{if(}g\mathtt{)\ \{}p\mathtt{;\ while(}g\mathtt{)}\ p\mathtt{\};}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\pi\ \mathtt{while(}g\mathtt{)}\ p;\ \omega]\phi, \Delta}$$

Figure 7.5: Two-state calculus: unmodified Java rules from Weiß [2011].

or equivalently

$$\ltimes\text{cut}\ \frac{\Gamma \Longrightarrow \ltimes\phi, \Delta \quad \Gamma, \phi \Longrightarrow \Delta \quad \Gamma \Longrightarrow \phi, \Delta}{\Gamma \Longrightarrow \Delta}$$

The rules $\ltimes$andRight, $\ltimes$orLeft and $\ltimes$impRight can be understood as an application of the $\ltimes$cut rule with subsequent simplification.

7.3.2 Java Rules

Figure 7.5 shows the subset of the classical Java DL rules which are sound in two-state semantics and therefore included without modification in the two-state calculus. Again, mainly the splitting Java rules like the conditional rule have to be adjusted. They are used in the variant shown in Figure 7.6.

The rule $\ltimes$expandMethod is a slight variation of the rule expandMethod from Weiß [2011]: the two premisses of expandMethod are merged into one. The rules $\ltimes$expandMethod and expandMethod are equivalent in one-state semantics.

The rule $\ltimes$conditional has a third premiss compared to the rule conditional. This third premiss guarantees that both runs evaluate the guard of the conditional

$$\ltimes\text{expandMethod}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \quad \{u\}\mathit{exactInstance}_A(o) \\ \qquad \wedge\{u\}[\pi\ \texttt{method-frame(result=r, this=o)}: \\ \qquad\qquad \{\ \ \mathit{body}(\mathtt{m}, A)\ \ \}\ \omega]\phi, \Delta\end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \texttt{r = o.m();}\ \omega]\phi, \Delta}$$

$$\ltimes\text{conditional}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \{u\}\ltimes(g), \Delta \\ \Gamma, \{u\}g \doteq \texttt{true} \Longrightarrow \{u\}[\pi\ p_1;\ \omega]\phi, \Delta \\ \Gamma, \{u\}g \doteq \texttt{false} \Longrightarrow \{u\}[\pi\ p_2;\ \omega]\phi, \Delta\end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \texttt{if(}g\texttt{)}\ p_1\ \texttt{else}\ p_2;\ \omega]\phi, \Delta}$$

$$\ltimes\text{extConditional}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \{u\}[\pi\ \texttt{if(}g\texttt{)}\ p_1\ \texttt{else}\ p_2](\mathit{Inv} \wedge \ltimes(\mathit{Inv})), \Delta \\ \mathit{Inv} \Longrightarrow [\pi\ \omega]\phi\end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \texttt{if(}g\texttt{)}\ p_1\ \texttt{else}\ p_2;\ \omega]\phi, \Delta}$$

for an arbitrary formula Inv

$$\ltimes\text{loopInvariant}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \{u\}(\mathit{Inv} \wedge \ltimes(\mathit{Inv}) \wedge \ltimes(g)), \Delta \\ \mathit{Inv}, g \doteq \texttt{true} \Longrightarrow [p](\mathit{Inv} \wedge \ltimes(\mathit{Inv}) \wedge \ltimes(g)) \\ \mathit{Inv}, g \doteq \texttt{false} \Longrightarrow [\pi\ \omega]\phi\end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \texttt{while(}g\texttt{)}\ p;\ \omega]\phi, \Delta}$$

for an arbitrary formula Inv

$$\ltimes\text{extLoopInvariant}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \{u\}[\pi\ \texttt{while(}g\texttt{)}\ p](\mathit{Inv} \wedge \ltimes(\mathit{Inv})), \Delta \\ \mathit{Inv} \Longrightarrow [\pi\ \omega]\phi\end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \texttt{while(}g\texttt{)}\ p;\ \omega]\phi, \Delta}$$

for an arbitrary formula Inv

$$\ltimes\text{createObj}\ \frac{\begin{array}{l}\Gamma, o' \doteq \texttt{null}, \mathit{exactInstance}_A(o'), \\ \{u\}(\quad \mathit{wellFormed}(\texttt{heap}) \\ \qquad \rightarrow \mathit{select}_{Boolean}(\texttt{heap}, o', \mathit{created}) \doteq \texttt{false}) \\ \Longrightarrow \{u\}\{\texttt{heap} := \mathit{create}(\texttt{heap}, o')\}\{\texttt{o} := o'\}[\pi\ \omega]\phi, \Delta\end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \texttt{o = A.alloc();}\ \omega]\phi, \Delta}$$

where $o' : A$ is a fresh program variable.

Figure 7.6: Two-state calculus: special Java rules. $\ltimes$createObj, $\ltimes$expandMethod, $\ltimes$conditional and $\ltimes$loopInvariant are variations of rules from Weiß [2011].

in the same way. Thus, it is excluded that the one run takes the then branch whereas the second one takes the else branch (which would not be expressible in the two-state logic). To be able to reason about programs with high guards, the rule $\ltimes$extConditional is introduced. The rule decouples the reasoning about the conditional from the reasoning about the rest of the program. In this way it is possible to use precise self-composition style reasoning for the conditional statement (see Section 7.3.5) and the approximate calculus for the rest of the program. The first premiss of $\ltimes$extConditional guarantees that the formula Inv is valid after the execution of the conditional "in both final states", that is, $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} Inv$ and $\mathcal{D}, s_2^\alpha, s_1^\alpha, \beta \vDash_{2\downarrow 1} Inv$ (and by the way also $\mathcal{D}, s_1^\alpha, s_1^\alpha, \beta \vDash_{2\downarrow 1} Inv$) hold if the execution of the conditional terminates in states s_1^α and s_2^α, respectively. (Inv ensures that $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} Inv$ or $\mathcal{D}, s_2^\alpha, s_1^\alpha, \beta \vDash_{2\downarrow 1} Inv$ holds and $\ltimes(Inv)$ makes an "and" out of the "or".) The second premiss ensures that if $\mathcal{D}, s_3, s_3, \beta \vDash_{2\downarrow 1} Inv$, $\mathcal{D}, s_3, s_4, \beta \vDash_{2\downarrow 1} Inv$ and $\mathcal{D}, s_4, s_3, \beta \vDash_{2\downarrow 1} Inv$ hold for arbitrary states s_3 and s_4, then $\mathcal{D}, s_3^\alpha, s_3^\alpha, \beta \vDash_{2\downarrow 1} \phi$ and either $\mathcal{D}, s_3^\alpha, s_4^\alpha, \beta \vDash_{2\downarrow 1} \phi$ or $\mathcal{D}, s_4^\alpha, s_3^\alpha, \beta \vDash_{2\downarrow 1} \phi$ hold if the execution of the rest of the program terminates in states s_3^α and s_4^α, respectively. Together this implies the conclusion. Note that the presented rule loses the context Γ and Δ. As usual, it is possible to design a rule which preserves the context as far as possible with the help of anonymizing updates.

The loop invariant rule $\ltimes$loopInvariant does not differ much from the usual loop invariant rule (in its simplest form). Similar to $\ltimes$extConditional, the first premiss of the rule ensures that Inv holds "in both states" before the loop and similar to $\ltimes$conditional that the guard of the loop evaluates in the same way in those states. The second premiss ensures that these additional requirements are preserved by the loop body. As in the case of $\ltimes$extConditional the rule looses the context Γ and Δ. Again it can be preserved by the usage of anonymizing updates, see for instance Weiß [2011]. The rule $\ltimes$extLoopInvariant allows self-composition style reasoning for loops with high guards in the same fashion as the rule $\ltimes$extConditional allows this for conditional statements.

The only difference between the rule $\ltimes$createObj and the original rule createObj from Weiß [2011] is that o' is a fresh program variable instead of a fresh function symbol. In one-state semantics this makes no difference. In two-state semantics there is a difference: a fresh function symbol would be interpreted by $\mathcal{D}$ and thus in the same way for the two runs. There is, however, no reason why the two runs should allocate the same new object. A fresh program variable, on the other hand, is interpreted by the two states potentially differently. Note that the only change to be made in the calculus to check for Strong Object-Sensitive Noninterference (Definition 25) instead of Conditional Noninterference (Definition 3) is to use the original rule createObj from Weiß [2011] instead of $\ltimes$createObj.

$$\ltimes\mathsf{not}\quad \ltimes(\neg\phi) \rightsquigarrow \ltimes(\phi)$$

$$\ltimes\mathsf{approxPred}\ \frac{\Gamma \Longrightarrow \ltimes(t_1) \wedge \ldots \wedge \ltimes(t_n), \Delta}{\Gamma \Longrightarrow \ltimes p(t_1, \ldots, t_n), \Delta}$$

where p is a predicate

$$\ltimes\mathsf{approxAnd}\ \frac{\Gamma \Longrightarrow \ltimes(\phi) \wedge \ltimes(\psi), \Delta}{\Gamma \Longrightarrow \ltimes(\phi \wedge \psi), \Delta}$$

$$\ltimes\mathsf{constant}\quad \ltimes(c) \rightsquigarrow \mathit{true}$$

where c is a constant

$$\ltimes\mathsf{approxOr}\ \frac{\Gamma \Longrightarrow \ltimes(\phi) \wedge \ltimes(\psi), \Delta}{\Gamma \Longrightarrow \ltimes(\phi \vee \psi), \Delta}$$

$$\ltimes\mathsf{approxFunc}\ \frac{\Gamma \Longrightarrow \ltimes(t_1) \wedge \ldots \wedge \ltimes(t_n), \Delta}{\Gamma \Longrightarrow \ltimes f(t_1, \ldots, t_n), \Delta}$$

where f is a function

$$\ltimes\mathsf{approxEq}\ \frac{\Gamma \Longrightarrow \ltimes(s) \wedge \ltimes(t), \Delta}{\Gamma \Longrightarrow \ltimes(s = t), \Delta}$$

Figure 7.7: Two-state calculus: rules approximating $\ltimes$.

Figures 7.5 and 7.6 show the box variants of the rules only. The diamond variants can be constructed analog.

7.3.3 Update Simplification Rules

The two-state calculus uses the same update simplification rules as Weiß [2011] and Rümmer [2006]. Their soundness is ensured by Theorem 52.

7.3.4 $\ltimes$ Approximation Rules

Sometimes it is useful if $\ltimes$formulas can be simplified. For instance, the universal validity of $\ltimes(x) \Longrightarrow \ltimes(x+1)$ cannot be shown with the help of the rules discussed so far. Figure 7.7 shows rules for the approximation of $\ltimes$formulas. With the help of these rules the universality of $\ltimes(x) \Longrightarrow \ltimes(x+1)$ can be shown:

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{\ltimes(x) \Longrightarrow \ltimes(x)}\,\text{close}}{\ltimes(x) \Longrightarrow \ltimes(x) \wedge \mathit{true}}\,\text{andTrue}}{\ltimes(x) \Longrightarrow \ltimes(x) \wedge \ltimes(1)}\,\ltimes\text{constant}}{\ltimes(x) \Longrightarrow \ltimes(x+1)}\,\ltimes\text{approxFunc}$$

The rules from Figure 7.7 handle simple cases efficiently. If higher precision is necessary, it is possible to switch to self-composition style reasoning instead. The next Section shows how self-composition style reasoning can be used with the help of an on-the-fly conversion to one-state semantics.

7.3.5 Conversion to One-State Semantics

The two-state semantics can be unfolded into one-state semantics according to Definitions 33 (Two-State Evaluation) and Definition 31 (Restricted Two-State Evaluation). The following rule preforms such a conversion. The transformation triplicates each formula on the sequent and is closely related to self composition.

$$\dfrac{\begin{array}{ll} \Longrightarrow \forall h.\forall h_2.\forall \bar{x}.\forall \bar{x}_2. & \\ \quad (\bigwedge_{1\le i\le n} \gamma_i^{2|1} \wedge \bigwedge_{1\le i\le n} \gamma_i^{2|2}) & \gamma_1^1,\ldots,\gamma_n^1 \\ \quad \rightarrow (\bigvee_{1\le j\le m} \delta_j^{2|1} \vee \bigvee_{1\le j\le m} \delta_j^{2|2}) & \Longrightarrow \delta_1^1,\ldots,\delta_m^1 \end{array}}{\gamma_1,\ldots,\gamma_n \Longrightarrow \delta_1,\ldots,\delta_m}\,\text{toOneState}$$

where γ_i^1 and δ_j^1 equal γ_i and δ_j, respectively, except that all $\ltimes$predicates are replaced by true and where h, $\bar{x}$, h_2, $\bar{x}_2$ do not occur free in γ_i and δ_j. The construction of $\gamma_i^{2|1}$, $\gamma_i^{2|2}$, $\delta_j^{2|1}$ and $\delta_j^{2|2}$ is more involved, at least if those formulas contain modalities. We consider the construction for $\delta_j^{2|1}$. The formula $\gamma_j^{2|1}$ is constructed in the same way as $\delta_j^{2|1}$ and the only difference in the construction of $\gamma_i^{2|2}$ and $\delta_i^{2|2}$ is in step 1, where the updates $\{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}\{\texttt{heap} := h_2 \parallel \bar{\texttt{x}} := \bar{x}_2\}_2$ have to be replaced by $\{\texttt{heap} := h_2 \parallel \bar{\texttt{x}} := \bar{x}_2\}\{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}_2$.

To describe the intermediate steps of the transformation it is useful to extend the two-state logic by updates which effect the second state only. Therefore, we enrich the syntax of the two-state logic by formulas $\{u\}_2\phi$, where ϕ is a formula and u has the same syntax as in ordinary updates. Their semantics is defined by

an extension of the restricted two-state evaluation: $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \{u\}_2\phi$ holds if and only if $\mathcal{D}, s_1, s_2^u, \beta \vDash_{2|1} \phi$ holds, where s_2^u results from s_2 by application of $\{u\}$. The updates $\{u\}_2$ are useful to describe the intermediate steps of the transformation, but do not occur in $\delta_j^{2|1}$ any more.

The formula δ_j is transformed in several steps from the outside to the inside.

1. Construct $\delta_j^1 = \{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}\{\texttt{heap} := h_2 \parallel \bar{\texttt{x}} := \bar{x}_2\}_2\delta_j$.

   Example. *Consider the sequent* $l > 0 \land \ltimes(l) \Longrightarrow [\alpha]\ltimes(l)$. *The first step transforms* $\gamma_1 = l > 0 \land \ltimes(l)$ *to* $\{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}\{\texttt{heap} := h_2 \parallel \bar{\texttt{x}} := \bar{x}_2\}_2(l > 0 \land \ltimes(l))$ *and* $\delta_1 = [\alpha]\ltimes(l)$ *to* $\{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}\{\texttt{heap} := h_2 \parallel \bar{\texttt{x}} := \bar{x}_2\}_2[\alpha]\ltimes(l)$.

2. Shift all updates in δ_j^1 to the arguments of $\land$, $\lor$, $\neg$, $\forall$ and $\exists$ till they appear solely in front of $\ltimes$operators, modalities, predicates or other updates. (If an update contains a free variable which would be bound by shifting the update in the scope of the quantifier, then rename the variable of the quantifier beforehand.)

   Example. *Because* $\ltimes(l)$ *is only a short form for* $\forall y.\ltimes(y = l)$, *the second step transforms* $\{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}\{\texttt{heap} := h_2 \parallel \bar{\texttt{x}} := \bar{x}_2\}_2(l > 0 \land \ltimes(l))$ *to* $\{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}\{\texttt{heap} := h_2 \parallel \bar{\texttt{x}} := \bar{x}_2\}_2(l > 0) \land \forall y.\{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}\{\texttt{heap} := h_2 \parallel \bar{\texttt{x}} := \bar{x}_2\}_2\ltimes(y = l)$. *The second formula* $\{\texttt{heap} := h \parallel \bar{\texttt{x}} := \bar{x}\}\{\texttt{heap} := h_2 \parallel \bar{\texttt{x}} := \bar{x}_2\}_2[\alpha]\ltimes(l)$ *is not modified.*

3. If there are updates in front of $\ltimes$operators, modalities, and predicates of the form $\{u\}\{u_2\}_2\{v_1\}\ldots\{v_n\}$ (where $\{u\}$ and $\{u_2\}_2$ update the complete state, respectively), transform them into $\{u; v_1; \ldots; v_n\}\{u_2; v_1; \ldots; v_n\}_2$.

   Example. *In the example there has nothing to be done.*

4. As long as the resulting formula contains subformulas

   (1) $\{u\}\{u_2\}_2\ltimes\phi$,

   (2) $\{u\}\{u_2\}_2[\alpha]\phi$,

   (3) $\{u\}\{u_2\}_2\langle\alpha\rangle\phi$ or

   (4) $\{u\}\{u_2\}_2\phi_p$, where ϕ_p is a predicate,

   replace

   (1) $\{u\}\{u_2\}_2\ltimes\phi$ by $\{u\}\{u_2\}_2\phi \leftrightarrow \{u_2\}\{u\}_2\phi$,

(2) $\{u\}\{u_2\}_2[\alpha]\phi$ by

$$\begin{aligned}&\forall h_{post}.\forall h_{post2}.\forall \bar{x}_{post}.\forall \bar{x}_{post2}.\\ &\quad \{u\}\langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post})\\ &\wedge \{u_2\}\langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2})\\ &\rightarrow \{\mathtt{heap} := h_{post} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post}\}\{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2\phi\end{aligned}$$

(3) $\{u\}\{u_2\}_2\langle\alpha\rangle\phi$ by

$$\begin{aligned}&\exists h_{post}.\exists h_{post2}.\exists \bar{x}_{post}.\exists \bar{x}_{post2}.\\ &\quad \{u\}\langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post})\\ &\wedge \{u_2\}\langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2})\\ &\wedge \{\mathtt{heap} := h_{post} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post}\}\{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2\phi\end{aligned}$$

and

(4) $\{u\}\{u_2\}_2\phi_p$, where ϕ_p is a predicate, by $\{u\}\phi_p$.

Example. *The fourth step transforms* $\{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}\{\mathtt{heap} := h_2 \mid\mid \bar{\mathtt{x}} := \bar{x}_2\}_2(l > 0) \wedge \forall y.\{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}\{\mathtt{heap} := h_2 \mid\mid \bar{\mathtt{x}} := \bar{x}_2\}_2 \ltimes (y = l)$ *to* $\{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}(l > 0) \wedge \forall y.(\{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}(y = l) \leftrightarrow \{\mathtt{heap} := h_2 \mid\mid \bar{\mathtt{x}} := \bar{x}_2\}(y = l))$ *and* $\{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}\{\mathtt{heap} := h_2 \mid\mid \bar{\mathtt{x}} := \bar{x}_2\}_2[\alpha] \ltimes (l)$ *to*

$$\begin{aligned}&\forall h_{post}.\forall h_{post2}.\forall \bar{x}_{post}.\forall \bar{x}_{post2}.\\ &\quad \{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}\langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post})\\ &\wedge \{\mathtt{heap} := h_2 \mid\mid \bar{\mathtt{x}} := \bar{x}_2\}\langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2})\\ &\rightarrow \{\mathtt{heap} := h_{post} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post}\}\{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2 \ltimes (l)\end{aligned}$$

The subformula $\{\mathtt{heap} := h_{post} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post}\}\{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2 \ltimes (l)$ *is not replaced yet, because* $\ltimes (l)$ *is only a short form for* $\forall y. \ltimes (y = l)$.

