Vladimir Klebanov

Extending the Reach and Power of Deductive Program Verification

Dissertation

Koblenz, June 2009

Department of Computer Science
Universität Koblenz-Landau

Vladimir Klebanov

Extending the Reach and Power of Deductive Program Verification

Vom Promotionsausschuss des Fachbereichs 4: Informatik der Universität Koblenz-Landau zur Verleihung des akademischen Grades Doktor der Naturwissenschaften (Dr. rer. nat.) genehmigte Dissertation.

Die wissenschaftliche Aussprache fand am 23. Juli 2009 statt.

Vorsitzender des Promotionsausschusses:	Prof. Dr. Dieter Zöbel
Vorsitzender der Promotionskommission:	Prof. Dr. Lutz Priese
Berichterstatter:	Prof. Dr. Bernhard Beckert
	Prof. Dr. Einar Broch Johnsen

Acknowledgements

Most of all, I want to thank my *Doktorvater* and mentor Prof. Bernhard Beckert for his invaluable scientific guidance as well as for his patience, his humor, for being a role model, and for other things that are too numerous to mention. I have never learned more from anybody else in my life.

I'm thankful to Prof. Einar Broch Johnsen for being the second reviewer of this thesis and for inviting me to eat snake in my first PhD year.

I'm thankful to the (other) founders of the KeY project: to Prof. P. H. Schmitt for his support from the very early days of my studies on and for giving me the opportunity to be part of the KeY project; to Prof. Reiner Hähnle for his support and for being a wonderful host for me in Sweden; to Prof. Menzel for his philosophical insights.

I'm much obliged to Dr. Matthias Zenger for upholding my faith in computer science when I first started to study it in Karlsruhe.

My special gratitude goes to Prof. Furbach for being an inspiring scientist, a charismatic group leader, and for always showing me the bigger perspective.

It was a pleasure for me to work on the KeY project, and I'm thankful to all its members, past and present, and in particular to Prof. Wolfgang Ahrendt, Dr. Thomas Baar, Dr. Richard Bubel, Dr. Martin Giese, Dr. Wojciech Mostowski, Dr. Andreas Roth, Dr. Philipp Rümmer, Dr. Steffen Schlager, and last but not least, Angela Wallenburg.

The same applies to my colleagues at the AGKI group in Koblenz, and in particular to Dr. Oliver Obst, Dr. Gerd Beuster, Claudia Obermaier and Thorsten Bormer.

I'm thankful to all the students who worked with me and in particular to Volker Klasen for helping with the calculus implementation.

I extend my thanks to my friends in Koblenz and elsewhere and to my family.

I gratefully acknowledge the support of my work by the following European funding agencies: Deutsche Forschungsgemeinschaft (DFG), Deutscher Akademischer Austauschdienst (DAAD), European Cooperation in Science and Technology (COST), and, of course, the University of Koblenz-Landau.

Koblenz, June 2009 *Vladimir Klebanov*

Abstract

Software is vital for modern society. The efficient development of correct and reliable software is of ever-growing importance. An important technique to achieve this goal is deductive program verification: the construction of logical proofs that programs are correct.

In this thesis, we address three important challenges for deductive verification on its way to a wider deployment in the industry:

1. verification of thread-based concurrent programs
2. correctness management of verification systems
3. change management in the verification process.

These are consistently brought up by practitioners when applying otherwise mature verification systems. The three challenges correspond to the three parts of this thesis (not counting the introductory first part, providing technical background on the KeY verification approach).

In the first part, we define a novel program logic for specifying correctness properties of object-oriented programs with unbounded thread-based concurrency. We also present a calculus for the above logic, which allows verifying actual Java programs. The calculus is based on symbolic execution resulting in its good understandability for the user. We describe the implementation of the calculus in the KeY verification system and present a case study.

In the second part, we provide a first systematic survey and appraisal of factors involved in reliability of formal reasoning. We elucidate the potential and limitations of self-application of formal methods in this area and give recommendations based on our experience in design and operation of verification systems.

In the third part, we show how the technique of similarity-based proof reuse can be applied to the problems of industrial verification life cycle. We address issues (e.g., coping with changes in the proof system) that are important in verification practice, but have been neglected by research so far.

Contents

List of Tables

List of Figures

1

Setting the Stage

1.1 What Software Verification Needs in Practice

Software is vital for modern society. The efficient development of correct and reliable software is of ever-growing importance. An important technique to achieve this goal is deductive program verification: the construction of logical proofs that programs are correct. Logic-based technologies for the formal description, construction, analysis, and validation of software can be expected to complement and partly replace traditional software engineering methods in the future.