5. If the resulting formula still contains updates $\{u\}_2$, then repeat starting at step 2. Otherwise the construction terminates and the resulting formula is $\delta_j^{2\downarrow 1}$.

Example. *Another repetition of steps 2 to 5 yields*

$$\begin{aligned}\gamma_1^{2\downarrow 1} = \quad &\{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}(l > 0)\\ &\wedge \forall y.(\quad \{\mathtt{heap} := h \mid\mid \bar{\mathtt{x}} := \bar{x}\}(y = l)\\ &\qquad \leftrightarrow \{\mathtt{heap} := h_2 \mid\mid \bar{\mathtt{x}} := \bar{x}_2\}(y = l)\ )\end{aligned}$$

(the second round did not affect this formula any more) and

$$\begin{aligned}\delta_1^{2|1} = \forall h_{post}.\forall h_{post2}.\forall \bar{x}_{post}.\forall \bar{x}_{post2}.\\ \{\texttt{heap} := h \mid\mid \bar{\texttt{x}} := \bar{x}\}\langle\alpha\rangle(\texttt{heap} = h_{post} \wedge \bar{\texttt{x}} = \bar{x}_{post})\\ \wedge \{\texttt{heap} := h_2 \mid\mid \bar{\texttt{x}} := \bar{x}_2\}\langle\alpha\rangle(\texttt{heap} = h_{post2} \wedge \bar{\texttt{x}} = \bar{x}_{post2})\\ \rightarrow \forall y.(\quad \{\texttt{heap} := h_{post} \mid\mid \bar{\texttt{x}} := \bar{x}_{post}\}(y = l)\\ \leftrightarrow \{\texttt{heap} := h_{post2} \mid\mid \bar{\texttt{x}} := \bar{x}_{post2}\}(y = l)\ )\end{aligned}$$

Because $\forall y.(\{u\}(y = l) \leftrightarrow \{u_2\}(y = l))$ *equals* $\{u\}l = \{u_2\}l$ *(see Lemma 32)* $\gamma_1^{2|1}$ *and* $\delta_1^{2|1}$ *can be rewritten to*

$$\begin{aligned}\gamma_1^{2|1} = \quad \{\texttt{heap} := h \mid\mid \bar{\texttt{x}} := \bar{x}\}(l > 0)\\ \wedge \{\texttt{heap} := h \mid\mid \bar{\texttt{x}} := \bar{x}\}l = \{\texttt{heap} := h_2 \mid\mid \bar{\texttt{x}} := \bar{x}_2\}l\end{aligned}$$

and

$$\begin{aligned}\delta_1^{2|1} = \forall h_{post}.\forall h_{post2}.\forall \bar{x}_{post}.\forall \bar{x}_{post2}.\\ \{\texttt{heap} := h \mid\mid \bar{\texttt{x}} := \bar{x}\}\langle\alpha\rangle(\texttt{heap} = h_{post} \wedge \bar{\texttt{x}} = \bar{x}_{post})\\ \wedge \{\texttt{heap} := h_2 \mid\mid \bar{\texttt{x}} := \bar{x}_2\}\langle\alpha\rangle(\texttt{heap} = h_{post2} \wedge \bar{\texttt{x}} = \bar{x}_{post2})\\ \rightarrow \quad \{\texttt{heap} := h_{post} \mid\mid \bar{\texttt{x}} := \bar{x}_{post}\}l\\ = \{\texttt{heap} := h_{post2} \mid\mid \bar{\texttt{x}} := \bar{x}_{post2}\}l\end{aligned}$$

$\gamma_1^{2|2}$ *and* $\delta_1^{2|2}$ *are constructed analog. We have*

$$\begin{aligned}\gamma_1^{2|2} = \quad \{\texttt{heap} := h_2 \mid\mid \bar{\texttt{x}} := \bar{x}_2\}(l > 0)\\ \wedge \{\texttt{heap} := h_2 \mid\mid \bar{\texttt{x}} := \bar{x}_2\}l = \{\texttt{heap} := h \mid\mid \bar{\texttt{x}} := \bar{x}\}l\end{aligned}$$

and $\delta_1^{2|2} = \delta_1^{2|1}$.

The formulas occurring in the premiss of the rule toOneState do not contain the $\ltimes$predicate any more and each modality is prefixed by an update of the form $\{\texttt{heap} := h \mid\mid \bar{\texttt{x}} := \bar{x}\}$, where h and $\bar{x}$ are variables. Thus, the premiss can safely be handled by the usual Java DL calculus (Lemma 49.6) if it is not possible to reintroduce $\ltimes$ to the sequent (for instance by hide and reinsert rules).

Note that the two-state calculus mainly excludes infeasible combinations of paths through the program and at some point approximates $\ltimes$ formulas. Values of program variables are abstracted by invariants and contracts only and thus can be kept highly precise. Switching to self-composition style preserves this very precise knowledge on *both* program states. Thus, the self-composition style reasoning approach from Chapter 5 can build on precise knowledge to draw precise conclusions.

The rule toOneState is very complex. The transformation can be performed in smaller steps on the basis of predicate transformers. Figures 7.8 and 7.9 show a set of conversion rules based on predicate transformers. Their soundness proof, however, is out of scope of this thesis.

7.4 Soundness of the Two-State Calculus

The following lemmas show the soundness of the two-state calculus.

Lemma 53. *The rules from Figure 7.2 are sound with respect to two-state semantics.*

Proof. The soundness of the rules andLeft and orRight follows directly by the definition of the meaning formulas (Definition 37) of the premisses and the conclusions of the rules.

The rules impRight, notLeft, notRight, close, closeTrue and closeFalse fulfill the requirements of Theorem 52 and therefore are sound in two-state semantics.

The soundness of the rules allRight, allLeft, exRight and exLeft remains to be shown.

- allRight: We have to show that

$$\vDash_2 \bigwedge \Gamma \to ((\phi)[x/c] \vee \bigvee \Delta) \tag{7.35}$$

  implies

$$\vDash_2 \bigwedge \Gamma \to ((\forall x.\phi) \vee \bigvee \Delta)\,. \tag{7.36}$$

  By Definition 33, equation (7.36) holds if and only if

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to ((\forall x.\phi) \vee \bigvee \Delta) \\ \text{and } \big(\quad &\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to ((\forall x.\phi) \vee \bigvee \Delta) \\ \text{or } &\mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to ((\forall x.\phi) \vee \bigvee \Delta)\big) \end{aligned} \tag{7.37}$$

  for all two-state structures $(\mathcal{D}, s_1, s_2)$ and all variable assignments β. Let $(\mathcal{D}, s_1, s_2)$ be an arbitrary two-state structure and let β be a variable assignment. If

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \bigvee \Delta \\ \text{and } \big(\quad &\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta \\ \text{or } &\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta\big) \end{aligned}$$

$$\textsf{pToOneState}\ \frac{\begin{array}{l} \lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \gamma_1 \gtrdot_{2\downarrow 1}, \\ \ldots, \\ \lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \gamma_n \gtrdot_{2\downarrow 1} \\ \lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \gamma_1 \gtrdot_{2\downarrow 2}, \\ \ldots, \\ \lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \gamma_n \gtrdot_{2\downarrow 2} \\ \Longrightarrow \lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \delta_1 \gtrdot_{2\downarrow 1}, \\ \ldots, \\ \lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \delta_m \gtrdot_{2\downarrow 1} \\ \lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \delta_1 \gtrdot_{2\downarrow 2}, \\ \ldots, \\ \lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \delta_m \gtrdot_{2\downarrow 2} \end{array} \quad \begin{array}{l} \\ \\ \\ \lessdot \gamma_1 \gtrdot_{elim \ltimes}, \\ \ldots, \\ \lessdot \gamma_n \gtrdot_{elim \ltimes} \\ \Longrightarrow \lessdot \delta_1 \gtrdot_{elim \ltimes}, \\ \ldots, \\ \lessdot \delta_m \gtrdot_{elim \ltimes} \\ \\ \\ \\ \end{array}}{\gamma_1, \ldots, \gamma_n \Longrightarrow \delta_1, \ldots, \delta_m}$$

where h^1, h^2 are fresh heap symbols and $\bar{x}^1, \bar{x}^2$ are fresh function symbols, two for each program variable.

$$\textsf{pTwoStateLogicalOp}\ \frac{\lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \phi_1 \gtrdot_{2\downarrow i} \circ \lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \phi_2 \gtrdot_{2\downarrow i}}{\lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \phi_1 \circ \phi_2 \gtrdot_{2\downarrow i}}$$

where $\circ$ is a logical operator.

$$\textsf{pElimAgreeLogicalOp}\ \frac{\lessdot \phi_1 \gtrdot_{elim \ltimes} \circ \lessdot \phi_2 \gtrdot_{elim \ltimes}}{\lessdot \phi_1 \circ \phi_2 \gtrdot_{elim \ltimes}}$$

where $\circ$ is a logical operator.

$$\textsf{pTwoStateAgree}\ \frac{\{\bar{\mathtt{x}} := \bar{x}^1 \parallel \mathtt{heap} := h^1\}\phi = \{\bar{\mathtt{x}} := \bar{x}^2 \parallel \mathtt{heap} := h^2\}\phi}{\lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \ltimes \phi \gtrdot_{2\downarrow i}}$$

$$\textsf{pElimAgree}\ \frac{\mathit{true}}{\lessdot \ltimes \phi \gtrdot_{elim \ltimes}}$$

Figure 7.8: Transformation from two-state semantics to one-state semantics with the help of predicate transformers. (Part 1)

$$\mathsf{pTwoStatePredicate}\ \frac{\{\bar{\mathtt{x}} := \bar{x}^i \parallel \mathtt{heap} := h^i\}p(\bar{t})}{\lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, p(\bar{t}) \gtrdot_{2|i}}$$

where p is a predicate and $\bar{t}$ terms of proper type.

$$\mathsf{pElimAgreePredicate}\ \frac{p(\bar{t})}{\lessdot p(\bar{t}) \gtrdot_{elim\ltimes}}$$

where p is a predicate and $\bar{t}$ terms of proper type.

$$\mathsf{pTwoStateUpdate}\ \frac{\lessdot \{u\}\bar{x}^1, \{u\}h^1, \{u\}\bar{x}^2, \{u\}h^2, \phi \gtrdot_{2|i}}{\lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, \{u\}\phi \gtrdot_{2|i}}$$

$$\mathsf{pElimAgreeUpdate}\ \frac{\{u\} \lessdot \phi \gtrdot_{elim\ltimes}}{\lessdot \{u\}\phi \gtrdot_{elim\ltimes}}$$

$$\mathsf{pTwoStateModality}\ \frac{\begin{array}{r}\{\bar{\mathtt{x}} := \bar{x}^1 \parallel \mathtt{heap} := h^1\}[p](\bar{x} = \bar{x}^1_{post} \wedge \mathtt{heap} = h^1_{post}) \\ \wedge\ \{\bar{\mathtt{x}} := \bar{x}^2 \parallel \mathtt{heap} := h^2\}[p](\bar{x} = \bar{x}^2_{post} \wedge \mathtt{heap} = h^2_{post}) \\ \rightarrow\ \lessdot \bar{x}^1_{post}, {h_{post}}^1, \bar{x}^2_{post}, {h_{post}}^2, \phi \gtrdot_{2|i}\end{array}}{\lessdot \bar{x}^1, h^1, \bar{x}^2, h^2, [p]\phi \gtrdot_{2|i}}$$

where h^1_{post}, h^2_{post} are fresh heap symbols and $\bar{x}^1_{post}, \bar{x}^2_{post}$ are fresh function symbols, two for each program variable.

$$\mathsf{pElimAgreeModality}\ \frac{[p] \lessdot \phi \gtrdot_{elim\ltimes}}{\lessdot [p]\phi \gtrdot_{elim\ltimes}}$$

Figure 7.9: Transformation from two-state semantics to one-state semantics with the help of predicate transformers. (Part 2)

hold, then we are done. Hence, assume the contrary.

Again by Definition 33, equation (7.35) holds if and only if

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to ((\phi)[x/c] \vee \bigvee \Delta) \\ \text{and } \big(\quad &\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to ((\phi)[x/c] \vee \bigvee \Delta) \\ \text{or } &\mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to ((\phi)[x/c] \vee \bigvee \Delta)\big) \end{aligned} \tag{7.38}$$

for all two-state structures $(\mathcal{D}, s_1, s_2)$ and variable assignments β. Thus, we may assume

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} (\phi)[x/c] \\ \text{and } &\big(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} (\phi)[x/c] \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} (\phi)[x/c]\big) \ . \end{aligned} \tag{7.39}$$

We define $S_{\mathcal{D}} = \{\mathcal{D}' \mid c^{\mathcal{D}'} \in D \text{ and all other symbols interpreted as in } \mathcal{D}\}$. Because c is a fresh constant symbol and therefore does neither occur in $\bigwedge \Gamma$ nor in $\bigvee \Delta$, we also have for all $\mathcal{D}' \in S_{\mathcal{D}}$ that

$$\begin{aligned} &\mathcal{D}', s_1, s_1, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}', s_1, s_1, \beta \vDash_{2\downarrow 1} \bigvee \Delta \\ \text{and } (\quad &\mathcal{D}', s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}', s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta \\ \text{or } &\mathcal{D}', s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}', s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta) \end{aligned}$$

does not hold and hence

$$\begin{aligned} &\mathcal{D}', s_1, s_1, \beta \vDash_{2\downarrow 1} (\phi)[x/c] \\ \text{and } &\big(\mathcal{D}', s_1, s_2, \beta \vDash_{2\downarrow 1} (\phi)[x/c] \text{ or } \mathcal{D}', s_2, s_1, \beta \vDash_{2\downarrow 1} (\phi)[x/c]\big) \end{aligned} \tag{7.40}$$

holds.

Let β' be defined as $\beta'(y) = ((y)[x/c])^{\mathcal{D}', s_i, \beta}$ for all variables y. Then, (7.40) is equivalent to

$$\begin{aligned} &\mathcal{D}', s_1, s_1, \beta' \vDash_{2\downarrow 1} \phi \\ \text{and } &\big(\mathcal{D}', s_1, s_2, \beta' \vDash_{2\downarrow 1} \phi \text{ or } \mathcal{D}', s_2, s_1, \beta' \vDash_{2\downarrow 1} \phi\big) \end{aligned} \tag{7.41}$$

for all $\mathcal{D}' \in S_{\mathcal{D}}$. By the definition of $S_{\mathcal{D}}$, for all $d \in D$ there is a structure $\mathcal{D}' \in S_{\mathcal{D}}$ such that $c^{\mathcal{D}'} = d$. Therefore,

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta^{x/d} \vDash_{2\downarrow 1} \phi \\ \text{and } &\big(\mathcal{D}, s_1, s_2, \beta^{x/d} \vDash_{2\downarrow 1} \phi \text{ or } \mathcal{D}, s_2, s_1, \beta^{x/d} \vDash_{2\downarrow 1} \phi\big) \end{aligned} \tag{7.42}$$

for all $d \in D$ and hence

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \forall x.\phi \\ \text{and } &\left(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \forall x.\phi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \forall x.\phi\right). \end{aligned} \tag{7.43}$$

Equation (7.43) implies (7.37), which finishes the proof.

- exLeft:

  The soundness of exLeft can be reduced to the soundness of allRight. The premiss can be rearranged as follows:

$$\begin{aligned} (\bigwedge \Gamma \wedge (\phi)[x/c]) \rightarrow \bigvee \Delta \quad &\equiv_{2\downarrow 1} \neg(\bigwedge \Gamma) \vee \neg(\phi)[x/c] \vee \bigvee \Delta \\ \equiv_{2\downarrow 1} \neg(\bigwedge \Gamma) \vee [x/c](\neg\phi) \vee \bigvee \Delta \quad &\equiv_{2\downarrow 1} \bigwedge \Gamma \rightarrow ([x/c](\neg\phi) \vee \bigvee \Delta) \end{aligned}$$

  Similarly, the conclusion can be rearranged:

$$\begin{aligned} (\bigwedge \Gamma \wedge (\exists x.\phi)) \rightarrow \bigvee \Delta \quad &\equiv_{2\downarrow 1} \neg(\bigwedge \Gamma) \vee \neg(\exists x.\phi) \vee \bigvee \Delta \\ \equiv_{2\downarrow 1} \neg(\bigwedge \Gamma) \vee (\forall x.\neg\phi) \vee \bigvee \Delta \quad &\equiv_{2\downarrow 1} \bigwedge \Gamma \rightarrow (\forall x.\neg\phi) \vee \bigvee \Delta) \end{aligned}$$

  Therefore, by the proof of allRight, $\vDash_{2\downarrow 1} (\bigwedge \Gamma \wedge (\phi)[x/c]) \rightarrow \bigvee \Delta$ implies $\vDash_{2\downarrow 1} (\bigwedge \Gamma \wedge (\exists x.\phi)) \rightarrow \bigvee \Delta$.

- exRight: Let $(\mathcal{D}, s_1, s_2)$ be a two-state structure and let β be a variable assignment. By Lemma 50 it is sufficient to prove that

$$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow (\exists x.\phi \vee [x/t](\phi) \vee \bigvee \Delta) \tag{7.44}$$

  implies

$$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow (\exists x.\phi \vee \bigvee \Delta). \tag{7.45}$$

  Therefore, assume (7.44). Then,

$$\begin{aligned} \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \quad \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \exists x.\phi \\ \text{or } \quad \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} [x/t](\phi) \text{ or } \quad \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta. \end{aligned}$$

  If either $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \exists x.\phi$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta$ holds, then (7.45) holds and we are done. Therefore, assume the contrary. We show that in this case $\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} [x/t](\phi)$ holds. By assumption, $\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \exists x.\phi$ holds which equals $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\exists x.\phi$ and hence $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \forall x.\neg\phi$. The latter implies in particular $\mathcal{D}, s_1, s_2, \beta' \vDash_{2\downarrow 1} \neg\phi$ for β' defined as $\beta'(y) = ([x/t](y))^{\mathcal{D}, s_i, \beta}$ for all variables y. Therefore, $\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} [x/t](\phi)$ holds as desired.

- allLeft:

  The soundness of allLeft can be reduced to the soundness of exRight. The premiss can be rearranged as follows:

$$\begin{aligned}
&\big(\bigwedge \Gamma \wedge (\forall x.\phi) \wedge [x/t](\phi)\big) \to \bigvee \Delta \\
\equiv_{2\downarrow 1}\ &\neg\big(\bigwedge \Gamma\big) \vee \neg(\forall x.\phi) \vee \neg[x/t](\phi) \vee \bigvee \Delta \\
\equiv_{2\downarrow 1}\ &\neg\big(\bigwedge \Gamma\big) \vee (\exists x.\neg\phi) \vee [x/t](\neg\phi) \vee \bigvee \Delta \\
\equiv_{2\downarrow 1}\ &\bigwedge \Gamma \to \big((\exists x.\neg\phi) \vee [x/t](\neg\phi) \vee \bigvee \Delta\big)
\end{aligned}$$

  Similarly, the conclusion can be rearranged:

$$\begin{aligned}
&\big(\bigwedge \Gamma \wedge (\forall x.\phi)\big) \to \bigvee \Delta && \equiv_{2\downarrow 1}\ \neg\big(\bigwedge \Gamma\big) \vee \neg(\forall x.\phi) \vee \bigvee \Delta \\
\equiv_{2\downarrow 1}\ &\neg\big(\bigwedge \Gamma\big) \vee (\exists x.\neg\phi) \vee \bigvee \Delta && \equiv_{2\downarrow 1}\ \bigwedge \Gamma \to \big((\exists x.\neg\phi) \vee \bigvee \Delta\big)
\end{aligned}$$

  Thus, by the proof of exRight, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \big(\bigwedge \Gamma \wedge (\forall x.\phi) \wedge [x/t](\phi)\big) \to \bigvee \Delta$ implies $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \big(\bigwedge \Gamma \wedge (\forall x.\phi)\big) \to \bigvee \Delta$ for all two-state structures $(\mathcal{D}, s_1, s_2)$ and variable assignments β. The latter implies by Lemma 50 that $\vDash_2 \big(\bigwedge \Gamma \wedge (\forall x.\phi) \wedge [x/t](\phi)\big) \to \bigvee \Delta$ implies $\vDash_2 \big(\bigwedge \Gamma \wedge (\forall x.\phi)\big) \to \bigvee \Delta$.

□

Lemma 54. *The rules from Figure 7.3 are sound with respect to two-state semantics.*

Proof. The soundness of the rules from Figure 7.3 follows by Lemma 49 and the fact that those rules are sound in one-state semantics (see for instance [Ahrendt et al., Chapter 2]). □

Lemma 55. *The rules from Figure 7.4 are sound with respect to two-state semantics.*

Proof. Let $(\mathcal{D}, s_1, s_2)$ be a two-state structure and let β be a variable assignment.

- $\ltimes$andRight: We show that any two-state model of the meaning formulas of the premisses is a two-state model for the conclusion, too. Let $(\mathcal{D}, s_1, s_2)$ be a two-state structure and let β be a variable assignment. We have to show that $\mathcal{D}, s_1, s_2, \beta \vDash_2 \bigwedge \Gamma \to \big(\ltimes\phi \vee \ltimes\psi \vee \bigvee \Delta\big)$ in combination with $\mathcal{D}, s_1, s_2, \beta \vDash_2 \bigwedge \Gamma \to (\phi \vee \bigvee \Delta)$ and $\mathcal{D}, s_1, s_2, \beta \vDash_2 \bigwedge \Gamma \to (\psi \vee \bigvee \Delta)$ implies $\mathcal{D}, s_1, s_2, \beta \vDash_2 \bigwedge \Gamma \to \big((\phi \wedge \psi) \vee \bigvee \Delta\big)$.

By Definition 33 (Two-State Evaluation) $\mathcal{D}, s_1, s_2, \beta \vDash_2 \bigwedge \Gamma \rightarrow \left(\ltimes \phi \vee \ltimes \psi \vee \bigvee \Delta\right)$ holds if and only if

$$\begin{aligned} \Big(\quad & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow \left(\ltimes \phi \vee \ltimes \psi \vee \bigvee \Delta\right) \\ & \text{or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow \left(\ltimes \phi \vee \ltimes \psi \vee \bigvee \Delta\right)\Big) \\ \text{and } & \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow \left(\ltimes \phi \vee \ltimes \psi \vee \bigvee \Delta\right). \end{aligned} \tag{7.46}$$

Because $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \ltimes \phi$ is always true, Equation (7.46) holds if, and only if,

$$\begin{aligned} & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow \left(\ltimes \phi \vee \ltimes \psi \vee \bigvee \Delta\right) \\ \text{or } & \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow \left(\ltimes \phi \vee \ltimes \psi \vee \bigvee \Delta\right). \end{aligned}$$

This again is equivalent to

$$\begin{aligned} & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \\ \text{or } & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigvee \Delta \\ \text{or } & (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ iff } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi) \\ \text{or } & (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi \text{ iff } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi) \end{aligned} \tag{7.47}$$

by Definition 31 (Restricted Two-State Evaluation).

Additionally, $\mathcal{D}, s_1, s_2, \beta \vDash_2 \bigwedge \Gamma \rightarrow (\phi \vee \bigvee \Delta)$ holds if and only if

$$\begin{aligned} \Big(\quad & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow (\phi \vee \bigvee \Delta) \\ & \text{or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow (\phi \vee \bigvee \Delta)\Big) \\ \text{and } & \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow (\phi \vee \bigvee \Delta) \end{aligned}$$

which is equivalent to

$$\begin{aligned} (\quad & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \\ & \text{or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigvee \Delta \\ & \text{or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi) \\ \text{and } (\quad & \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \bigvee \Delta \\ & \text{or } \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \phi). \end{aligned} \tag{7.48}$$

Similarly, $\mathcal{D}, s_1, s_2, \beta \models_2 \bigwedge \Gamma \to (\psi \vee \bigvee \Delta)$ holds if and only if

$$\begin{aligned}
&(\quad \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_2, s_1, \beta \models_{2\downarrow 1} \neg \bigwedge \Gamma \\
&\text{or } \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \bigvee \Delta \text{ or } \mathcal{D}, s_2, s_1, \beta \models_{2\downarrow 1} \bigvee \Delta \\
&\text{or } \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \psi \text{ or } \mathcal{D}, s_2, s_1, \beta \models_{2\downarrow 1} \psi) \qquad (7.49) \\
\text{and } &(\quad \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \bigvee \Delta \\
&\text{or } \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \psi) .
\end{aligned}$$

We have to show that (7.47), (7.48) and (7.49) imply $\mathcal{D}, s_1, s_2, \beta \models_2 \bigwedge \Gamma \to \big((\phi \wedge \psi) \vee \bigvee \Delta\big)$.

$\mathcal{D}, s_1, s_2, \beta \models_2 \bigwedge \Gamma \to \big((\phi \wedge \psi) \vee \bigvee \Delta\big)$ holds if and only if

$$\begin{aligned}
&\Big(\quad \mathcal{D}, s_1, s_2, \beta \models_{2\downarrow 1} \bigwedge \Gamma \to \big((\phi \wedge \psi) \vee \bigvee \Delta\big) \\
&\text{or } \mathcal{D}, s_2, s_1, \beta \models_{2\downarrow 1} \bigwedge \Gamma \to \big((\phi \wedge \psi) \vee \bigvee \Delta\big)\Big) \qquad (7.50) \\
\text{and } &\mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \bigwedge \Gamma \to \big((\phi \wedge \psi) \vee \bigvee \Delta\big) . \qquad (7.51)
\end{aligned}$$

We first show that (7.48) and (7.49) imply (7.51). By (7.48) and (7.49) we get

$$\begin{aligned}
&(\quad \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \bigvee \Delta \\
&\text{or } \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \phi) \\
\text{and } &(\quad \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \bigvee \Delta \\
&\text{or } \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \psi)
\end{aligned}$$

which is equivalent to

$$\begin{aligned}
&\mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \bigvee \Delta \\
&\text{or } (\mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \phi \text{ and } \mathcal{D}, s_1, s_1, \beta \models_{2\downarrow 1} \psi)
\end{aligned}$$

and hence to (7.51).

Further, we show that (7.47), (7.48) and (7.49) imply (7.50). Equations

(7.48) and (7.49) imply that

$$\begin{aligned}
&(\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \\
&\text{or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigvee \Delta \\
&\text{or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi) \\
&\text{and } (\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \\
&\text{or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigvee \Delta \\
&\text{or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi)
\end{aligned} \tag{7.52}$$

holds which is equivalent to

$$\begin{aligned}
&\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \\
&\text{or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigvee \Delta \\
&\text{or } \big(\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi) \\
&\quad \text{and } (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi) \big) .
\end{aligned} \tag{7.53}$$

If

$$\begin{aligned}
&\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \\
&\text{or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigvee \Delta
\end{aligned}$$

holds, then also (7.50) holds and we are done. Therefore, assume the contrary. Then we have by (7.53)

$$\begin{aligned}
&(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi) \\
&\text{and } (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi)
\end{aligned} \tag{7.54}$$

and by (7.47)

$$\begin{aligned}
&(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ iff } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi) \\
&\text{or } (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi \text{ iff } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi) .
\end{aligned} \tag{7.55}$$

Without loss of generality let $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi$ iff $\mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi$ be true. Then (7.54) is equivalent to

$$\begin{aligned}
&\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \\
&\text{and } (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi)
\end{aligned} \tag{7.56}$$

which again is equivalent to

$$\begin{aligned}
&(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi) \\
&\text{or } (\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ and } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi) .
\end{aligned} \tag{7.57}$$

Again because of $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi$ iff $\mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi$ equation (7.57) holds if and only if

$$\begin{array}{c}(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi) \\ \text{or } (\mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi \text{ and } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi)\end{array} \tag{7.58}$$

which implies (7.50) by Definition 31 (Restricted Two-State Evaluation).

- $\ltimes$orLeft: The soundness of the rule $\ltimes$orLeft can be reduced to the soundness of the rule $\ltimes$andRight. To this end, by Lemma 50, it is sufficient to show that the meaning formulas of the premisses and conclusion of $\ltimes$orLeft can be rearranged in restricted two-state evaluation such that they fit the form of the premisses and conclusion of $\ltimes$andRight:
  - The first premiss of $\ltimes$orLeft is identical to the one of $\ltimes$andRight.
  - For the second premiss $\bigwedge \Gamma \wedge \phi \rightarrow \bigvee \Delta \equiv_{2\downarrow 1} \bigwedge \Gamma \rightarrow \big(\neg\phi \vee \bigvee \Delta\big)$ holds.
  - For the third premiss $\bigwedge \Gamma \wedge \psi \rightarrow \bigvee \Delta \equiv_{2\downarrow 1} \bigwedge \Gamma \rightarrow \big(\neg\psi \vee \bigvee \Delta\big)$ holds.
  - For the conclusion $\bigwedge \Gamma \wedge (\phi \vee \psi) \rightarrow \bigvee \Delta \equiv_{2\downarrow 1} \bigwedge \Gamma \rightarrow \big((\neg\phi \wedge \neg\psi) \vee \bigvee \Delta\big)$ holds.
- $\ltimes$impLeft: As before, the soundness of the rule $\ltimes$impLeft can be reduced to the soundness of the rule $\ltimes$andRight. To this end, by Lemma 50, it is sufficient to show that the meaning formulas of the premisses and conclusion of $\ltimes$impLeft can be rearranged in restricted two-state evaluation such that they fit the form of the premisses and conclusion of $\ltimes$andRight:
  - The first and the second premiss of $\ltimes$impLeft are identical to the ones of $\ltimes$andRight.
  - For the third premiss $\bigwedge \Gamma \wedge \psi \rightarrow \bigvee \Delta \equiv_{2\downarrow 1} \bigwedge \Gamma \rightarrow \big(\neg\psi \vee \bigvee \Delta\big)$ holds.
  - For the conclusion $\bigwedge \Gamma \wedge (\phi \rightarrow \psi) \rightarrow \bigvee \Delta \equiv_{2\downarrow 1} \bigwedge \Gamma \rightarrow \big((\phi \wedge \neg\psi) \vee \bigvee \Delta\big)$ holds.

□

Lemma 56. *The rules from Figure 7.5 are sound with respect to two-state semantics.*

Proof. The rule schemata emptyModality, assignLocal, assignField, assignArray and unwindLoop fulfill the requirements of Theorem 52. Therefore, by Theorem 52, these rules are sound in two-state semantics, too. □

Lemma 57. *The rules from Figure 7.6 are sound with respect to two-state semantics.*

Proof.

- ⋉expandMethod: The rule

$$\text{expandMethod}\ \frac{\begin{array}{c}\Gamma \Longrightarrow \{u\}[\pi\ \texttt{method-frame(result=r, this=o)}: \\ \qquad \{\ \ body(\texttt{m}, A)\ \ \}\ \omega]\phi, \Delta \\ \Gamma \Longrightarrow \{u\} exactInstance_A(o), \Delta\end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \texttt{r = o.m();}\ \omega]\phi, \Delta}$$

  from Weiß [2011] equals ⋉expandMethod in the sense that any one-state model for the meaning formulas of the premisses of expandMethod is a model for the meaning formula of the premiss of ⋉expandMethod and vice versa. Therefore, the premiss of ⋉expand-Method logically implies its conclusion in one-state semantics. Hence, by Theorem 52, ⋉expandMethod is sound in two-state semantics, too.

- ⋉conditional: By Lemma 50, it is sufficient to show that the conjunction of the meaning formulas of the premisses imply the meaning formula of the conclusion in restricted two-state evaluation. Let $(\mathcal{D}, s_1, s_2)$ be an arbitrary two-state structure and let β be a variable assignment. If $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg \bigwedge \Gamma$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \bigvee \Delta$ holds, then we are done. Hence, assume the contrary. In this case

$$\begin{array}{l}\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \{u\} \ltimes (g), \\ \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} (\{u\} g \doteq \texttt{true}) \rightarrow \{u\}[\pi\ p_1;\ \omega]\phi \text{ and} \\ \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} (\{u\} g \doteq \texttt{false}) \rightarrow \{u\}[\pi\ p_2;\ \omega]\phi\end{array} \tag{7.59}$$

  need to hold. Equation (7.59) is equivalent to

$$\begin{array}{l}\mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2|1} \ltimes (g), \\ \mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2|1} (g \doteq \texttt{true}) \rightarrow [\pi\ p_1;\ \omega]\phi \text{ and} \\ \mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2|1} (g \doteq \texttt{false}) \rightarrow [\pi\ p_2;\ \omega]\phi\end{array} \tag{7.60}$$

  where s_1^u and s_2^u result from s_1 and s_2 by application of $\{u\}$, respectively. $\mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2|1} \ltimes (g)$ implies $g^{\mathcal{D}, s_1^u, \beta} = g^{\mathcal{D}, s_2^u, \beta}$.

  Case distinction.

  Case $g^{\mathcal{D}, s_1^u, \beta} = g^{\mathcal{D}, s_2^u, \beta} = true$. In this case $\texttt{if(}g\texttt{)}\ p_1\ \texttt{else}\ p_2;\ \omega$ started in s_i^u terminates in state $s_i^{u,\alpha}$ if and only if $p_1;\ \omega$ started in s_i^u terminates in $s_i^{u,\alpha}$. The latter follows by (7.60), line two. Therefore, we have $\mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2|1} [\pi\ \texttt{if(}g\texttt{)}\ p_1\ \texttt{else}\ p_2;\ \omega]\phi$ and hence $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \{u\}[\pi\ \texttt{if(}g\texttt{)}\ p_1\ \texttt{else}\ p_2;\ \omega]\phi$.

Case $g^{\mathcal{D},s_1^u,\beta} = g^{\mathcal{D},s_2^u,\beta} = \mathit{false}$. In this case $\texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2;\ \omega$ started in s_i^u terminates in state $s_i^{u,\alpha}$ if and only if $p_2;\ \omega$ started in s_i^u terminates in $s_i^{u,\alpha}$. The latter follows by (7.60), line three. Therefore, we have $\mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2|1} [\pi\ \texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2;\ \omega]\phi$ and hence $\mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \{u\}[\pi\ \texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2;\ \omega]\phi$.

- $\ltimes$extConditional: We may assume

$$\begin{aligned} &\vDash_2 \bigwedge \Gamma \to \left(\{u\}[\pi\ \texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2](\mathit{Inv} \wedge \ltimes(\mathit{Inv})) \vee \bigvee \Delta\right) \\ &\text{and } \vDash_2 \mathit{Inv} \to [\pi\ \omega]\phi \end{aligned} \tag{7.61}$$

and have to show

$$\vDash_2 \bigwedge \Gamma \to \left(\{u\}[\pi\ \texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2;\ \omega]\phi \vee \bigvee \Delta\right). \tag{7.62}$$

By Definition 33, equation (7.62) holds if and only if

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2|1} \bigwedge \Gamma \to (\{u\}[\pi\ \texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2;\ \omega]\phi \vee \bigvee \Delta) \text{ and} \\ &\Big(\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \bigwedge \Gamma \to (\{u\}[\pi\ \texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2;\ \omega]\phi \vee \bigvee \Delta) \\ &\quad \text{or } \mathcal{D}, s_2, s_1, \beta \vDash_{2|1} \bigwedge \Gamma \to (\{u\}[\pi\ \texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2;\ \omega]\phi \vee \bigvee \Delta)\Big) \end{aligned} \tag{7.63}$$

for all two-state structures $(\mathcal{D}, s_1, s_2)$ and all variable assignments β. Let $(\mathcal{D}, s_1, s_2)$ be an arbitrary two-state structure and let β be a variable assignment. If

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2|1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_1, \beta \vDash_{2|1} \bigvee \Delta \\ \text{and } \Big(&\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \bigvee \Delta \\ &\text{or } \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \bigvee \Delta\Big) \end{aligned}$$

hold, then we are done. Hence, assume the contrary.