Already, program verification methods have outgrown the area of academic case studies, and industry is showing serious interest. While the basic ideas of software verification have been known for a long time, research is still needed in order to achieve reach and power to assure reliability of object-oriented programs in the industrial setting.

This thesis presents work towards this goal. In particular, we address three important challenges for deductive verification on its way to a wider deployment in the industry:

1. verification of thread-based concurrent programs
2. correctness management of verification systems
3. change management in the verification process.

While there are numerous other challenges in the field (including software and proof modularization, efficient deduction, proof visualization and counterexample generation), these are the ones consistently brought up by practitioners when applying otherwise mature verification systems in the industry.

Industrial software practice differs from academic practice in one central aspect: verification practitioners do not have full control of the setting. In particular, they cannot dictate the choice of programming language, programming environment (hardware and software) and programming style. Industrial development is characterized by presence of development cycles, third-party dependencies, and the fact that the final product indeed may be deployed, often on a variety of different platforms.

We now describe the individual challenges in more detail.

Verification of multi-threaded programs

Academic research has mostly postponed working on deductive verification of object-oriented thread-based concurrent programs. At this point, further postponing is no longer appropriate. Concurrency is probably *the* single aspect of programming practice that has gained most importance lately. On the hardware side this trend is fueled by proliferation of multi-core processors and embedded platforms supporting multi-threading. On the software side, most modern applications concerned with networking, graphical user interfaces or resource control have concurrent aspects. Indeed, formal method practitioners now often claim that "all [their] problems involve concurrency" [Cook, 2007].

Correctness management

Correctness is never an absolute. Simply applying deductive verification does not automatically lead to a system that never fails. Instead, correctness has to be managed. Not properly understanding the issues involved leads to wasted resources at best or a false sense of security at worst. This is particularly true in an environment involving third-party products, such as hardware, compilers, libraries, etc.

Issues that research has to address in this field are:

- to show how verification works together with other engineering techniques for achieving high assurance with the minimal use of resources
- to develop, evaluate and compare methods for assuring correctness of verification calculi and their implementations
- to clarify the guarantees that formal methods provide (or not provide) and their respective assumptions.

Management of change

All components in the industrial verification process—programs, specifications, and also the verification systems themselves—undergo constant change. Correctness assurances (in our case proofs) constructed with a lot of effort quickly become obsolete during the development or maintenance cycle. The current state of affairs in the field resembles programming prior to the invention of modern version control systems. Formal methods practitioners at NASA state that re-certification costs—with deductive verification contributing a big part—are the biggest bottleneck in construction of dependable software [Denney and Fischer, 2005].

Proper change management is crucial for cost-effective production of verified software. This involves keeping track of correctness assertions as their constituents evolve (dependency management) and salvaging efforts invested into construction of their previous versions (proof reuse).

1.2 Contributions

The narrator pronoun "we" in this thesis already refers to *pluralis auctoris*, the author's plural. Still, to avoid ambiguity, I use the singular form in this section.

Contribution Summary

The main contributions of this thesis are:

- a novel program logic for an object-oriented language with unbounded thread-based concurrency
- an implemented calculus for full functional verification of a substantial fragment of multi-threaded JAVA programs
- a first implemented calculus for proving JAVA Memory Model-safety of programs that takes both locking and volatile variable synchronization into account
- a first comprehensive analysis of the factors involved in reliability of formal reasoning in large domain-specific theories and the methods to achieve it
- a novel application of a proof reuse technique developed by me earlier to solve several important problems in the verification life cycle.

Many of the results presented in this thesis have already been published in workshop and conference proceedings, or as book chapters. These publications are referred to in the following, and their full list is included on page 157.

Chapter Breakdown

Part I

Chapter 2 provides standard introductory material on the KeY approach. The content of this chapter is based to a large extent on the Chapter "Dynamic Logic" in the KeY book [Beckert et al., 2007b], which I have co-authored. I have reworked, simplified and compressed the presentation, though, abstracting from details unimportant here. This chapter also incorporates material I have co-written for [Beckert et al., 2007a].

Part II

Chapter 3 gives a quick first glance at the proposed logic and proof system for multi-threaded JAVA programs. I discuss which features of JAVA concurrency are supported and survey related work.