Again by Definition 33, line one of equation (7.61) holds if and only if

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2|1} \\ &\quad \bigwedge \Gamma \to (\{u\}[\pi\ \texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2](\mathit{Inv} \wedge \ltimes(\mathit{Inv}))) \vee \bigvee \Delta) \\ \text{and } \Big(&\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2|1} \\ &\qquad \bigwedge \Gamma \to (\{u\}[\pi\ \texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2](\mathit{Inv} \wedge \ltimes(\mathit{Inv}))) \vee \bigvee \Delta) \\ &\text{or } \mathcal{D}, s_2, s_1, \beta \vDash_{2|1} \\ &\qquad \bigwedge \Gamma \to (\{u\}[\pi\ \texttt{if}\,(g)\ p_1\,\texttt{else}\ p_2](\mathit{Inv} \wedge \ltimes(\mathit{Inv}))) \vee \bigvee \Delta)\Big) \end{aligned} \tag{7.64}$$

for all two-state structures $(\mathcal{D}, s_1, s_2)$ and variable assignments β. Thus, we may assume

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \{u\}[\pi \texttt{ if(}g\texttt{) } p_1 \texttt{ else } p_2](Inv \wedge \ltimes(Inv)) \\ \text{and } \big(&\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}[\pi \texttt{ if(}g\texttt{) } p_1 \texttt{ else } p_2](Inv \wedge \ltimes(Inv)) \\ &\text{or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \{u\}[\pi \texttt{ if(}g\texttt{) } p_1 \texttt{ else } p_2](Inv \wedge \ltimes(Inv))\big) \end{aligned} \tag{7.65}$$

which is equivalent to

$$\begin{aligned} &\mathcal{D}, s_1^\alpha, s_1^\alpha, \beta \vDash_{2\downarrow 1} Inv \\ \text{and } \big(&\quad \big(\quad \mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} Inv \text{ and } g^{\mathcal{D}, s_1^\alpha, \beta} = g^{\mathcal{D}, s_2^\alpha, \beta} \\ &\qquad \text{and } (\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} Inv \text{ iff } \mathcal{D}, s_2^\alpha, s_1^\alpha, \beta \vDash_{2\downarrow 1} Inv)\big) \\ &\text{or } \big(\quad \mathcal{D}, s_2^\alpha, s_1^\alpha, \beta \vDash_{2\downarrow 1} Inv \text{ and } g^{\mathcal{D}, s_1^\alpha, \beta} = g^{\mathcal{D}, s_2^\alpha, \beta} \\ &\qquad \text{and } (\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} Inv \text{ iff } \mathcal{D}, s_2^\alpha, s_1^\alpha, \beta \vDash_{2\downarrow 1} Inv))\big) \end{aligned} \tag{7.66}$$

for all s_1^α, s_2^α such that $\texttt{if(}g\texttt{) } p_1 \texttt{ else } p_2$ started in s_1^u terminates in s_1^α and $\texttt{if(}g\texttt{) } p_1 \texttt{ else } p_2$ started in s_2^u terminates in s_2^α and where s_1^u and s_2^u result from s_1 and s_2 by application of $\{u\}$, respectively. Equation (7.66) simplifies to

$$\begin{aligned} &\mathcal{D}, s_1^\alpha, s_1^\alpha, \beta \vDash_{2\downarrow 1} Inv \text{ and } g^{\mathcal{D}, s_1^\alpha, \beta} = g^{\mathcal{D}, s_2^\alpha, \beta} \\ \text{and } &\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2\downarrow 1} Inv \text{ and } \mathcal{D}, s_2^\alpha, s_1^\alpha, \beta \vDash_{2\downarrow 1} Inv\ . \end{aligned} \tag{7.67}$$

By $\vDash_2 Inv \rightarrow [\pi\ \omega]\phi$ we get

$$\begin{aligned} &\mathcal{D}, s_3, s_3, \beta \vDash_{2\downarrow 1} Inv \rightarrow [\pi\ \omega]\phi \\ \text{and } \big(&\quad \mathcal{D}, s_3, s_4, \beta \vDash_{2\downarrow 1} Inv \rightarrow [\pi\ \omega]\phi \\ &\text{or } \mathcal{D}, s_4, s_3, \beta \vDash_{2\downarrow 1} Inv \rightarrow [\pi\ \omega]\phi\big) \end{aligned} \tag{7.68}$$

for all two-state structures $(\mathcal{D}, s_3, s_4)$ and all variable assignments β. Remember that $\mathcal{D}, s_3, s_4, \beta \vDash_{2\downarrow 1} \psi_1 \rightarrow \psi_2$ or $\mathcal{D}, s_4, s_3, \beta \vDash_{2\downarrow 1} \psi_1 \rightarrow \psi_2$ holds if and only if $\mathcal{D}, s_3, s_4, \beta \vDash_{2\downarrow 1} \psi_1$ and $\mathcal{D}, s_4, s_3, \beta \vDash_{2\downarrow 1} \psi_1$ imply that either $\mathcal{D}, s_3, s_4, \beta \vDash_{2\downarrow 1} \psi_2$ or $\mathcal{D}, s_4, s_3, \beta \vDash_{2\downarrow 1} \psi_2$ hold. Therefore, for all states s_3, s_4 such that $\mathcal{D}, s_3, s_3, \beta \vDash_{2\downarrow 1} Inv$, $\mathcal{D}, s_3, s_4, \beta \vDash_{2\downarrow 1} Inv$ and $\mathcal{D}, s_4, s_3, \beta \vDash_{2\downarrow 1} Inv$ hold, the program ω started in s_3, s_4 either does not terminate or it terminates in states s_3^α, s_4^α, respectively, such that $\mathcal{D}, s_3^\alpha, s_3^\alpha, \beta \vDash_{2\downarrow 1} \phi$ and either $\mathcal{D}, s_3^\alpha, s_4^\alpha, \beta \vDash_{2\downarrow 1} \phi$ or $\mathcal{D}, s_4^\alpha, s_3^\alpha, \beta \vDash_{2\downarrow 1} \phi$ hold. Thus, ω started in s_1^α, s_2^α either does not terminate or it terminates in states s_3^α, s_4^α, respectively, such that $\mathcal{D}, s_3^\alpha, s_3^\alpha, \beta \vDash_{2\downarrow 1} \phi$ and either $\mathcal{D}, s_3^\alpha, s_4^\alpha, \beta \vDash_{2\downarrow 1} \phi$ or $\mathcal{D}, s_4^\alpha, s_3^\alpha, \beta \vDash_{2\downarrow 1} \phi$ hold. Hence, (7.63) is true.

- $\ltimes$loopInvariant: We may assume

$$\begin{aligned}&\vDash_2 \bigwedge \Gamma \to (\{u\}(Inv \wedge \ltimes(Inv) \wedge \ltimes(g)) \vee \bigvee \Delta),\\ &\vDash_2 (Inv, g \doteq \texttt{true}) \to [p](Inv \wedge \ltimes(Inv) \wedge \ltimes(g)) \text{ and}\\ &\vDash_2 (Inv, g \doteq \texttt{false}) \to [\pi\ \omega]\phi\end{aligned} \tag{7.69}$$

and have to show

$$\vDash_2 \bigwedge \Gamma \to (\{u\}[\pi\ \texttt{while}(g)\ p;\ \omega]\phi \vee \bigvee \Delta)\ . \tag{7.70}$$

By Definition 33, equation (7.70) holds if and only if

$$\begin{aligned}&\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to (\{u\}[\pi\ \texttt{while}(g)\ p;\ \omega]\phi \vee \bigvee \Delta) \text{ and}\\ &\big(\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to (\{u\}[\pi\ \texttt{while}(g)\ p;\ \omega]\phi \vee \bigvee \Delta)\\ &\quad \text{or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to (\{u\}[\pi\ \texttt{while}(g)\ p;\ \omega]\phi \vee \bigvee \Delta)\big)\end{aligned} \tag{7.71}$$

for all two-state structures $(\mathcal{D}, s_1, s_2)$ and all variable assignments β. Let $(\mathcal{D}, s_1, s_2)$ be an arbitrary two-state structure and let β be a variable assignment. If

$$\begin{aligned}&\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \bigvee \Delta\\ \text{and } \big(\quad &\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta\\ &\text{or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma \text{ or } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta\big)\end{aligned}$$

hold, then we are done. Hence, assume the contrary.

Again by Definition 33, line one of equation (7.69) holds if and only if

$$\begin{aligned}&\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to (\{u\}(Inv \wedge \ltimes(Inv) \wedge \ltimes(g)) \vee \bigvee \Delta) \text{ and}\\ &\big(\quad \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to (\{u\}(Inv \wedge \ltimes(Inv) \wedge \ltimes(g)) \vee \bigvee \Delta)\\ &\quad \text{or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \bigwedge \Gamma \to (\{u\}(Inv \wedge \ltimes(Inv) \wedge \ltimes(g)) \vee \bigvee \Delta)\big)\end{aligned} \tag{7.72}$$

holds for all two-state structures $(\mathcal{D}, s_1, s_2)$ and all variable assignments β. Therefore, we may assume

$$\begin{aligned}&\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \{u\}(Inv \wedge \ltimes(Inv) \wedge \ltimes(g))\\ \text{and } \big(\quad &\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}(Inv \wedge \ltimes(Inv) \wedge \ltimes(g))\\ &\text{or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \{u\}(Inv \wedge \ltimes(Inv) \wedge \ltimes(g))\big)\end{aligned} \tag{7.73}$$

which is equivalent to

$$\begin{aligned}
&\mathcal{D}, s_1^u, s_1^u, \beta \vDash_{2|1} \mathit{Inv} \\
\text{and } \big(\quad (\quad &\mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2|1} \mathit{Inv} \text{ and } g^{\mathcal{D},s_1^u,\beta} = g^{\mathcal{D},s_2^u,\beta} \\
&\text{and } (\mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2|1} \mathit{Inv} \text{ iff } \mathcal{D}, s_2^u, s_1^u, \beta \vDash_{2|1} \mathit{Inv})) \\
\text{or } (\quad &\mathcal{D}, s_2^u, s_1^u, \beta \vDash_{2|1} \mathit{Inv} \text{ and } g^{\mathcal{D},s_1^u,\beta} = g^{\mathcal{D},s_2^u,\beta} \\
&\text{and } (\mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2|1} \mathit{Inv} \text{ iff } \mathcal{D}, s_2^u, s_1^u, \beta \vDash_{2|1} \mathit{Inv}))\big)
\end{aligned} \tag{7.74}$$

where s_1^u and s_2^u result from s_1 and s_2 by application of $\{u\}$, respectively. Equation (7.74) simplifies to

$$\begin{aligned}
&\mathcal{D}, s_1^u, s_1^u, \beta \vDash_{2|1} \mathit{Inv} \text{ and } g^{\mathcal{D},s_1^u,\beta} = g^{\mathcal{D},s_2^u,\beta} \\
\text{and } &\mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2|1} \mathit{Inv} \text{ and } \mathcal{D}, s_2^u, s_1^u, \beta \vDash_{2|1} \mathit{Inv} \ .
\end{aligned} \tag{7.75}$$

By $\vDash_2 (\mathit{Inv}, g \doteq \texttt{true}) \rightarrow [p](\mathit{Inv} \wedge \ltimes(\mathit{Inv}) \wedge \ltimes(g))$ we get

$$\begin{aligned}
&\mathcal{D}, s_3, s_3, \beta \vDash_{2|1} (\mathit{Inv}, g \doteq \texttt{true}) \rightarrow [p](\mathit{Inv} \wedge \ltimes(\mathit{Inv}) \wedge \ltimes(g)) \text{ and} \\
\big(\quad &\mathcal{D}, s_3, s_4, \beta \vDash_{2|1} (\mathit{Inv}, g \doteq \texttt{true}) \rightarrow [p](\mathit{Inv} \wedge \ltimes(\mathit{Inv}) \wedge \ltimes(g)) \\
\text{or } &\mathcal{D}, s_4, s_3, \beta \vDash_{2|1} (\mathit{Inv}, g \doteq \texttt{true}) \rightarrow [p](\mathit{Inv} \wedge \ltimes(\mathit{Inv}) \wedge \ltimes(g))\big)
\end{aligned} \tag{7.76}$$

for all two-state structures $(\mathcal{D}, s_3, s_4)$ and all variable assignments β. Remember that $\mathcal{D}, s_3, s_4, \beta \vDash_{2|1} \psi_1 \rightarrow \psi_2$ or $\mathcal{D}, s_4, s_3, \beta \vDash_{2|1} \psi_1 \rightarrow \psi_2$ holds if and only if $\mathcal{D}, s_3, s_4, \beta \vDash_{2|1} \psi_1$ and $\mathcal{D}, s_4, s_3, \beta \vDash_{2|1} \psi_1$ imply that either $\mathcal{D}, s_3, s_4, \beta \vDash_{2|1} \psi_2$ or $\mathcal{D}, s_4, s_3, \beta \vDash_{2|1} \psi_2$ hold. Therefore, for all states s_3, s_4 such that $\mathcal{D}, s_3, s_3, \beta \vDash_{2|1} \mathit{Inv}$, $\mathcal{D}, s_3, s_4, \beta \vDash_{2|1} \mathit{Inv}$, $\mathcal{D}, s_4, s_3, \beta \vDash_{2|1} \mathit{Inv}$ and $g^{\mathcal{D},s_3,\beta} = g^{\mathcal{D},s_4,\beta} = \mathit{true}$ hold, the program p started in s_3, s_4 either does not terminate or it terminates in states s_3^α, s_4^α, respectively, such that again $\mathcal{D}, s_3^\alpha, s_3^\alpha, \beta \vDash_{2|1} \mathit{Inv}$, $\mathcal{D}, s_3^\alpha, s_4^\alpha, \beta \vDash_{2|1} \mathit{Inv}$, $\mathcal{D}, s_4^\alpha, s_3^\alpha, \beta \vDash_{2|1} \mathit{Inv}$ and $g^{\mathcal{D},s_3^\alpha,\beta} = g^{\mathcal{D},s_4^\alpha,\beta}$ hold. Thus, $\texttt{while}(g)\ p$ started in s_1^u, s_2^u either does not terminate or it terminates in states s_1^α, s_2^α, respectively, such that again $\mathcal{D}, s_1^\alpha, s_1^\alpha, \beta \vDash_{2|1} \mathit{Inv}$, $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2|1} \mathit{Inv}$, $\mathcal{D}, s_2^\alpha, s_1^\alpha, \beta \vDash_{2|1} \mathit{Inv}$ and $g^{\mathcal{D},s_1^\alpha,\beta} = g^{\mathcal{D},s_2^\alpha,\beta}$ hold.

Finally, in case the loop terminates in states s_1^α, s_2^α, respectively, such that $\mathcal{D}, s_1^\alpha, s_1^\alpha, \beta \vDash_{2|1} \mathit{Inv}$, $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2|1} \mathit{Inv}$, $\mathcal{D}, s_2^\alpha, s_1^\alpha, \beta \vDash_{2|1} \mathit{Inv}$ and $g^{\mathcal{D},s_1^\alpha,\beta} = g^{\mathcal{D},s_2^\alpha,\beta}$ hold, then $g^{\mathcal{D},s_1^\alpha,\beta} = g^{\mathcal{D},s_2^\alpha,\beta} = \mathit{false}$ needs to hold. In this case $\vDash_2 (\mathit{Inv}, g \doteq \texttt{false}) \rightarrow [\pi\ \omega]\phi$ implies that $\mathcal{D}, s_1^\alpha, s_1^\alpha, \beta \vDash_{2|1} [\pi\ \omega]\phi$ and either $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2|1} [\pi\ \omega]\phi$ or $\mathcal{D}, s_2^\alpha, s_1^\alpha, \beta \vDash_{2|1} [\pi\ \omega]\phi$ holds.

Altogether we get that the program $\texttt{while}(g)\ p;\ \omega$ started in s_1^u, s_2^u either does not terminate or it terminates in states s_1^α, s_2^α, respectively, such that $\mathcal{D}, s_1^\alpha, s_1^\alpha, \beta \vDash_{2|1} \phi$ and either $\mathcal{D}, s_1^\alpha, s_2^\alpha, \beta \vDash_{2|1} \phi$ or $\mathcal{D}, s_2^\alpha, s_1^\alpha, \beta \vDash_{2|1} \phi$ holds. Hence, (7.71) is true.

- $\ltimes$extLoopInvariant: analog to $\ltimes$extConditional.
- $\ltimes$createObj: Weiß [2011] argues why it is sufficient to require that o' is different from `null`, its dynamic type is A and the object is not yet created if the heap is wellformed. As already mentioned, the only difference between the rule $\ltimes$createObj and the original rule createObj from Weiß [2011] is that o' is a fresh program variable instead of a fresh function symbol. In one-state semantics this makes no difference, but in two-state semantics there is a difference: a fresh function symbol would be interpreted by $\mathcal{D}$ and thus in the same way for the two runs. There is, however, no reason why the two runs should allocate the same new object. A fresh program variable, on the other hand, is interpreted by the two states potentially differently.

□

Lemma 58. *The rules from Figure 7.7 are sound with respect to two-state semantics.*

Proof.

- $\ltimes$not: By Definition 33 (Two-State Evaluation), $\mathcal{D}, s_1, s_2, \beta \vDash_2 \ltimes(\phi)$ holds if and only if

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \ltimes(\phi) \\ \text{and } &(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \ltimes(\phi) \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \ltimes(\phi)) \end{aligned} \tag{7.77}$$

  hold. By Definition 31 (Restricted Two-State Evaluation), (7.77) is equivalent to

$$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ iff } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi \tag{7.78}$$

  and hence to

$$\mathcal{D}, s_1, s_2, \beta \nvDash_{2\downarrow 1} \phi \text{ iff } \mathcal{D}, s_2, s_1, \beta \nvDash_{2\downarrow 1} \phi\ . \tag{7.79}$$

  Again by Definition 31, equation (7.78) is equivalent to

$$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\phi \text{ iff } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \neg\phi \tag{7.80}$$

  which in turn is equals

$$\begin{aligned} &\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \ltimes(\neg\phi) \\ \text{and } &(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \ltimes(\neg\phi) \text{ or } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \ltimes(\neg\phi)) \end{aligned} \tag{7.81}$$

  and hence $\mathcal{D}, s_1, s_2, \beta \vDash_2 \ltimes(\neg\phi)$, as desired.

- $\ltimes$approxAnd: By Lemma 50 it is sufficient to prove that

$$\vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow ((\ltimes(\phi) \wedge \ltimes(\psi)) \vee \bigvee \Delta) \tag{7.82}$$

  implies

$$\vDash_{2\downarrow 1} \bigwedge \Gamma \rightarrow (\ltimes(\phi \wedge \psi) \vee \bigvee \Delta)\,. \tag{7.83}$$

  Therefore, assume (7.82). Then, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \ltimes(\phi) \wedge \ltimes(\psi)$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta$ for all two-state structures $(\mathcal{D}, s_1, s_2)$ and all variable assignments β.

  Let $(\mathcal{D}, s_1, s_2)$ be an arbitrary two-state structure and let β be a variable assignment. If $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg \bigwedge \Gamma$ or $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \bigvee \Delta$ holds, then (7.83) is true and we are done. Hence, assume the contrary. In this case $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \ltimes(\phi) \wedge \ltimes(\psi)$ needs to hold.

  By Definition 31 (Restricted Two-State Evaluation), $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \ltimes(\phi) \wedge \ltimes(\psi)$ holds if and only if

$$\begin{aligned}&(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ iff } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi)\\ \text{and } &(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi \text{ iff } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi)\end{aligned} \tag{7.84}$$

  hold. Equation (7.84) implies

$$\begin{aligned}&(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \text{ and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \psi)\\ \text{iff } &(\mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi \text{ and } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \psi)\end{aligned} \tag{7.85}$$

  which in turn equals

$$\begin{aligned}&\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \phi \wedge \psi\\ \text{iff } &\mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \phi \wedge \psi\ .\end{aligned} \tag{7.86}$$

  Finally, (7.86) holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \ltimes(\phi \wedge \psi)$.