Chapter 4 defines a novel logic (syntax and model-theoretic semantics) for an object-oriented language with unbounded thread-based concurrency. The logic has good "understandability" as it is close to the programmer's view of the language. On the semantical side, I show—surprisingly maybe—how scheduling non-determinism can be modeled adequately by an underspecified deterministic scheduler. An early form of the logic has been published in [Beckert and Klebanov, 2007a].

Chapter 5 refines the basic version of the logic with a more verification-friendly scheduler formulation. For this, I have taken the notion of symmetry reduction that is well-established in model checking and extended it significantly for the use in deductive verification. The refined scheduler model avoids explicit thread enumeration or total ordering of many independent events, thus making reasoning about unbounded multi-threading efficient.

Chapter 6 presents a calculus for verification of actual concurrent Java programs. The calculus is based on symbolic execution, which is—to our knowledge—the first application of this technique to verification of multi-threaded programs. The advantage of symbolic execution ("forward reasoning") lies in its good understandability for the user. The calculus has been published in [Beckert and Klebanov, 2007a].

Chapter 7 includes several extensions and refinements of the basic calculus. One is an experimental extension to verify programs with condition variables (revising [Beckert and Klebanov, 2007b]). Another is the first implemented synchronization-complete calculus for establishing safety of real Java programs w.r.t. the Java Memory Model. Other verification systems available to date either ignore the issue altogether or are incomplete in this regard (do not consider synchronization edges based on volatile variables). An early precursor of this work has been published in [Klebanov, 2004]. The chapter concludes with a discussion of future work.

Chapter 8 describes the implementation of the calculus in the KeY system and presents case studies.

Part III

Chapter 9 provides a first pragmatic survey and appraisal of all the factors involved in reliability of formal reasoning. Experience shows that the field suffers from ambiguous terminology, misconceptions, and methodical bias. Though the individual issues raised may be known to many experts, there has been so far no unified view on the problem.

I provide such a view and, in particular, elucidate the potential and limitations of self-application of formal methods. I summarize the situation in the KeY project and give recommendations based on my experience in design of verification systems.

An early version of this work has been published as [Beckert and Klebanov, 2006].

Part IV

Chapter 10 presents a framework for similarity-based proof reuse in deductive software verification. While the technique itself, which I have developed earlier (but include here for completeness), is not part of this thesis' contribution—its novel applications in the industrial verification life cycle are. I show how the technique can be used to make the life of verification engineer easier. I address the issues (e.g., coping with changes in the proof system) that are important in verification practice, but have been neglected by research so far.

The new results in this chapter have been published in [Beckert and Klebanov, 2004; Beckert et al., 2005] and in the Chapter "Proof Reuse" of the KeY book [Klebanov, 2007] (which I authored).

1.3 Typographic Conventions

In this text we use a number of typesetting conventions. Concrete expressions from programming languages are written in `typewriter` font. Mathematical meta symbols are set in *math* font. Names of calculus rules are set in sans serif font.

Part I

Background

2

What You Need to Know about KeY

This chapter provides the context for the work documented in this thesis and makes this text mostly self-contained. Details mentioned as irrelevant can be found in the KeY book [Beckert et al., 2007]. We still hope that even without this primer the subsequent chapters are understandable for a reader with a background in formal methods and a superficial knowledge of the KeY approach.

First, we give a brief overview of the KeY system followed by a not-so-brief technical background. It begins with a discussion of the main concepts of the JAVA CARD Dynamic Logic of KeY. The syntax and semantics of the logic are formally defined in Sections 2.3 and 2.4. Finally, in Section 2.5–2.10, we present the JAVA CARD DL calculus, which KeY uses for verification of JAVA CARD programs.

2.1 KeY is a Verification System for JAVA

The KeY system is the main software product of the KeY project, a joint effort between the University of Karlsruhe, Chalmers University of Technology in Göteborg, and the University of Koblenz. The KeY system is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. At the core of the system is a deductive verification component that implements a free-variable sequent calculus for first-order Dynamic Logic for JAVA.

The architecture of the KeY system is shown in Fig. 2.1. Optional plugins to the popular Eclipse IDE and to the Borland Together CASE tool suite have been developed to lower the entry hurdle for users with no or little training in formal methods. KeY supports several languages for specifying properties of object-oriented models. Many people working with UML or model-driven development have familiarity with the specification language OCL (Object Constraint Language), a part of UML 2.0. Another supported specification language, which enjoys popularity among JAVA developers, is JML (Java Modeling Language). KeY can also translate OCL expressions to natural language (English and German).