- $\ltimes$approxOr: Analog to $\ltimes$approxAnd.
- $\ltimes$approxEq: Analog to $\ltimes$approxAnd.
- $\ltimes$approxPred: Analog to $\ltimes$approxAnd.
- $\ltimes$constant: By Lemma 50 it is sufficient to prove that $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \ltimes(c)$ holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \mathit{true}$ holds. Therefore, we show $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \ltimes(c)$. We have $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \ltimes(c)$ if and only if $(\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} c$ iff $\mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} c)$ which in turn equals ($c^{\mathcal{D}}$ iff $c^{\mathcal{D}}$). The latter is obviously true.
- $\ltimes$approxFunc: Analog to $\ltimes$approxAnd.

□

Lemma 59. *The rule* toOneState *(Section 7.3.5) is sound in two-state semantics.*

Proof. According to Definition 37 (Validity of Sequents in Two-State Semantics) the sequent $\gamma_1, \ldots, \gamma_n \Longrightarrow \delta_1, \ldots, \delta_m$ is universally valid in two-state semantics if and only if $\mathcal{D}, s_1, s_2 \vDash_2 \bigwedge_{1\leq i\leq n} \gamma_i \to \bigvee_{1\leq j\leq m} \delta_j$ for all two-state structures $(\mathcal{D}, s_1, s_2)$. By Definition 33 (Two-State Evaluation) and Lemma 39 the latter is equivalent to

$$\mathcal{D}, s_1 \vDash \bigwedge_{1\leq i\leq n} \gamma_i \to \bigvee_{1\leq j\leq m} \delta_j \tag{7.87}$$

$$\begin{aligned} \text{and} \qquad \mathcal{D}, s_1, s_2 &\vDash_{2\downarrow 1} \bigwedge_{1\leq i\leq n} \gamma_i \to \bigvee_{1\leq j\leq m} \delta_j \\ \text{or } \mathcal{D}, s_2, s_1 &\vDash_{2\downarrow 1} \bigwedge_{1\leq i\leq n} \gamma_i \to \bigvee_{1\leq j\leq m} \delta_j \end{aligned} \tag{7.88}$$

for all two-state structures $(\mathcal{D}, s_1, s_2)$.

(7.87) follows from the second premiss of the rule toOneState: Because in one-state semantics $\ltimes$ is interpreted as *true*, (7.87) holds if and only if

$$\mathcal{D}, s_1 \vDash \bigwedge_{1\leq i\leq n} \gamma_i^1 \to \bigvee_{1\leq j\leq m} \delta_j^1 .$$

By Lemma 49.1, $\bigwedge_{1\leq i\leq n} \gamma_i^1 \to \bigvee_{1\leq j\leq m} \delta_j^1$ is universally valid in one-state semantics if it is universally valid in two-state semantics, thus if the sequent $\gamma_1^1, \ldots, \gamma_n^1 \Longrightarrow \delta_1^1, \ldots, \delta_m^1$ is universally valid in two-state semantics.

(7.88) follows from the first premiss of the rule toOneState: By Lemma 49.1, the meaning formula

$$\forall h.\forall h_2.\forall \bar{x}.\forall \bar{x}_2.\Big(\bigwedge_{1\leq i\leq n} \gamma_i^{2\downarrow 1} \wedge \bigwedge_{1\leq i\leq n} \gamma_i^{2\downarrow 2}\Big) \to \Big(\bigvee_{1\leq j\leq m} \delta_j^{2\downarrow 1} \vee \bigvee_{1\leq j\leq m} \delta_j^{2\downarrow 2}\Big) \tag{7.89}$$

of the sequent

$$\Longrightarrow \forall h.\forall h_2.\forall \bar{x}.\forall \bar{x}_2.\Big(\bigwedge_{1\leq i\leq n} \gamma_i^{2\downarrow 1} \wedge \bigwedge_{1\leq i\leq n} \gamma_i^{2\downarrow 2}\Big) \to \Big(\bigvee_{1\leq j\leq m} \delta_j^{2\downarrow 1} \vee \bigvee_{1\leq j\leq m} \delta_j^{2\downarrow 2}\Big)$$

is universally valid in one-state semantics if it is universally valid in two-state semantics. Further, (7.89) is logically equivalent (in one-state semantics) to

$$\forall h.\forall h_2.\forall \bar{x}.\forall \bar{x}_2.\Big(\bigwedge_{1\leq i\leq n} \gamma_i^{2\downarrow 1} \to \bigvee_{1\leq j\leq m} \delta_j^{2\downarrow 1}\Big) \vee \Big(\bigwedge_{1\leq i\leq n} \gamma_i^{2\downarrow 2} \to \bigvee_{1\leq j\leq m} \delta_j^{2\downarrow 2}\Big) . \tag{7.90}$$

It remains to be shown that the universal validity of (7.90) in one-state semantics implies (7.88) for all two-state structures $(\mathcal{D}, s_1, s_2)$.

For any two-state structure $(\mathcal{D}, s_1, s_2)$ and any variable assignment β' there is a variable assignment β such that $h^\beta = \mathtt{heap}^{\mathcal{D},s_1}$, $\bar{x}^\beta = \bar{\mathtt{x}}^{\mathcal{D},s_1}$, $h_2^\beta = \mathtt{heap}^{\mathcal{D},s_2}$ and $\bar{x}_2^\beta = \bar{\mathtt{x}}^{\mathcal{D},s_2}$ and $v^\beta = v^{\beta'}$ else. Let δ_j^i denote the formula resulting from execution of step i of the construction of $\delta_j^{2\downarrow 1}$. We show for each step of the construction that $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \delta_j^{i-1}$ holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \delta_j^i$ holds. Because $\delta_j^{2\downarrow 1}$ does not contain the $\ltimes$predicate any more and each modality is prefixed by an update of the form $\{\mathtt{heap} := h \parallel \bar{\mathtt{x}} := \bar{x}\}$, where h and $\bar{x}$ are variables, we can conclude by Lemma 49.4 that $\mathcal{D}, s, \beta \vDash \delta_j^{2\downarrow 1}$ holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \delta_j$.

1. By Definition 31 and Lemma 8, $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{\mathtt{heap} := h \parallel \bar{\mathtt{x}} := \bar{x}\}\{\mathtt{heap} := h_2 \parallel \bar{\mathtt{x}} := \bar{x}_2\}_2\delta_j$ if and only if $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \{\mathtt{heap} := h_2 \parallel \bar{\mathtt{x}} := \bar{x}_2\}_2\delta_j$. Another appeal to Definition 31 and Lemma 8 yields that $\mathcal{D}, s_1, s_1, \beta \vDash_{2\downarrow 1} \{\mathtt{heap} := h_2 \parallel \bar{\mathtt{x}} := \bar{x}_2\}_2\delta_j$ holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \delta_j$ holds.

2. We have to show that shifting updates over $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$ does not change the result of the evaluation of the formulas in restricted two-state semantics.

   By Definition 31 we have:

   $$\begin{aligned}
   & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}(\phi_1 \wedge \phi_2) \\
   \Leftrightarrow\ & \mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2\downarrow 1} \phi_1 \wedge \phi_2 \\
   \Leftrightarrow\ & \mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2\downarrow 1} \phi_1 \text{ and } \mathcal{D}, s_1^u, s_2^u, \beta \vDash_{2\downarrow 1} \phi_2 \\
   \Leftrightarrow\ & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\phi_1 \text{ and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\phi_2 \\
   \Leftrightarrow\ & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\phi_1 \wedge \{u\}\phi_2
   \end{aligned}$$

   The other cases follow analog.

3. We need to show $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2\{v_1\}\dots\{v_n\}\phi$ if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u; v_1; \dots; v_n\}\{u_2; v_1; \dots; v_n\}_2\phi$.

   $$\begin{aligned}
   & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2\{v_1\}\dots\{v_n\}\phi \\
   \Leftrightarrow\ & \mathcal{D}, s_u, s_u, \beta \vDash_{2\downarrow 1} \{u_2\}_2\{v_1\}\dots\{v_n\}\phi \\
   & \text{where } \{u\} \text{ defines completely the state } s_u \\
   \Leftrightarrow\ & \mathcal{D}, s_u, s_{u_2}, \beta \vDash_{2\downarrow 1} \{v_1\}\dots\{v_n\}\phi \\
   & \text{where } \{u_2\}_2 \text{ defines completely the state } s_{u_2} \\
   \Leftrightarrow\ & \mathcal{D}, s_u^{v_1,\dots,v_n}, s_{u_2}^{v_1,\dots,v_n}, \beta \vDash_{2\downarrow 1} \phi
   \end{aligned}$$

where $s_u^{v_1,\ldots,v_n}$ and $s_{u_2}^{v_1,\ldots,v_n}$ are the states s_u and s_{u_2} updated according to $\{v_1\}\ldots\{v_n\}$

$\Leftrightarrow \mathcal{D}, s_u^{v_1,\ldots,v_n}, s_u^{v_1,\ldots,v_n}, \beta \vDash_{2\downarrow 1} \{u_2; v_1; \ldots; v_n\}_2\phi$
because $\{u_2; v_1; \ldots; v_n\}_2$ defines completely the state $s_{u_2}^{v_1,\ldots,v_n}$

$\Leftrightarrow \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u; v_1; \ldots; v_n\}\{u_2; v_1; \ldots; v_n\}_2\phi$
because $\{u; v_1; \ldots; v_n\}$ defines completely the state $s_u^{v_1,\ldots,v_n}$

4. (1) We have to show that $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2 \ltimes \phi$ holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2\phi \leftrightarrow \{u_2\}\{u\}_2\phi$ holds.

$$
\begin{aligned}
&\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2 \ltimes \phi \\
\Leftrightarrow\ &\mathcal{D}, s_u, s_{u_2}, \beta \vDash_{2\downarrow 1} \ltimes \phi \\
\Leftrightarrow\ &\mathcal{D}, s_u, s_{u_2}, \beta \vDash_{2\downarrow 1} \phi \\
&\quad \text{iff } \mathcal{D}, s_{u_2}, s_u, \beta \vDash_{2\downarrow 1} \phi \\
\Leftrightarrow\ &\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2\phi \\
&\quad \text{iff } \mathcal{D}, s_2, s_1, \beta \vDash_{2\downarrow 1} \{u_2\}\{u\}_2\phi \\
\Leftrightarrow\ &\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2\phi \leftrightarrow \{u_2\}\{u\}_2\phi
\end{aligned}
$$

(2) We have to show that $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2[\alpha]\phi$ holds if and only if

$$
\begin{aligned}
\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1}\ &\forall h_{post}.\forall h_{post2}.\forall \bar{x}_{post}.\forall \bar{x}_{post2}. \\
&\{u\}\langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post}) \\
&\wedge \{u_2\}\langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2}) \\
&\rightarrow \{\mathtt{heap} := h_{post} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post}\} \\
&\quad \{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2\phi
\end{aligned}
$$

holds.

$\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2[\alpha]\phi$

$\Leftrightarrow \mathcal{D}, s_u, s_{u_2}, \beta \vDash_{2\downarrow 1} [\alpha]\phi$

$\Leftrightarrow \mathcal{D}, s_u^\alpha, s_{u_2}^\alpha, \beta \vDash_{2\downarrow 1} \phi$ for all s_u^α, $s_{u_2}^\alpha$ such that α started in s_u, s_{u_2} terminates in s_u^α, $s_{u_2}^\alpha$, respectively

$\Leftrightarrow$ forall h_{post}, h_{post2}, $\bar{x}_{post}$, $\bar{x}_{post2}$: if α started in s_u terminates in the state described by h_{post}, $\bar{x}_{post}$ (which we denote by s_u^α) and if α started in s_{u_2} terminates in the state described by h_{post2}, $\bar{x}_{post2}$ (which we denote by $s_{u_2}^\alpha$) then $\mathcal{D}, s_u^\alpha, s_{u_2}^\alpha, \beta \vDash_{2\downarrow 1} \phi$

$$
\begin{aligned}
&\Leftrightarrow \text{forall } h_{post}, h_{post2}, \bar{x}_{post}, \bar{x}_{post2}: \\
&\qquad \text{if } \mathcal{D}, s_u, \beta \vDash \langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post}) \\
&\qquad \text{and } \mathcal{D}, s_{u_2}, \beta \vDash \langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2}) \\
&\qquad \text{then } \mathcal{D}, s_u^\alpha, s_{u_2}^\alpha, \beta \vDash_{2\downarrow 1} \phi \\
&\Leftrightarrow \text{forall } h_{post}, h_{post2}, \bar{x}_{post}, \bar{x}_{post2}: \\
&\qquad \text{if } \mathcal{D}, s_u, s_u, \beta \vDash_{2\downarrow 1} \langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post}) \\
&\qquad \text{and } \mathcal{D}, s_{u_2}, s_{u_2}, \beta \vDash_{2\downarrow 1} \langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2}) \\
&\qquad \text{then } \mathcal{D}, s_u^\alpha, s_u^\alpha, \beta \vDash_{2\downarrow 1} \{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2\phi \\
&\Leftrightarrow \text{forall } h_{post}, h_{post2}, \bar{x}_{post}, \bar{x}_{post2}: \\
&\qquad \text{if } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post}) \\
&\qquad \text{and } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u_2\}\langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2}) \\
&\qquad \text{then } \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{\mathtt{heap} := h_{post} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post}\} \\
&\qquad\qquad\qquad \{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2\phi \\
&\Leftrightarrow \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \\
&\qquad \forall h_{post}.\forall h_{post2}.\forall \bar{x}_{post}.\forall \bar{x}_{post2}. \\
&\qquad\qquad \{u\}\langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post}) \\
&\qquad\qquad \wedge \{u_2\}\langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2}) \\
&\qquad\qquad \rightarrow \{\mathtt{heap} := h_{post} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post}\} \\
&\qquad\qquad\quad \{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2\phi
\end{aligned}
$$

(3) We have to show that $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2\langle\alpha\rangle\phi$ holds if and only if

$$
\begin{aligned}
\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1}\ & \exists h_{post}.\exists h_{post2}.\exists \bar{x}_{post}.\exists \bar{x}_{post2}. \\
& \{u\}\langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post}) \\
& \wedge \{u_2\}\langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2}) \\
& \wedge\ \{\mathtt{heap} := h_{post} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post}\} \\
& \quad \{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2\phi
\end{aligned}
$$

holds.

$$
\begin{aligned}
&\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2\langle\alpha\rangle\phi \\
&\Leftrightarrow \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2\neg[\alpha]\neg\phi \\
&\Leftrightarrow \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \neg\{u\}\{u_2\}_2[\alpha]\neg\phi
\end{aligned}
$$

$$
\begin{aligned}
\Leftrightarrow\ & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \\
& \neg\, \forall h_{post}.\forall h_{post2}.\forall \bar{x}_{post}.\forall \bar{x}_{post2}. \\
& \qquad \{u\}\langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post}) \\
& \qquad \wedge \{u_2\}\langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2}) \\
& \qquad \rightarrow \{\mathtt{heap} := h_{post} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post}\} \\
& \qquad\quad \{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2 \neg\phi \\
\Leftrightarrow\ & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \\
& \exists h_{post}.\exists h_{post2}.\exists \bar{x}_{post}.\exists \bar{x}_{post2}. \\
& \qquad \{u\}\langle\alpha\rangle(\mathtt{heap} = h_{post} \wedge \bar{\mathtt{x}} = \bar{x}_{post}) \\
& \qquad \wedge \{u_2\}\langle\alpha\rangle(\mathtt{heap} = h_{post2} \wedge \bar{\mathtt{x}} = \bar{x}_{post2}) \\
& \qquad \wedge\ \{\mathtt{heap} := h_{post} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post}\} \\
& \qquad\quad \{\mathtt{heap} := h_{post2} \mid\mid \bar{\mathtt{x}} := \bar{x}_{post2}\}_2 \phi
\end{aligned}
$$

(4) We have to show that $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2\phi_p$ holds if and only if $\mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\phi_p$ holds (where ϕ_p is a predicate).

$$
\begin{aligned}
& \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\{u_2\}_2\phi_p \\
\Leftrightarrow\ & \mathcal{D}, s_u, s_{u_2}, \beta \vDash_{2\downarrow 1} \phi_p \\
\Leftrightarrow\ & \mathcal{D}, s_u, \beta \vDash \phi_p \\
\Leftrightarrow\ & \mathcal{D}, s_u, s_u, \beta \vDash_{2\downarrow 1} \phi_p \\
\Leftrightarrow\ & \mathcal{D}, s_1, s_2, \beta \vDash_{2\downarrow 1} \{u\}\phi_p
\end{aligned}
$$

5. The construction terminates, because in each step the updates $\{u\}\{u_2\}_2$ are either pushed to subformulas or $\{u_2\}_2$ is eliminated.

□

7.5 Discussion

This chapter presents a two-state logic and an approximate calculus for it. The calculus can be used to efficiently reason about conditional noninterference, a property which can be expressed naturally in the logic. The approach was inspired by Amtoft et al. [2006]. The main advances of the presented approach are its less restrictive syntax and the possibility to switch on-the-fly from the approximate reasoning to precise self-composition style reasoning. The approximate calculus mainly excludes infeasible combinations of paths through the program and at some point approximates $\ltimes$ formulas. Values of program

variables are abstracted by invariants and contracts only and thus can be kept highly precise. Switching to self-composition style preserves this precise knowledge on *both* program states. Thus, the self-composition style reasoning approach from Chapter 5 can build on precise knowledge to draw precise conclusions. This is a unique property of the presented two-state logic and calculus.

8 Combining Analysis Techniques

This chapter shows that approximate and precise information flow verification techniques can also be integrated on the specification level. To this end, a semantically correct translation of the most important specification elements of the Java Information Flow approach by Myers [1999a] into the JML extension of Chapter 4 is presented. Because both, the verification techniques presented in this thesis as well as JIF, are modular on the method level, this allows to verify each method with the technique most appropriate for it.

A first version of the translation was developed in the Bachelor's thesis of Nikolov [2014] which was supervised by the author of this thesis. The presented translation is a substantially revised version of the approach discussed by Nikolov [2014].

JIF specifications are written in the decentralized label model (DLM) by Myers and Liskov [2000]. DLM contains static and dynamic features. This chapter considers the static features of DLM only, since this thesis is interested in static program analysis.

8.1 The Decentralized Label Model (DLM)

Listing 8.1 shows a slightly modified version of the password checker example from Listing 2.1, implemented in JIF. This section explains the considered DLM specifications based on the implementation shown in Listing 8.1.

The most basic block of DLM are *principals*. Principals are thought of as entities with the ability to observe and modify parts of a program. Principals are related by a reflexive and transitive *acts-for relation* $\succcurlyeq$. A principal q which acts for p, written $q \succcurlyeq p$, has at least the same abilities as p. DLM has a top principal $\star$ and a bottom principal _, that is, $\star \succcurlyeq p \succcurlyeq \_$ for all principals p. The example in Listing 8.1 uses the bottom principal _ and a principal called `root`, where the principal hierarchy is $\star \succcurlyeq$ `root` $\succcurlyeq \_$.

DLM primarily annotates types with *labels*. In line 3 of Listing 8.1, the type **`int`** is annotated with the label `{root->}`. In general, labels have the form

```
1  class PasswordFile authority(root) {
2    private int{root->}[]{root->} names, passwords;
3    private int{root->} numOfFailedChecks;
4
5    public boolean{_->_} check{root->} (int{_->_} user,
6                                        int{_->_} password) : {_->_}
7      where authority(root)
8    {
9      try {
10       for (int i = 0; i < names.length; i++) {
11         if (names[i] == user && passwords[i] == password) {
12             numOfFailedChecks = 0;
13           declassify ({root->} to {_->_}) {
14             return true;
15           }
16         }
17       }
18     } catch (NullPointerException e) {
19     } catch (ArrayIndexOutOfBoundsException e) {
20     }
21
22     numOfFailedChecks++;
23     declassify ({root->} to {_->_}) {
24       return false;
25     }
26   }
27 }
```

Listing 8.1: Example of a password checker in JIF.

`{c;d}`, where `c` is a *confidentiality policy* and `d` is an *integrity policy*. Confidentiality policies are build up from elementary confidentiality policies `o->r` in combination with the join $\sqcup$ and meet $\sqcap$ of confidentiality policies. Elementary confidentiality policies `o->r` consist of two principals `o` and `r`. The principal `o` is called owner of the policy and the principal `r` is called reader of the policy. If `r` is omitted, then the top principal $\star$ is assumed. Likewise, integrity policies are build up from elementary integrity policies `o<-w` in combination with the join $\sqcup$ and meet $\sqcap$ of integrity policies. In integrity policies, `o` is called owner of the policy and `w` is called writer of the policy. If `w` is omitted, then also the top principal $\star$ is assumed.

Intuitively, a confidentiality policy `o->r` expresses that all principals p with $o \succcurlyeq p$ think that only principals q with $q \succcurlyeq o$ or $q \succcurlyeq r$ should be able to read the information the policy is assigned to. Formally, the set of readers which a principal p believes should be able to read the information is defined by the function $readers(p, \texttt{o->r}) = \{q \mid o \succcurlyeq p \text{ implies } (q \succcurlyeq o \text{ or } q \succcurlyeq r)\}$. Similarly, the set of writers which a principal p believes should be able to change some information is defined by the function $writers(p, \texttt{o<-r}) = \{q \mid o \succcurlyeq p \text{ implies } (q \succcurlyeq o \text{ or } q \succcurlyeq r)\}$. The sets of readers and writers can be extended to joint and meet policies as follows:

$$\begin{aligned} readers(p, c \sqcup d) &= readers(p, c) \cap readers(p, d) \\ readers(p, c \sqcap d) &= readers(p, c) \cup readers(p, d) \\ writers(p, c \sqcup d) &= writers(p, c) \cup writers(p, d) \\ writers(p, c \sqcap d) &= writers(p, c) \cap writers(p, d) \end{aligned}$$

Confidentiality and integrity policies are preordered by the set of readers and writers, respectively. A confidentiality policy c is less restrictive than d, written $c \sqsubseteq_{\mathrm{C}} d$, if and only if $readers(p, c) \supseteq readers(p, d)$ for all principals p. An integrity policy c is less restrictive than d, written $c \sqsubseteq_{\mathrm{I}} d$, if and only if $writers(p, c) \subseteq writers(p, d)$ for all principals p. $\sqsubseteq_{\mathrm{C}}$ and $\sqsubseteq_{\mathrm{I}}$ are preorders, their equivalence classes form lattices with join operator $\sqcup$ and meet operator $\sqcap$. The policy `_->_` is the least restrictive confidentiality policy, $\star$`->`$\star$ is the most restrictive one. Likewise, $\star$`<-`$\star$ is the least restrictive integrity policy and `_<-_` is the most restrictive one.

Labels are ordered according to $\sqsubseteq_{\mathrm{C}}$ and $\sqsubseteq_{\mathrm{I}}$: a label $\{c_1; d_1\}$ is less restrictive than $\{c_2; d_2\}$, written $\{c_1; d_1\} \sqsubseteq_{\mathrm{L}} \{c_2; d_2\}$, if and only if $c_1 \sqsubseteq_{\mathrm{C}} c_2$ and $d_1 \sqsubseteq_{\mathrm{I}} d_2$. The equivalence classes of the set of labels according to $\sqsubseteq_{\mathrm{L}}$ form in combination with join operator $\sqcup$ and meet operator $\sqcap$ a lattice, where $\{c_1; d_1\} \sqcup \{c_2; d_2\}$ is defined as $\{c_1 \sqcup c_2; d_1 \sqcup d_2\}$ and $\{c_1; d_1\} \sqcap \{c_2; d_2\}$ is defined as $\{c_1 \sqcap c_2; d_1 \sqcap d_2\}$.

If the confidentiality policy is omitted in a label, then the *least* restrictive confidentiality policy `_->_` is assumed. If the integrity policy is missing, then the

most restrictive integrity policy _<-_ is assumed. In Listing 8.1 all integrity policies are omitted. Therefore, in all cases the most restrictive integrity policy _<-_ is assumed.

In contrast to usual types, arrays have two labels: the usual type label placed after the type declaration and a base type label placed between the base type and the square brackets. The usual type label refers to the array object whereas the base type label refers to the content of the array. In the example in Listing 8.1 the array declaration in line 2 uses the same label as usual type label and base type label.

Labels are not only annotated to types, but occur at several additional places. In the context of this thesis method begin labels, method end labels and labels in declassifications are considered only.

Method begin and end labels constrain the control flow information hidden in the program counter. Method begin labels are placed right after the method name. They are an upper bound for the label of the program counter when the method is called. Method end labels are annotated with a preceding colon after the closing bracket of the parameter list. It is an upper bound for the label of the program counter on termination of the method.

Declassification in DLM is in essence a relabeling of an expression or the program counter. Listing 8.1 shows in lines 13 to 15 and lines 23 to 25 a relabeling of the program counter: if the program counter has a less restrictive label than the first label after the `declassify` keyword, then the program counter is relabeled to the second label for the code block between the curly brackets. Beside the program counter, expressions can be declassified. The declassification of an expression e has the form `declassify(`e`, `c` to `d`)`, where c and d are labels. In case the label of e is less restrictive than c, then e is relabeled to d.

Beside information flow control mechanisms, DLM contains some access control mechanisms. Part of the access control mechanisms is that declassification may take place only if the corresponding code is executed with sufficient authority. Lines 1 and 7 in Listing 8.1 serve this purpose. Access control is, however, not subject to this thesis and therefore the corresponding specifications of DLM are not considered any further.

The overall semantics of a DLM specification is given by the lattice of labels (Myers [1999b]): information stored in a variable or field labeled with L may flow at most to variables and fields which are labeled with $L' \sqsupseteq_{\mathrm{L}} L$; except for information which is released by a `declassify` statement: this information may also flow to variables and fields which are less restrictive than L, but more restrictive than the second label of the `declassify` statement. Because

`declassify` statements may occur after complex computations, it is often difficult to figure out what information exactly is allowed to flow in a DLM specification. It is already difficult to see on a first glance what information exactly is declassified in Listing 8.1.

8.2 Basic JIF to JML Translation

The basic translation does not consider arrays and `declassify` statements. It consists of three steps. Firstly, the security lattice and the security policy in the sense of Section 3.4 are extracted from the DLM specification according to its semantics. Afterwards, we use Lemma 7 to express the multilevel noninterference property by a set of two-level noninterference properties. The set of two-level noninterference properties can then be specified in the JML extension from Chapter 4 by a set of determines clauses.

8.2.1 Extraction of the Security Lattice and Security Policy from DLM Annotations

Extracting the security policy from DLM annotations is easy: the security policy is the mapping of program variables and fields to the labels annotated to them. Method begin and end labels need to be considered separately, because they constrain the information implicitly stored in the program counter. How they are translated is described in Section 8.2.3.

The label ordering can be determined according to the complete relabeling rule (Myers [1999b]). In essence, $\{c_1; d_1\} \sqsubseteq_{\mathrm{L}} \{c_2; d_2\}$ if

- c_1 can be relabeled to c_2 by an arbitrary, repeated application of the following incremental relabeling rules:
  - A reader `r` may be replaced by a reader `r2` if `r2` acts for `r`: an elementary confidentiality policy `o->r` may be relabeled to `o->r2` if `r2`$\succcurlyeq$`r`.
  - An owner `o` may be replaced by an owner `o2` if `o2` acts for `o`: an elementary confidentiality policy `o->r` may be relabeled to `o2->r` if `o2`$\succcurlyeq$`o`.
  - A new elementary confidentiality policy may be added to the label: L may be relabeled to $L \sqcup$ `o->r` for arbitrary principals `o` and `r`. (Remember that $\sqcup$ means "and" for confidentiality policies.)

- d_1 can be obtained from d_2 by an arbitrary, repeated application of the following rules:
  - An integrity policy may be removed: $I \sqcup$ o<-w may be replaced by I for arbitrary principals o and w.
  - A writer w may be replaced by a writer w2 that it acts for: an elementary integrity policy o<-w may be replaced by o<-w2 if w$\succcurlyeq$w2.
  - A policy o<-w may be added to an integrity policy $I \sqcup$ o2<-w if o2 acts for o, that is, $I \sqcup$ o2<-w may be replaced by $I \sqcup$ o2<-w $\sqcup$ o<-w if o2$\succcurlyeq$o.

An efficient implementation of the containment check can be found in the JIF system.

A representative is chosen for the labels of each equivalence class and all labels are replaced by their representative in the security policy extracted from the DLM specification.

8.2.2 Translating Multi-Level Noninterference to Two-Level Noninterference

The multilevel noninterference property flow$(\alpha, \mathcal{L}, p)$ characterized by the security lattice $\mathcal{L} = (L, \sqsubseteq_{\mathrm{L}})$ (where L is the set of equivalence classes of labels) in combination with the security policy p from the last section can be expressed according to Lemma 7 as a set of two-level noninterference properties: let FP be the (finite) set of fields and program variables (without heap) occurring in the program under consideration and let $\mathrm{FP}_{\sqsubseteq_{\mathrm{L}} p(x)}$ be the set of all $y \in \mathrm{FP}$ such that $p(y) \sqsubseteq_{\mathrm{L}} p(x)$. Further, let n be the number of elements in $\mathrm{FP}_{\sqsubseteq_{\mathrm{L}} p(x)}$ and let for all $x \in \mathrm{FP}$ the sequence $Seq_{\sqsubseteq_{\mathrm{L}} p(x)} = \langle x_1, \ldots, x_n \rangle$ be defined such that

- for all $y \in \mathrm{FP}_{\sqsubseteq_{\mathrm{L}} p(x)} \cap \mathrm{PV}$ (where PV denotes the set of program variables, see page 8) there exists a unique index i such that $x_i = y$ and
- for all $y \in \mathrm{FP}_{\sqsubseteq_{\mathrm{L}} p(x)} \cap \mathit{Field}$ there exists an index i such that $x_i = \mathit{toSeq}($heap$, y)$ where the function $\mathit{toSeq} : \mathit{Heap} \times \mathit{Field} \to \mathit{Seq}$ maps each heap-field-pair (h, f) to a sequence $\langle o_1.f, \ldots, o_m.f \rangle$ such that $\{o_1, \ldots, o_m\}$ is the set of all objects created in h which have a field f. In case the number of created objects is infinite (this is possible in heaps which are not wellformed), the result of toSeq is underspecified.

Similar to Lemma 7, flow$(\alpha, \mathcal{L}, p)$ holds if and only if flow$(\alpha, Seq_{\sqsubseteq_{\mathrm{L}} p(x)}, x)$ holds for all $x \in \mathrm{FP} \cap \mathrm{PV}$ and flow$(\alpha, Seq_{\sqsubseteq_{\mathrm{L}} p(x)}, \mathit{toSeq}($heap$, x))$ holds for all $x \in \mathrm{FP} \cap \mathit{Field}$.

8.2.3 Specifying a set of Two-Level Noninterference properties in JML

The information flow properties $\{\text{flow}(\alpha, Seq_{\sqsubseteq_{\mathrm{L}} p(x)}, x) \mid x \in \mathrm{FP} \cap \mathrm{PV}\}$ and $\{\text{flow}(\alpha, Seq_{\sqsubseteq_{\mathrm{L}} p(x)}, toSeq(\texttt{heap}, x)) \mid x \in \mathrm{FP} \cap \overline{Field}\}$ can easily be expressed in the JML extension from Chapter 4 by information flow contracts: each information flow property $\text{flow}(\alpha, Seq_{\sqsubseteq_{\mathrm{L}} p(x)}, x)$ is specified according to the semantics of determines clauses from Section 4.3 by a clause

determines x **\by** y_1, ..., y_n;

where $Seq_{\sqsubseteq_{\mathrm{L}} p(x)} = \langle y_1, \ldots, y_n \rangle$. In the determines clause, the function $toSeq$ is replaced by a new function `\to_seq` of type **\seq**. In contrast to $toSeq$, the function `\to_seq` has only one argument: a field. `\to_seq` is always interpreted in the current heap and therefore does not take an explicit heap argument. Hence, `\to_seq(f)` is equivalent to $toSeq$(`heap`,`f`).

The translation of method begin and end labels remains to be explained. A begin label L allows assignments only to variables and fields which are annotated by a more restrictive label $L' \sqsupseteq_{\mathrm{L}} L$. This property can be expressed by a usual JML assignable clause:

assignable `*.`y_1, ..., `*.`y_n;

where $\{y_1, \ldots, y_n\}$ is the set of all fields with $p(y_i) \sqsupseteq_{\mathrm{L}} L$ and where `*.f` represents the location set $\{(o, f) \mid o \in Obj\}$.

The end label restricts the information which may leak by observing whether the method terminates normally or whether an exception is thrown. For the translation of the end label we extend JML by the keyword `\exception` which is interpreted as a special variable. This variable points to the thrown exception, if any is thrown, or to **null** otherwise. An end label L can then be translated by the determines clause

determines `\exception` **\by** y_1, ..., y_n;

where $\{y_1, \ldots, y_n\}$ is the set of all variables and fields with $p(y_i) \sqsubseteq_{\mathrm{L}} L$.

Note that JIF checks for object-oriented secure information flow. Therefore, we have to use the `\new_objects` keyword from Chapter 6 to list all newly created objects in the determines clauses from above. Additionally, we have to add the modifier **nullable** to each field of object type and the modifier **helper** to each method, in order to disable all JML default specifications.

8.3 Optimizations

The basic translation may lead to a large set of determines clauses. The set of clauses can be reduced in two ways.

Firstly, all determines clauses with the same **\by** part can be combined into one clause: the set of clauses

determines x_1 **\by** y_1, ..., y_n;

⋮

determines x_m **\by** y_1, ..., y_n;

is semantically equivalent to

determines x_1, ..., x_m **\by** y_1, ..., y_n;

and hence can be replaced by this one.

Secondly, one can omit all those determines clauses

determines x **\by** y_1, ..., y_n;

where x is a variable or field whose value is not changed by the method. All determines clauses for fields which do not occur in the assignable clause of the method can be omitted. A simple static analysis—which approximates the set of assigned variables and fields syntactically—should reduce the number of determines clauses further.

8.4 Translation of Arrays

The translation of arrays is limited to one-dimensional arrays up to now. The type label placed after the square brackets translates like the label of any other field, but the additional base type label needs special treatment. To this end, we introduce the function $contentToSeq : Heap \times Field \to Seq$ which maps each heap-field-pair (h, f) to a sequence

$$\langle o_1.f[0], \ldots, o_1.f[o_1.f.length - 1], \ldots, o_m.f[0], \ldots, o_m.f[o_m.f.length - 1] \rangle$$

such that $\{o_1, \ldots, o_m\}$ is the set of all objects created in h which have a field f. In case the field f is not of type Seq or in case the number of created objects is infinite, the result of $contentToSeq$ is underspecified. For every field f of type array we add $\mathrm{flow}(\alpha, Seq_{\sqsubseteq_L p(f)}, contentToSeq(\mathtt{heap}, f))$ to the set of two-state noninterference properties and additionally add $contentToSeq(\mathtt{heap}, f)$ to all sequences $Seq_{\sqsubseteq_L p(x)}$ with $p(f) \sqsubseteq_L p(x)$. In JML, the function $contentToSeq$

is replaced by a new function `\content_to_seq` of type **`\seq`**. The function `\content_to_seq` is always interpreted in the current heap and therefore takes only one argument, a field: `\content_to_seq(f)` is equivalent to $contentToSeq(\texttt{heap},\texttt{f})$.

The assignable clause resulting form the translation of the begin label L has to be joined with `*.f.*` for all fields f of array type whose base type is labeled with $L' \sqsupseteq_{\mathrm{L}} L$. The notation `*.f.*` stands for the location set $\{(o.f, g) \mid o \in Obj, g \in Field\}$

8.5 Translation of Declassify Statements

The translation of declassify statements is possible, but difficult. Because declassify statements may occur after complex computations, it seems to be questionable whether there exists a good algorithm for the translation of declassify statements. Already the translation of the declassify statements from Listing 8.1 is non-trivial, as the example in the next section shows. Those statements have to be translated on a case-by-case basis to JML declassifies statements which are part of determines clauses.

8.6 Example

The JIF specification from Listing 8.1 translates to the JML specification in Listing 8.2: the fields `numOfFailedChecks`, `names` and `passwords` as well as the base type of `names` and `passwords` have the most restrictive label occurring in the example and therefore may depend on any other variable and field. The determines clauses resulting from the basic translation of `numOfFailedChecks`, `names`, `passwords` and their base types have been merge into one (lines 5 to 11). The result of the method, denoted by **`\result`** in JML, has the least restrictive label and therefore may (theoretically) depend on the parameters `user` and `password` only (line 12). A careful examination of the declassify statements reveals that **`\result`** may additionally depend on the accumulated information whether (1) `names` is **`null`** or (2) `passwords` is **`null`** or (3) if for all indices `i` grater than zero and smaller than `names.length` and smaller than `passwords.length` the passed user name is different from `names[i]` or the passed password is different from `passwords[i]` (lines 13 to 20). The parameters `user` and `password` cannot be accessed after termination of the method any more and therefore their determines clauses are omitted. The end label is the least restrictive label and therefore whether or not an exception is thrown may depend at most on the parameters `user` and `password`

```
1  class PasswordFile {
2    private /*@ nullable */ int[] names, passwords;
3    private int numOfFailedChecks;
4
5    /*@ determines \to_seq(names), \content_to_seq(names),
6      @            \to_seq(passwords), \content_to_seq(passwords),
7      @            \to_seq(numOfFailedChecks)
8      @        \by user, password,
9      @            \to_seq(names), \content_to_seq(names),
10     @            \to_seq(passwords), \content_to_seq(passwords),
11     @            \to_seq(numOfFailedChecks);
12     @ determines \result \by user, password
13     @            \declassifies
14     @                names == null ||
15     @                passwords == null ||
16     @                ( \forall int i;
17     @                        0 <= i && i < names.length
18     @                     && i < passwords.length;
19     @                        names[i] != user
20     @                     || passwords[i] != password );
21     @ determines \exception \by user, password;
22     @ assignable *.names, *.names.*, *.passwords, *.passwords.*,
23     @            *.numOfFailedChecks;
24     @*/
25   //@ helper
26   public boolean check(int user, int password)
27   {
28     try {
29       for (int i = 0; i < names.length; i++) {
30         if (names[i] == user && passwords[i] == password) {
31           numOfFailedChecks = 0;
32           return true;
33         }
34       }
35     } catch (NullPointerException e) {
36     } catch (ArrayIndexOutOfBoundsException e) {
37     }
38
39     numOfFailedChecks++;
40     return false;
41   }
42 }
```

Listing 8.2: Example translation to JML.