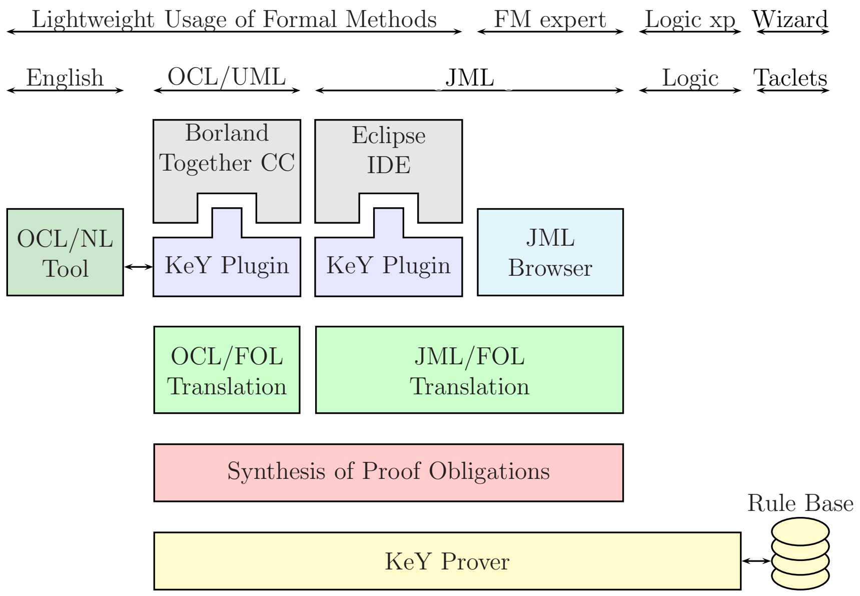


Figure 2.1. Architecture and interfaces of the KeY system

The target programming language for verification in KeY is Java Card 2.2.1. KeY is the only publicly available verification tool that supports the full Java Card standard including the persistent/transient memory model of the card devices and the atomic transactions. Rich specifications of the Java Card API are available both in OCL and JML. Java 1.4 programs that respect the limitations of Java Card (no floats, no reflection, no dynamic class loading) can be verified as well. Verification of (restricted) multi-threaded programs has become possible with this work.

The system is not a classical verification condition generator (VCG), but a theorem prover for program logic that combines a variety of automated reasoning techniques. The KeY prover is distinguished from most other deductive verification systems in that symbolic execution of programs, first-order reasoning, arithmetic simplification, external decision procedure calls, and symbolic state simplification are interleaved. Symbolic execution is typically carried out in a fully automated manner as long as loops are bounded or an invariant is available.

While we constantly strive to increase the overall degree of automation, user interaction remains indispensable in deductive program verification. The main design goal of the KeY prover is thus a seamless integration of automated and interactive proving. Efficiency must be measured in terms of user plus prover, not just prover alone. Accordingly, the strong point of KeY is a combination of a good user interface for proof state presentation and rule application, a high level of automation, extensibility of the rule base, and a calculus without backtracking.

KeY itself is made up of ca. 124,000 lines[1] of Java code. The standard rule base consists of 1,725 rules that are written in about 15,000 lines of KeY's "taclet" rule description language. About 1,300 of these formalize the semantics of the Java programming language. The system has been created by 14 implementors since 1999, who spent a total of about 30 person years. Version 1.0 of the KeY system has been released in connection with the KeY book [Beckert et al., 2007]; current version is KeY 1.4. The KeY tool is available under GPL and can be downloaded from `www.key-project.org`.

2.2 Foundations of Dynamic Logic

The logical basis of the KeY system's software verification component is an instance of Dynamic Logic (DL) [Harel, 1984; Beckert, 2001]. The principle of DL is the formulation of statements about program behavior by integrating programs and formulas within a single language. Such a language is constructed by extending some non-dynamic logic with parameterized modal operators $\langle p \rangle$ and $[p]$ for every legal program p of some programming language. In our case, the non-dynamic base logic is typed first-order predicate logic, and the programming language is Java Card. The programs p within the modal operators are Java Card statements, and the logic of KeY is called Java Card Dynamic Logic or, for short, Java Card DL.

The operators (modalities) $\langle p \rangle$ and $[p]$ refer to the final state of p and can be placed in front of any formula. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds, while $[p]\phi$ does not demand termination and expresses that *if* p terminates, then ϕ holds in the final state. For example, "when started in a state where `x` is zero, `x++;` terminates in a state where `x` is one" can in DL be expressed as $\mathtt{x} = 0 \longrightarrow \langle \mathtt{x{+}{+}} \rangle (\mathtt{x} = 1)$.

Presence of modalities raises an important semantical issue of *program determinism*. Determinism here means that a program, for the same initial state resp. the same inputs, always has the same behavior—in particular, the same final state (if it terminates) resp. the same outputs. When we do not (exactly) know what the initial state resp. the inputs are, we may not know what (exactly) the behavior is, but it is still deterministic. In particular, we do not consider unknown inputs as a source of non-determinism.

In contrast, there can be more than one final state if the programming language contains non-deterministic constructs and a program uses them. The Java Card language is sequential and deterministic, and there is exactly one final state (if p terminates normally, i.e., does not terminate abruptly due to an uncaught exception) or there is no such state (if p does not terminate or terminates abruptly). We will discuss determinism of multi-threaded programs later in Chapter 3.

Deduction in DL, and in particular in Java Card DL is based on symbolic program execution and simple program transformations (⇒ Sect. 2.5.5) and is, thus, close to a programmer's understanding of Java.

[1] Not counting comments. These numbers are based on our estimates and the results of the SLOCCount tool (`www.dwheeler.com/sloccount`).

2.3 Syntax of JAVA CARD DL

We start with the definition of the underlying type hierarchies and the signatures of JAVA CARD DL. Then, we define the syntax of JAVA CARD DL, which consists of terms, formulas, and a new category of expressions called *updates*.

2.3.1 Type Hierarchy

The type system of the KeY logic is designed to match the JAVA type system. In JAVA, there are two type concepts that should not be confused:

1. Every object created during the execution of a JAVA program has a *dynamic type*. If an object is created with the expression `new C(...)`, then `C` is the dynamic type of the newly created object. The dynamic type of an object is fixed from its creation until it is garbage collected. The dynamic type of an object can never be an interface type or an abstract class type.
2. Every expression occurring in a JAVA program has a *static type*. The dynamic type of an object that results from evaluating an expression is always a subtype of the static type of that expression. In contrast to dynamic types, static types can also be abstract class types or interface types.

This distinction is reflected in the logic by assigning static types to expressions (terms) and dynamic types to their values (domain elements). The logic also includes *type casts* (changing the static type of a term) and *type predicates* (checking the dynamic type of a term) in order to reason about inheritance and polymorphism. These operators are not important for this work, though.

The notion of a *type hierarchy*, groups all the relevant information about types and their subtyping relationships.

Definition 2.1. A *type hierarchy* is a quadruple $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ of

- a finite set of *static types* $\mathcal{T}$,
- a finite set of *dynamic types* $\mathcal{T}_d$,
- a finite set of *abstract types* $\mathcal{T}_a$, and
- a *subtype relation* $\sqsubseteq$ on $\mathcal{T}$,

such that

- $\mathcal{T} = \mathcal{T}_d \mathbin{\dot{\cup}} \mathcal{T}_a$
- There is an *empty type* $\bot \in \mathcal{T}_a$ and a *universal type* $\top \in \mathcal{T}_d$.
- $\sqsubseteq$ is a reflexive partial order on $\mathcal{T}$,
- $\bot \sqsubseteq A \sqsubseteq \top$ for all $A \in \mathcal{T}$.
- $\mathcal{T}$ is closed under greatest lower bounds w.r.t. $\sqsubseteq$. We write $A \sqcap B$ for the greatest lower bound of A and B and call it the *intersection type* of A and B. The existence of $A \sqcap B$ also guarantees the existence of the least upper bound $A \sqcup B$ of A and B, called the *union type* of A and B.
- Every non-empty abstract type $A \in \mathcal{T}_a \setminus \{\bot\}$ has a non-abstract subtype: $B \in \mathcal{T}_d$ with $B \sqsubseteq A$.

We say that A *is a subtype of* B if $A \sqsubseteq B$. The set of non-empty static types is denoted by $\mathcal{T}_q = \mathcal{T} \setminus \{\perp\}$. $\lhd$

Note 2.2. We are, of course, only interested in type hierarchies that are "useful". The KeY system automatically populates the type hierarchy with the types relevant for the program(s) being verified. For instance, it is ensured that:

1. $A \in \mathcal{T}_a$ for all interface and abstract class types A declared in or imported into p.
2. $A \in \mathcal{T}_d$ for all non-abstract class types A declared in or imported into p.
3. $C \sqsubseteq D$ iff C is implicitly or explicitly declared as a subtype of D (using the keywords `extends` or `implements`), for all (abstract) class or interface types C, D declared in or imported into p.
4. the type hierarchies contain appropriate array types.