(line 21). Finally, the begin label is the most restrictive label of the example and therefore the method may assign to the fields `numOfFailedChecks`, `names` and `passwords` and the contents of the arrays `names` and `passwords` only (lines 22 to 23).

8.7 Discussion

The presented DLM to JML translation is a first step towards a comprehensive integration of type-based and deductive information flow verification on the specification level. It may serve as a basis for the translation of RIFL (which is a tool-independent specification language for information flow properties developed within the priority program "Reliably Secure Software Systems (RS^3)") specifications into the JML extension of Chapter 4. An alternative to the integration of type-based and deductive information flow verification on the specification level is presented by Banerjee et al. [2008]: the authors design a security type system that can make direct use of declassification statements comparable to the ones of the JML extension. Though this is an elegant solution, the integration on the specification level has the advantage that existing tools can be reused without modifications.

9 Case Studies

9.1 A Simple Electronic Voting System

Electronic voting (e-voting) systems which are used in public elections need to fulfill a broad range of strong requirements concerning both safety and security. Among those requirements are reliability, robustness, privacy of votes, coercion resistance and universal verifiability. Bugs in or manipulations of an e-voting system might have considerable influence on the life of humans. Therefore, e-voting systems are an obvious target for software verification. This case study proves the preservation of privacy of votes for a modified version of an e-voting system implemented by the group of Ralf Küsters at the University of Trier. The main modification compared to the original implementation described in Küsters et al. [2013] is that messages are transmitted synchronously instead of asynchronously. To the best of the authors knowledge this is the first time that preservation of privacy of votes could be shown *on the code level* for a (simple) e-voting system. Altogether the considered code comprises 8 classes and 19 methods in about 150 lines of code of a rich fragment of Java.

To prove that the system merely reveals the result of the election and nothing else about the votes, it is unavoidable to prove the functional property that the program calculates the result of the election correctly. The major part of the functional specification and verification was carried out by Bruns [2014]. The information flow specification and verification, on the other hand, was performed by the author of this thesis. The presentation of the case study focuses on the information flow part.

9.1.1 Verifying Programs containing Cryptography

The e-voting system consists of clients which send encrypted votes over a publicly available channel. A server receives and counts the messages. The public channel can be manipulated by an active adversary: adversaries may receive, drop or manipulate messages and they may do arbitrary polynomially bounded computations. Adversaries *cannot* access the memory of the client or server directly. In this system *strong secrecy* of votes—which can be expressed

as noninterference—is not fulfilled: the message which is sent over the network depends on the vote and could theoretically be decrypted by an adversary with unbounded computational power. As a consequence, KeY classifies the program insecure though it is secure from a cryptographic point of view. Küsters et al. [2011] proposed a solution to this problem: the authors showed that the real encryption of the system can be replaced by an implementation of ideal encryption. Ideal encryption completely decouples the sent message from the secret. Even an adversary with unbounded computational power cannot decrypt the message. The receiver can decrypt the message by some extra information sent over a secret channel which is not observable by adversaries. Küsters et al. [2011] showed that if in the system with ideal encryption votes do not interfere with the output on the public channel, then the system with real encryption guarantees privacy of votes. Therefore, it is sufficient to analyze the system with ideal encryption.

9.1.2 The System and its Specification

Figure 9.1 shows an UML class diagram of the considered e-voting system. The implementation comprises, beside the client (class `Voter`) and the server, an interface to the environment[1] and a setup. The `main` method of the setup models the e-voting process itself. This is necessary, because the security property—that privacy of votes is preserved up to the result of the election—can only be formulated with regard to a complete e-voting process rather than the implementation of the client and server only.

Listing 9.1 shows the implementation of `main`. In essence, the adversary decides in a loop which client should send the next vote until the server signals that the result of the election is ready. More precisely, the adversary is asked by a call to the method `Environment.untrustedInput` which client should send its vote. If subsequently the method `onSendBallot` is called on the corresponding `Voter` object, the client sends its secret vote (stored in the attribute `vote`) with the help of an implementation of ideal encryption (synchronously) to the server. The server immediately counts the vote, if the voter has not voted before. Finally, the server is asked by a call to the method `resultReady` whether the result of the election is ready. If so, the result is published by a call to the method `publishResult`.

The overall security requirement is specified as a method contract for `main` (Listing 9.2). The contract states that under the condition that the server is

[1] The verified implementation uses an abstract class instead of an interface, because the original implementation does. From a modeling as well as a presentational point of view it is more convenient to use an interface though. For the specification and verification of secure information flow it does not make a difference.

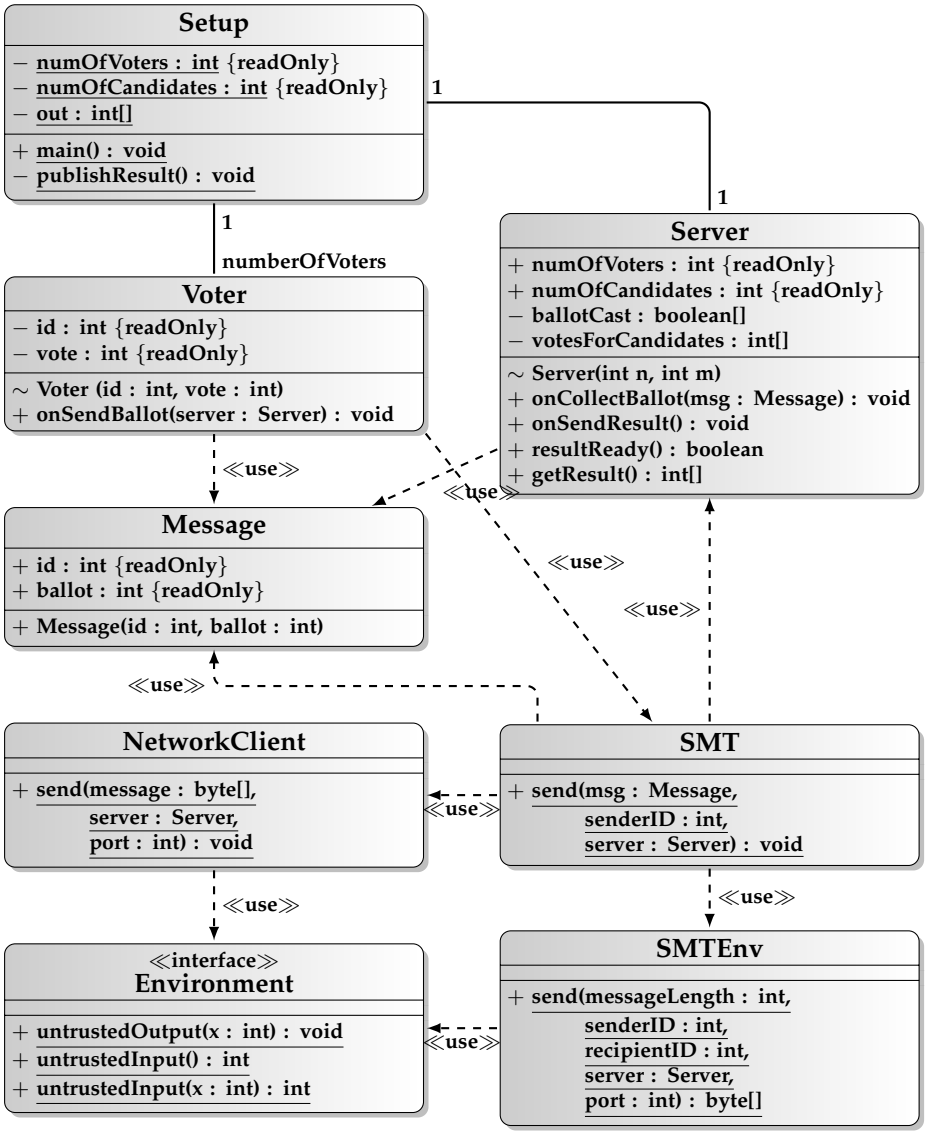


Figure 9.1: UML Class Diagram of the e-voting system.

```
public void main () {
    boolean resultReady = server.resultReady();
    while ( !resultReady ) { // possibly infinite loop
        // let adversary decide send order
        final int k = Environment.untrustedInput(voters.length);
        final Voter v = voters[k];
        v.onSendBallot(server);
        resultReady = server.resultReady();
    }
    publishResult();
}
```

Listing 9.1: Implementation of the `main` method.

```
/*@ normal_behavior
  @ requires    (\forall int j;
  @                0 <= j && j < numberOfVoters;
  @                !server.ballotCast[j]);
  @ requires    (\forall int i;
  @                0 <= i && i < numberOfCandidates;
  @                server.votesForCandidates[i]==0);
  @ determines  Environment.envState
  @             \by Environment.envState, numberOfVoters;
  @ determines  out, (\seq_def int i; 0; out.length; out[i])
  @             \by numberOfCandidates, numberOfVoters
  @             \declassifies (\seq_def int i; 0; numberOfCandidates;
  @                              (\num_of int j;
  @                                 0 <= j && j < numberOfVoters;
  @                                     voters[j].vote == i));
  @ assignable  server.ballotCast[*], server.votesForCandidates[*],
  @             Environment.rep, out;
  @ diverges true;
  @*/
public void main () { ... }
```

Listing 9.2: Information flow contract of the `main` method.

initialized correctly

1. the state of the environment, abstracted by `Environment.envState`, depends at most on its initial value as well as on the number of voters;
2. The array `out` itself as well as each of its entries (containing the final result of the election) depend at most on
   - the number of candidates,
   - the number of voters, and
   - for each candidate the correct sum of all votes for them;
3. at most locations of the server, the environment and the result array are changed; and
4. the election might not terminate (because the adversary might block votes of voters for ever).

The **`\declassifies`** keyword is syntactic sugar, but stresses that `out` depends on a well-considered part of the secret—the correct result of the election—only.

In order to show that `main` fulfills its contract, we need an information flow loop invariant (Listing 9.3). The loop invariant—read from bottom to top—states that

1. - the knowledge of the environment (`Environment.envState`),
   - the fact whether the result of the election is ready,
   - the number of voters, and
   - the information which voter has already been voted (stored in the array `server.ballotCast`)

   depend at most on the initial knowledge of the environment, whether the result of the election initially was ready, the initial number of voters, and the initial information which voter has already been voted;
2. at most the locations of the environment and the server are modified;
3. `resultReady` is true if and only if all voters voted;
4. the server calculated the partial result correctly; and
5. the class invariant of the class `Setup` is preserved.

Because `out` is not modified by the loop, it needs not be mentioned in the loop invariant explicitly. The fact that `out` itself as well as each of its entries depend at most on

```
/*@ maintaining \invariant_for(this);
  @
  @   // votes for candidates are sums from voters already voted
  @ maintaining (\forall int i; 0 <= i && i < numberOfCandidates;
  @                 server.votesForCandidates[i] ==
  @                     (\num_of int j; 0 <= j && j < numberOfVoters;
  @                             server.ballotCast[j]
  @                          && voters[j].vote == i));
  @
  @ maintaining     resultReady
  @              == (\forall int j; 0 <= j && j < numberOfVoters;
  @                     server.ballotCast[j]);
  @
  @ assignable server.ballotCast[*], server.votesForCandidates[*],
  @            Environment.rep;
  @ determines Environment.envState, resultReady, numberOfVoters,
  @            (\seq_def int i; 0; numberOfVoters;
  @               server.ballotCast[i])
  @        \by \itself;
  @*/
```

Listing 9.3: Loop invariant for the loop in `main`.

```
/*@ normal_behavior
  @ requires (\forall int i; 0 <= i && i < numberOfCandidates;
  @              server.votesForCandidates[i] ==
  @                  (\num_of int j; 0 <= j && j < numberOfVoters;
  @                     voters[j].vote == i));
  @ assignable out;
  @ determines  out, (\seq_def int i; 0; out.length; out[i])
  @         \by  numberOfCandidates, numberOfVoters,
  @              server.votesForCandidates
  @              \declassifies (\seq_def int i; 0; numberOfCandidates;
  @                                 (\num_of int j;
  @                                    0 <= j && j < numberOfVoters;
  @                                    voters[j].vote == i));
  @*/
private void publishResult () { ... }
```

Listing 9.4: Information flow contract of `publishResult`.

- the number of candidates,
- the number of voters, and
- for each candidate the correct sum of all votes for them

can be derived from the contract of `publishResult` (Listing 9.4) in combination with the assurance of the loop invariant that the server calculates the result correctly. Note that the functional knowledge that the server calculates the result correctly is mandatory for proving the declassification. Here, the tight integration of functional and information flow verification pays of.

The preservation of the loop invariant is proved with the help of contracts for `untrustedInput`, `onSendBallot` and `resultReady`. The method `untrustedInput` is declared in the interface `Environment` (Listing 9.5). This interface provides the connection to the environment which is controlled by the attacker. The state of the environment is abstracted by a (ghost) field of type sequence. Because any computable information can be encoded into a sequence of integers, this is a valid abstraction. Each method of the `Environment` has a contract which, in essence, guarantees that the environment cannot access any other part of the e-voting system. More precisely, each method is required to meet the following restrictions: (1) the final state of the environment depends at most on its initial state and the parameters of the method, (2) if the method has a result value, then also the result depends at most on the initial state of the environment and the parameters of the method, and (3) at most the state of the

```
public interface Environment {
    //@ public static ghost \seq envState;

    //@ public static model \locset rep;
    //@ public static represents rep = \locset(envState);
    //@ accessible rep : \locset(envState);

    /*@ normal_behavior
      @ ensures     true;
      @ assignable  rep;
      @ determines  Environment.envState, \result
      @         \by  Environment.envState;
      @*/
    //@ helper
    public static int untrustedInput();

    /*@ normal_behavior
      @ ensures     true;
      @ assignable  rep;
      @ determines  Environment.envState \by Environment.envState, x;
      @*/
    //@ helper
    public static void untrustedOutput(int x);

    /*@ normal_behavior
      @ ensures     0 <= \result && \result < x;
      @ assignable  rep;
      @ determines  Environment.envState, \result
      @             \by Environment.envState, x;
      @*/
    //@ helper
    public static int untrustedInput(int x);

}
```

Listing 9.5: Declaration of the interface `Environment`.

```
/*@ normal_behavior
  @ requires    ! server.ballotCast[id];
  @ requires    \invariant_for(server);
  @ ensures        server.votesForCandidates[vote]
  @             == \old(server.votesForCandidates[vote])+1;
  @ ensures     server.ballotCast[id];
  @ assignable  server.votesForCandidates[vote],
  @             server.ballotCast[id],
  @             Environment.rep;
  @ determines  Environment.envState \by \itself;
  @ also normal_behavior
  @ requires    server.ballotCast[id];
  @ requires    \invariant_for(server);
  @ ensures        \old(server.votesForCandidates[vote])
  @             == server.votesForCandidates[vote];
  @ ensures        \old(server.ballotCast[id])
  @             == server.ballotCast[id];
  @ assignable  Environment.rep;
  @ determines  Environment.envState \by \itself;
  @*/
public void onSendBallot(Server server) {
    Message message = new Message(id, vote);
    //@ set message.source = this;
    SMT.send(message, id, server);
}
```

Listing 9.6: Contract of `onSendBallot`.

environment is modified. The specification of `Environment` shows that the information flow specification and verification approach presented in this thesis can be used for the specification and verification of interfaces and consequently also for the specification and verification of open and interactive systems.[2]

The method `onSendBallot` of the class `Voter` generates a new message containing the vote of the voter and sends it over the network, see Listing 9.6. `onSendBallot` has two contracts. Both require that the invariant of the server holds and ensure that the final state of the environment depends at most on its initial value. They differ in the functional part (which was specified and verified by Bruns [2014]): the first contract requires additionally that the voter has not voted yet. In this case the contract ensures that the server counted the

[2]Note that we consider deterministic programs only.

vote correctly by incrementing `server.votesForCandidates[vote]`. The second contract requires that the voter did already vote and guarantees in this case that the server does not count the vote again.

The counting takes place in the method `onCollectBallot` declared in the class `Server`. It is called indirectly by `SMT.send`. Because `onCollectBallot` has a purely functional contract, it will not be considered in detail here. The same holds for the method `resultReady` which simply returns `true` if all voters voted.

The complete specification of the system consists of approximatly 270 lines of JML code.

9.1.3 Verification Effort

The verification of the functional part of the System was mainly carried out by Bruns [2014] and therefore is not reported here. The subsequent verification of its information flow took about four days. The final information flow proof consists of 23 subproofs with about 7,800 proof steps including 10 user interactions. The optimizations of Sections 5.2 and 5.3 proved to be indispensable for the scalability of the self-composition approach.

9.1.4 Discussion

To the best of the authors knowledge this is the first time that preservation of privacy of votes could be shown *on the code level* for a (simple) e-voting system. Systems like Bingo Voting (Bohli et al. [2007]), Civitas (Clarkson et al. [2008]), Helios (Adida [2008]) and Scantegrity (Chaum et al. [2009]), which are much (!) more elaborate, provide guarantees on the design level, but it is not clear whether their implementations preserve these guarantees. Clarkson et al. [2008] report on an information flow verification of Civitas with JIF, but it is not stated clearly which properties have been checked for.

More generally, programs containing implementations of cryptographic protocols have been verified rarely. Küsters et al. [2012] describe a framework for the cryptographic verification of Java-like programs, on which this case study relies on. Other approaches aim at the verification of cryptographic code in C or F# itself (Aizatulin et al. [2012], Bhargavan et al. [2008]) which, however, is not the focus of this case study.

The e-voting case study shows that precise information flow verification techniques like the ones presented in this thesis are essential for the verification

of complex information flow properties, in particular for the verification of expressive declassification. It also shows that the optimizations introduced in Sections 5.2 and 5.3 are indispensable for the feasibility of the self-composition approach.

9.2 Examples from Literature

Beside the e-voting case study, the approaches from Chapters 4, 5 and 6 have been successfully applied to about 100 smaller examples. These examples include many examples from literature (Darvas et al. [2005], Amtoft et al. [2006], Naumann [2006], Bubel et al. [2008]) and in particular all examples discussed in this thesis, including the examples from Sections 4.4 and 6.4. The examples can be inspected in the information flow version of KeY, available on the DeduSec KeY website: `http://www.key-project.org/DeduSec/`.