The type hierarchies also always contain the types such as boolean, the root reference type Object, and the type Null, which is a subtype of all reference types. Finally, they contain several integer types, including both the range-limited types of Java and the infinite integer type $\mathbb{Z}$. $\lhd$

Most of the notions defined in the remainder of this chapter depend on some type hierarchy. In order to avoid cluttering the notation, we assume that a certain fixed type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ is given, to which all later definitions refer.

2.3.2 Signature

We now define the set of symbols that the language Java Card DL consists of. In contrast to first-order signatures, we have two kinds of function and predicate symbols: *rigid* and *non-rigid* symbols. Consequently, the set of function symbols is divided into two disjoint subsets FSym_r and FSym_{nr} of rigid and non-rigid functions, respectively (the same applies to the set of predicate symbols). Intuitively, rigid symbols have the same meaning in all program states (e.g., the addition on integers or the equality predicate), whereas the meaning of non-rigid symbols may differ from state to state. Non-rigid symbols are used to model *program variables* such as (local) variables, attributes, and arrays outside of modalities; i.e., non-rigid symbols represent program variables as terms in the logic. Program variables can thus not be bound by quantifiers—in contrast to logical variables. Note that in classical DL there is no distinction between logical variables and program variables (non-rigid constants).

Definition 2.3 (Java Card DL signature). A *Java Card DL signature* (for a given type hierarchy) is a tuple

$$\Sigma = (\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$$

consisting of

- a set VSym of variables
- disjoint sets FSym_r of *rigid* function symbols and FSym_{nr} of *non-rigid* function symbols such that together $\mathrm{FSym} = \mathrm{FSym}_r \dot\cup \mathrm{FSym}_{nr}$

- disjoint sets PSym_r of *rigid* predicate symbols and PSym_{nr} of *non-rigid* predicate symbols such that together $\mathrm{PSym} = \mathrm{PSym}_r \dot{\cup} \mathrm{PSym}_{nr}$
- a typing function α,

such that[2]

- $\alpha(v) \in \mathcal{T}_q$ for all $v \in \mathrm{VSym}$,
- $\alpha(f) \in \mathcal{T}_q^* \times \mathcal{T}_q$ for all $f \in \mathrm{FSym}$, and
- $\alpha(p) \in \mathcal{T}_q^*$ for all $p \in \mathrm{PSym}$.
- There is a function symbol $(A) \in \mathrm{FSym}$ with $\alpha((A)) = ((\top), A)$ for any $A \in \mathcal{T}_q$, called the *cast to type A*.
- There is a predicate symbol $\doteq \in \mathrm{PSym}$ with $\alpha(\doteq) = (\top, \top)$.
- There is a predicate symbol $\sqsubseteq\!A \in \mathrm{PSym}$ with $\alpha(\sqsubseteq\!A) = (\top)$ for any $A \in \mathcal{T}$, called the *type predicate for type A*.

We use the following notations:

- $v{:}A$ for $\alpha(v) = A$,
- $f{:}A_1, \ldots, A_n \to A$ for $\alpha(f) = ((A_1, \ldots, A_n), A)$, and
- $p{:}A_1, \ldots, A_n$ for $\alpha(p) = (A_1, \ldots, A_n)$.

A *constant symbol* is a function symbol c with $\alpha(c) = ((), A)$ for some type A. $\lhd$

Note 2.4 (Symbols contained in the signature). To have a logic useful vor verification, we expect that the signature always contains certain *predefined* symbols. These are typically operators of common data types. For instance, we require that a Java Card DL signature contains constants $0, 1, \ldots$ representing the integer numbers, function symbols for arithmetical operations (addition, subtraction, etc.), and the typical ordering predicates on integers. A full list of predefined symbols for Java Card DL is given in [Beckert et al., 2007].