10 Conclusions

10.1 Summary

This thesis contributes to the specification and deductive verification of language-based secure information flow. It introduces a fine gained notion of secure (conditional) information flow based on observation expressions instead of program variables only. This allows for intuitive, knowledge-based information flow specifications. Programs fulfilling this notion of secure information flow are provably secure with respect to the considered attacker model.

Based on this notion of secure information flow, the thesis presents an extension of the Java Modeling Language (JML), the defacto standard language for behavioral specification of Java code, for the specification of information flow properties of programs, including expressive declassification. The presented approach states information flow properties in design by contract style, that is, information flow specifications are annotated in form of contracts to methods (or other code blocks). Declassification statements are part of these contracts and therefore *not* mere type casts as in other approaches. The extension is designed such that functional and information flow specifications integrate seamlessly.

On the verification side, this thesis presents a formalization of secure information flow in Java Dynamic Logic. This formalization in self-composition style allows the KeY tool to verify information flow properties out of the box. The naive approach, however, tends to be inefficient. A contribution on efficient self-composition style reasoning shows that if noninterference is formalized as in Theorem 9, then programs need to be symbolically executed only once. Further, the thesis shows that the number of final symbolic states to be compared can be reduced considerably if the program under consideration is compositional with respect to information flow. Both contributions are indispensable prerequisites for the scalability of the self-composition approach. A case-study on a simplified electronic voting system in cooperation with the research group of Prof. Ralf Küsters from the University of Trier underlines the potential of the presented approach. Efficient self-composition reasoning enables the verification of programs which were beyond the state of the art before, as the electronic voting case-study shows. Still, it is a heavyweight approach. In order to lighten

the burden of the verification process, the thesis presents an approximate information flow calculus along the lines of Amtoft et al. [2006] based on a two-state interpretation of Java Dynamic Logic formulas. This calculus has the unique property that it can switch on-the-fly from the approximate reasoning to precise self-composition style reasoning, if higher precision is needed anywhere in the proof. In essence, the approximate calculus excludes infeasible combinations of paths through the program by showing that the guards of conditional statements depend on publicly observable information only, but it keeps the knowledge on the program states highly precise. Switching to self-composition style preserves this precise knowledge on *both* runs. The soundness of the calculus is proven based on a novel two-state logic.

The so far mentioned specification and verification techniques target at Java programs, but handle variables and fields of object type similar to those of primitive type: two states are considered to be low equivalent if the values of all low variables coincide. Because references in Java are usually considered opaque (that is, references can be compared by the identity comparison operation == only), one can argue (see for instance Banerjee and Naumann [2002]) that it is sufficient to require the existence of an isomorphism between the values of object typed low variables only (instead of requiring equality). This thesis presents a feasible formalization of this notion of object-sensitive secure information flow in Java DL. The formalization is in self-composition style and compatible with the above verification techniques. The feasibility results from optimizations that are derived from an investigation into the concept of object-oriented secure information flow itself which is a contribution on its own.

Finally, an approach to combine deductive information flow verification with security type checking is considered. The thesis presents a semantically correct translation of the most important specifications of the prominent Java Information Flow approach by Myers [1999a] into the JML extension from Chapter 4. Because both techniques are modular on the method level, this allows verifying each method with the technique most appropriate for it.

The techniques described in Chapters 4, 5 and 6 have been implemented in the KeY system (accessible on the website `http://www.key-project.org/DeduSec/`) and have successfully been tested on examples from literature and, in cooperation with the research group of Prof. Ralf Küsters from the University of Trier, on an implementation of a simple electronic voting system. The case studies show that precise information flow verification techniques like the ones presented in this thesis are essential for the verification of complex information flow properties, in particular for the verification of expressive declassification. It also shows that the contributions on efficient self-composition style reasoning are indispensable for the feasibility of the self-composition approach.

10.2 Future Work

Modular verification of information flow properties takes advantage of the fact that methods usually affect a small part of the heap only. In KeY, specifying which part of the heap is affected by a method is based on the approach of *dynamic frames*. This approach is very flexible, but coming up with good dynamic frame specifications and their subsequent verification required about half of the overall verification effort in the case study. A considerable advance in this area would also significantly reduce the verification effort for the modular verification of information flow properties in KeY.

Another starting point in reducing the specification and therefore the overall verification overhead of the presented techniques is the automatic generation of auxiliary specifications like loop invariants and block contracts. The existence of good approximate information flow verification techniques promises the existence of good static analyses for the generation of information flow loop invariants and information flow block contracts.

A further possible line of future work is the implementation and experimental evaluation of the presented approximate information flow calculus. Studying the two-state logic of Chapter 7 in more detail could lead to additional, interesting insights. Likewise, studying and extending the JIF to JML translation would be a starting point for future work. The presented translation is a first step towards a comprehensive integration of security type checking and precise deductive information flow verification approaches which is certainly worth to be pursued further.

Finally, it would be interesting to study how the presented techniques can be extended to concurrent Java programs.

Publications

Parts of the presented work has already been published on well established conferences and workshops. Publications related to this thesis are:

Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *VSTTE 2014: 6th Working Conference on Verified Software: Theories, Tools, Experiments*, Lecture Notes in Computer Science. Springer International Publishing, 2014. To appear.

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Information flow in object-oriented software. In Gopal Gupta, editor, *Logic-Based Program Synthesis and Transformation, LOPSTR'13*. Springer Berlin Heidelberg, 2014. To appear.

Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen, and Mattias Ulbrich. The COST IC0701 verification competition 2011. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software*, volume 7421 of *Lecture Notes in Computer Science*, pages 3–21. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31761-3. doi: 10.1007/978-3-642-31762-0_2. URL `http://dx.doi.org/10.1007/978-3-642-31762-0_2`.

Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Jürgen Graf, and Christoph Scheben. A hybrid approach for proving noninterference and applications to the cryptographic verification of Java programs. 2013.

Christoph Scheben and Peter H. Schmitt. Verification of information flow properties of Java programs without approximations. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software*, volume 7421 of *Lecture Notes in Computer Science*, pages 232–249. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31761-3.

doi: 10.1007/978-3-642-31762-0_15. URL http://dx.doi.org/10.1007/978-3-642-31762-0_15.

Christoph Scheben and Peter H. Schmitt. Efficient self-composition for weakest precondition calculi. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 579–594. Springer International Publishing, 2014. ISBN 978-3-319-06409-3. doi: 10.1007/978-3-319-06410-9_39. URL http://dx.doi.org/10.1007/978-3-319-06410-9_39.

Further conference and workshop publications by the author are:

Jürgen Geisler and Christoph Scheben. Human processor modelling language (HPML): Estimate working memory load trough interaction. In *Analysis, Design, and Evaluation of Human-Machine Systems*, volume 10, pages 95–100, 2007.

Matthias Kuntz, Stefan Leue, and Christoph Scheben. Extending non-termination proof techniques to asynchronously communicating concurrent programs. In Andrei Voronkov, Laura Kovacs, and Nikolaj Bjorner, editors, *WING 2010*, volume 1 of *EPiC Series*, pages 132–147. EasyChair, 2012.

Christoph Scheben. Simulation of d'-dimensional cellular automata on d-dimensional cellular automata. In Samira Yacoubi, Bastien Chopard, and Stefania Bandini, editors, *Cellular Automata*, volume 4173 of *Lecture Notes in Computer Science*, pages 131–140. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-40929-8. doi: 10.1007/11861201_18. URL http://dx.doi.org/10.1007/11861201_18.

Bibliography

Ben Adida. Helios: Web-based open-audit voting. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1496711.1496734.

Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Vladimir Klebanov, and Peter H. Schmitt, editors. *The KeY Book 2*. LNCS. Springer. to appear.

Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *VSTTE 2014: 6th Working Conference on Verified Software: Theories, Tools, Experiments*, Lecture Notes in Computer Science. Springer International Publishing, 2014. To appear.

Mihhail Aizatulin, François Dupressoir, AndrewD. Gordon, and Jan Jürjens. Verifying cryptographic code in c: Some experience and the csec challenge. In Gilles Barthe, Anupam Datta, and Sandro Etalle, editors, *Formal Aspects of Security and Trust*, volume 7140 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-29419-8. doi: 10.1007/978-3-642-29420-4_1. URL http://dx.doi.org/10.1007/978-3-642-29420-4_1.

Torben Amtoft and Anindya Banerjee. Verification condition generation for conditional information flow. In *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, FMSE '07, pages 2–11, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-887-9. doi: 10.1145/1314436.1314438. URL http://doi.acm.org/10.1145/1314436.1314438.

Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Proceedings POPL*, pages 91–102. ACM, 2006.

Torben Amtoft, John Hatcliff, Edwin Rodríguez, Robby, Jonathan Hoag, and David Greve. Specification and checking of software contracts for conditional information flow. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors,

FM 2008: Formal Methods, volume 5014 of *Lecture Notes in Computer Science*, pages 229–245. Springer-Verlag, 2008. ISBN 978-3-540-68235-6.

Torben Amtoft, John Hatcliff, and Edwin Rodríguez. Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. In Andrew Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 43–63. Springer-Verlag, 2010. ISBN 978-3-642-11956-9.

Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *In IEEE Computer Security Foundations Workshop (CSFW*, pages 253–267. IEEE Computer Society Press, 2002.

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 339–353, Los Alamitos, CA, USA, May 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. doi: http://doi.ieeecomputersociety.org/10.1109/SP.2008.20.

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, CSFW '04, pages 100–115, Washington, USA, 2004. IEEE CS. ISBN 0-7695-2169-X. doi: 10.1109/CSFW.2004.17. URL `http://dx.doi.org/10.1109/CSFW.2004.17`.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer-Verlag, 2011.

Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference Java bytecode verifier. *Mathematical Structures in Comp. Sci.*, FirstView:1–50, 4 2013. ISSN 1469-8072. doi: 10.1017/S0960129512000850. URL `http://journals.cambridge.org/article_S0960129512000850`.

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. LNCS. Springer Berlin Heidelberg, 2007b. ISBN 3-540-68977-X, 978-3-540-68977-5.

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Information flow in object-oriented software. In Gopal Gupta, editor, *Logic-Based Program Synthesis and Transformation, LOPSTR'13*. Springer Berlin Heidelberg, 2014. To appear.

L. Beringer and M. Hofmann. Secure information flow and program logics. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 233–248, July 2007. doi: 10.1109/CSF.2007.30.

Karthikeyan Bhargavan, Ricardo Corin, Cedric Fournet, and Eugen Zalinescu. Cryptographically verified implementations for tls. In *15th ACM Conference on Computer and Communications Security (CCS'08)*. Association for Computing Machinery, Inc., October 2008. URL http://research.microsoft.com/apps/pubs/default.aspx?id=79576.

Jens-Matthias Bohli, Jörn Müller-Quade, and Stefan Röhrich. Bingo voting: Secure and coercion-free voting using a trusted random number generator. In Ammar Alkassar and Melanie Volkamer, editors, *E-Voting and Identity*, volume 4896 of *Lecture Notes in Computer Science*, pages 111–124. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-77492-1. doi: 10.1007/978-3-540-77493-8_10. URL http://dx.doi.org/10.1007/978-3-540-77493-8_10.

Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen, and Mattias Ulbrich. The COST IC0701 verification competition 2011. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software*, volume 7421 of *Lecture Notes in Computer Science*, pages 3–21. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31761-3. doi: 10.1007/978-3-642-31762-0_2. URL http://dx.doi.org/10.1007/978-3-642-31762-0_2.

AaronR. Bradley, Zohar Manna, and HennyB. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-31139-3. doi: 10.1007/11609773_28. URL http://dx.doi.org/10.1007/11609773_28.

Daniel Bruns. Formal verification of an electronic voting system. Technical Report 2014-11, Department of Informatics, Karlsruhe Institute of Technology, 2014. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000042284.

Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract interpretation of symbolic execution with explicit state updates. In de Boer et al. [2009], pages 247–277. ISBN 978-3-642-04166-2.

D. Chaum, R.T. Carback, J. Clark, A. Essex, Stefan Popoveniuc, R.L. Rivest, P. Y A Ryan, E. Shen, A.T. Sherman, and P.L. Vora. Scantegrity ii: End-to-end verifiability by voters of optical scan elections through confirmation codes.

Information Forensics and Security, IEEE Transactions on, 4(4):611–627, Dec 2009. ISSN 1556-6013. doi: 10.1109/TIFS.2009.2034919.

M.R. Clarkson, S. Chong, and A.C. Myers. Civitas: Toward a secure voting system. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 354–368, May 2008. doi: 10.1109/SP.2008.32.

Ellis S. Cohen. Information transmission in computational systems. In *SOSP*, pages 133–139, 1977.

Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005.

Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine, editors. *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*, volume 5751 of *LNCS*, 2009. Springer-Verlag. ISBN 978-3-642-04166-2.

Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977. ISSN 0001-0782. doi: 10.1145/359636.359712. URL `http://doi.acm.org/10.1145/359636.359712`.

Guillaume Dufay, Amy Felty, and Stan Matwin. Privacy-sensitive information flow with JML. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 738–738. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-28005-7. URL `http://dx.doi.org/10.1007/11532231_9`. 10.1007/11532231_9.

Jürgen Geisler and Christoph Scheben. Human processor modelling language (HPML): Estimate working memory load trough interaction. In *Analysis, Design, and Evaluation of Human-Machine Systems*, volume 10, pages 95–100, 2007.

Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

Christian Haack, Erik Poll, and Aleksy Schubert. Explicit information flow properties in JML. In *3rd Benelux Workshop on Information and System Security (WISSec)*, Nov 2008.

Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*, pages

87–96. IEEE, March 2006. URL http://pp.info.uni-karlsruhe.de/uploads/publikationen/hammer06issse.pdf.

R. R. Hansen and C. W. Probst. Non-interference and erasure policies for java card bytecode. In *6th International Workshop on Issues in the Theory of Security (WITS '06)*. 2006. URL http://www2.imm.dtu.dk/pubdb/p.php?4742.

David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0262082896.

Chris Hawblitzel, Ming Kawaguchi, ShuvenduK. Lahiri, and Henrique Rebêlo. Towards modularly comparing programs using automated theorem provers. In MariaPaola Bonacina, editor, *Automated Deduction – CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 282–299. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38573-5. doi: 10.1007/978-3-642-38574-2_20. URL http://dx.doi.org/10.1007/978-3-642-38574-2_20.

Daniel Hedin and David Sands. Timing aware information flow security for a JavaCard-like bytecode. In *BYTECODE*, volume 141:1 of *ENTCS*, pages 163 – 182. Elsevier, 2005.

Daniel Hedin and David Sands. Noninterference in the presence of non-opaque pointers. In *CSFW*, pages 217–229, 2006.

I. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23: 267–288, 2011. ISSN 0934-5043. URL http://dx.doi.org/10.1007/s00165-010-0152-5. 10.1007/s00165-010-0152-5.

Matthias Kuntz, Stefan Leue, and Christoph Scheben. Extending non-termination proof techniques to asynchronously communicating concurrent programs. In Andrei Voronkov, Laura Kovacs, and Nikolaj Bjorner, editors, *WING 2010*, volume 1 of *EPiC Series*, pages 132–147. EasyChair, 2012.

Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P)*, pages 538–553, Oakland, California, USA, 2011. IEEE Computer Society.

Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Jürgen Graf, and Christoph Scheben. A hybrid approach for proving noninterference and applications to the cryptographic verification of Java programs. 2013.

R. Küsters, T. Truderung, and J. Graf. A framework for the cryptographic verification of java-like programs. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 198–212, June 2012. doi: 10.1109/CSF.2012.9.

Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16 (10):613–615, 1973.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, May 2006. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1127878.1127884. URL http://doi.acm.org/10.1145/1127878.1127884.

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. Jml reference manual, 2008.

Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201432943.

J. Mccarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

Bertrand Meyer. *Object-Oriented Software Construction, 1st editon*. Prentice-Hall, 1988. ISBN 0-13-629031-0.

John C. Mitchell. Type systems for programming languages. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 365–458. 1990.

Andrew C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 228–241, New York, NY, USA, 1999a. ACM. ISBN 1-58113-095-3. doi: http://doi.acm.org/10.1145/292540.292561. URL http://doi.acm.org/10.1145/292540.292561.

Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, 1999b.

Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 2000. ISSN 1049-331X. doi: 10.1145/363516.363526. URL http://doi.acm.org/10.1145/363516.363526.

A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 165 –179, may 2011. doi: 10.1109/SP.2011.12.

David Naumann. From coupling relations to mated invariants for checking information flow. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *Computer Security – ESORICS 2006*, volume 4189 of *Lecture Notes in Computer Science*, pages 279–296. Springer-Verlag, 2006. ISBN 978-3-540-44601-9.

Pavel Nikolov. Combining theorem proving and type systems for precise and efficient information flow verification. Bachelor's thesis (Studienarbeit), Karlsruhe Institute of Technology, 2014.

Jing Pan. A theorem proving approach to analysis of secure information flow using data abstraction. Master's thesis, Dept. of Computer Science and Engineering, Chalmers University of Technology, 2005.

Quoc-Sang Phan. Self-composition by symbolic execution. In *Imperial College Computing Student Workshop (ICCSW'13)*, pages 95–102. Schloss Dagstuhl, 2013.

Arun D. Raghavan and Gary T. Leavens. Desugaring JML Method Specifications. Technical Report 00-03a, 2000. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.9308.

Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 4246 of *Lecture Notes in Computer Science*, pages 422–436. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-48281-9. doi: 10.1007/11916277_29. URL http://dx.doi.org/10.1007/11916277_29.

Andrei Sabelfeld and Andrew Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-23635-1. URL http://dx.doi.org/10.1007/978-3-540-37621-7_9. 10.1007/978-3-540-37621-7_9.

Christoph Scheben. Simulation of d'-dimensional cellular automata on d-dimensional cellular automata. In Samira Yacoubi, Bastien Chopard, and Stefania Bandini, editors, *Cellular Automata*, volume 4173 of *Lecture Notes in Computer Science*, pages 131–140. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-40929-8. doi: 10.1007/11861201_18. URL http://dx.doi.org/10.1007/11861201_18.

Christoph Scheben and Peter H. Schmitt. Verification of information flow properties of Java programs without approximations. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software*, volume 7421 of *Lecture Notes in Computer Science*, pages 232–249. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31761-3. doi: 10.1007/978-3-642-31762-0_15. URL http://dx.doi.org/10.1007/978-3-642-31762-0_15.

Christoph Scheben and Peter H. Schmitt. Efficient self-composition for weakest precondition calculi. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 579–594. Springer International Publishing, 2014. ISBN 978-3-319-06409-3. doi: 10.1007/978-3-319-06410-9_39. URL http://dx.doi.org/10.1007/978-3-319-06410-9_39.

Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in Java dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Revised Selected Papers, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, 2011.

Bart van Delft. Abstraction, objects and information flow analysis. Master's thesis, Institute for Computing and Information Science, Radboud University Nijmegen, 2011.

Simon Wacker. Blockverträge. Bachelor's thesis (Studienarbeit), Karlsruhe Institute of Technology, 2012.

Martijn Warnier. *Language Based Security for Java and JML*. PhD thesis, Radboud University Nijmegen, 2006.

Benjamin Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.

Index