Furthermore, the KeY system automatically populates the signature with the symbols needed to model program variables in a given program p:

1. The predefined non-rigid array access function symbol $[\,] : (\top, \mathbb{Z} \to \top) \in \mathrm{FSym}_{nr}$.
2. For all local variables and static field declarations "$A\ id;$" in p:
   a) If A is not an array type, then $id{:}A \in \mathrm{FSym}_{nr}$.
   b) If $A = A'[\,]^n$ is an array type, then $id{:}(A'[\,]^n) \in \mathrm{FSym}_{nr}$.
3. For all non-static field declarations "$A\ id;$" in a class C in p:
   a) If A is not an array type, then $id{:}\,(C \to A) \in \mathrm{FSym}_{nr}$.
   b) If $A = A'[\,]^n$ is an array type, then $id{:}\,(C \to A'[\,]^n) \in \mathrm{FSym}_{nr}$. $\lhd$

In contrast to first-order logic, the definition of terms and formulas (and also updates) in Java Card DL cannot be done separately, since their definitions are mutually recursive. For example, a formula may contain terms, which may contain updates. Updates in turn may contain formulas ("quantified updates"). Nevertheless, in order to improve readability we give separate definitions of updates, terms, and formulas in

[2] We use the standard notation A^* to denote the set of (possibly empty) *sequences* of elements of A.

the following. Also, in order to avoid cluttering the notation, we assume that a certain fixed signature $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ w.r.t. a type hierarchy is given, to which later definitions of this chapter refer.

2.3.3 Syntax of JAVA CARD DL Terms

Definition 2.5 (Terms of JAVA CARD DL). The system $\{Terms_A\}_{A\in\mathcal{T}}$ of sets of *terms of static type A* is inductively defined as the smallest system of sets such that:

- $x \in Terms_A$ for all variables $x{:}A \in \mathrm{VSym}$;
- $f(t_1, \ldots, t_n) \in Terms_A$ for all function symbols $f{:}A_1, \ldots, A_n \to A$ in FSym and terms $t_i \in Terms_{A'_i}$ with $A'_i \sqsubseteq A_i$ $(1 \le i \le n)$;
- $(\text{if } \phi \text{ then } t_1 \text{ else } t_2) \in Terms_A$ for all $\phi \in Formulas$ (⇒ Def. 2.8) and all terms $t_1 \in Terms_{A_1}$, $t_2 \in Terms_{A_2}$ with $A = A_1 \sqcup A_2$;
- $\{u\}\, t \in Terms_A$ for all updates $u \in Updates$ (⇒ Def. 2.6) and all terms $t \in Terms_A$.

In the style of JAVA CARD syntax we often write $t.f$ instead of $f(t)$ and $a[i]$ instead of $[\,](a, i)$.[3] ◁

Terms in JAVA CARD DL play the same role as in first-order logic, i.e., they denote elements of the domain. The syntactical difference to first-order logic is the existence of terms of the form $(\text{if } \phi \text{ then } t_1 \text{ else } t_2)$ (which could be defined for first-order logic as well). Informally, if ϕ holds, a conditional term $(\text{if } \phi \text{ then } t_1 \text{ else } t_2)$ denotes the domain element t_1 evaluates to. Otherwise, if ϕ does not hold, t_2 is evaluated.

Terms can be prefixed by updates, which we define next.

2.3.4 Syntax of JAVA CARD DL Updates

We now introduce an additional syntactic category called *updates* [Beckert, 2001]. Syntactic updates can be seen as a language for describing state transitions. Evaluating $\{loc := val\}\,\phi$ in some state is equivalent to evaluating ϕ in a modified state where *loc* evaluates to *val*. The difference between updates and assignments is that the syntax of updates is quite restricted, making analysis and simplification of state change effects easier and efficient. Updates (together with case distinctions) can be seen as a normal form for programs and, indeed, the idea of our calculus is to stepwise transform a program to be verified into a sequence of updates, which are then simplified and applied to first-order formulas.

Definition 2.6 (Syntactic updates of JAVA CARD DL). The set *Updates* of *syntactic updates* is inductively defined as the smallest set such that:

Function update $(f(t_1, \ldots, t_n) := t) \in Updates$ if $f(t_1, \ldots, t_n) \in Terms_A$ (⇒ Def. 2.5) with $f \in \mathrm{FSym}_{nr}$ and $t \in Terms_{A'}$ such that $A' \sqsubseteq A$;
Sequential update $(u_1\,;u_2) \in Updates$ if $u_1, u_2 \in Updates$;
Parallel update $(u_1 \,||\, u_2) \in Updates$ if $u_1, u_2 \in Updates$;

[3] Note that `[]` is a normal function symbol declared in the signature.

Quantified update $(\texttt{for}\ x;\ \phi;\ u) \in \mathit{Updates}$ if $u \in \mathit{Updates}$, $x \in \mathrm{VSym}$, and $\phi \in \mathit{Formulas}$ ($\Rightarrow$ Def. 2.8);

Update application $(\{u_1\}\, u_2) \in \mathit{Updates}$ if $u_1, u_2 \in \mathit{Updates}$. $\lhd$

In both sequential and parallel[4] updates, a later sub-update overrides an earlier one. The difference however is that with sequential updates the evaluation of the second sub-update is affected by the evaluation of the first one. This is not the case for parallel updates, which are evaluated simultaneously.

Example 2.7. Consider the updates

$$c := c + 1 ; c := c + 2$$

and

$$c := c + 1 \,||\, c := c + 2$$

where c is a non-rigid constant. Evaluating these updates in a state satisfying $c = 0$ results in a state satisfying

$$c = 3$$

in the first case resp.

$$c = 2$$

in the second case. $\lhd$

2.3.5 Syntax of Java Card DL Formulas

Java Card DL formulas can contain real Java code (sequence of statements). We assume that this sequence is to be understood as part of a program (set of class declarations), not appearing in the formula. The background program must furthermore satisfy certain sanity constraints (syntax correctness, etc.), which we do not show here. These constraints do not pose a real restriction.

Now we can define the set of Java Card DL formulas:

Definition 2.8 (Formulas of Java Card DL). The set *Formulas* of *Java Card DL formulas* is inductively defined as the smallest set such that:

- $R(t_1, \ldots, t_n) \in \mathit{Formulas}$ for all predicate symbols $R : A_1, \ldots, A_n \in \mathrm{PSym}$ and terms $t_i \in \mathit{Terms}_{A'_i}$ ($\Rightarrow$ Def. 2.5) with $A'_i \sqsubseteq A_i$ $(1 \le i \le n)$,
- true, false $\in \mathit{Formulas}$,
- $\neg\phi$, $(\phi \vee \psi)$, $(\phi \wedge \psi)$, $(\phi \longrightarrow \psi)$, $(\phi \longleftrightarrow \psi) \in \mathit{Formulas}$ for all $\phi, \psi \in \mathit{Formulas}$,
- $\forall x.\phi$, $\exists x.\phi \in \mathit{Formulas}$ for all $\phi \in \mathit{Formulas}$ and all variables $x \in \mathrm{VSym}$,
- $\{u\}\,\phi \in \mathit{Formulas}$ for all $\phi \in \mathit{Formulas}$ and $u \in \mathit{Updates}$ ($\Rightarrow$ Def. 2.6),
- $\langle p \rangle \phi$, $[p]\phi \in \mathit{Formulas}$ for all $\phi \in \mathit{Formulas}$ and any legal sequence p of Java Card DL program statements.

In the following we often abbreviate formulas of the form $(\phi \longrightarrow \psi) \wedge (\neg\phi \longrightarrow \xi)$ by if ϕ then ψ else ξ. $\lhd$

[4] It should be noted that "parallel" updates are not related to concurrency.

Example 2.9. Given an appropriate type hierarchy and signature, the following are JAVA CARD DL formulas:

$\{c := 0\}(c = 0)$	a formula with an update
$(\{c := 0\}c) = c$	a formula containing a term with an update
$\langle$`int i=0; v=i;`$\rangle$(`v`$=$`0`)	a formula with a program-containing modal operator (diamond)
$[$`while(true){}`$]$*false*	a formula with a box modal operator
`x`$<$`y` $\longrightarrow \langle$`x++;y++;`$\rangle$`x`$<$`y`	a diamond formula expressing a pre- and a postcondition $\triangleleft$

Note 2.10. In program verification, one is usually interested in proving that the program under consideration satisfies some property for all possible input values. Since, by definition, terms (except those declared as static fields) and in particular logical variables, i.e., variables from the set VSym, may not occur within modal operators, it can be a bit tricky to express such properties.

The canonical way to express the desired property is to bind the program variable to a quantified logical variable via an update:

$$\forall n.\{\mathtt{v} := n\}(\langle\ \mathtt{ArrayList\ al=new\ ArrayList();}\ \ \mathtt{v=al.inc(v);}\rangle(\mathtt{v} = n + 1))\ .$$

$\triangleleft$

2.4 Semantics of JAVA CARD DL

The syntax of JAVA CARD DL extends the syntax of first-order logic with updates and modalities. On the semantic level this is reflected by the fact that, instead of one first-order model, we now have an (infinite) set of such models representing different program states.

Our semantics of JAVA CARD DL is based on so-called *Kripke structures*, which are commonly used to define the semantics of modal logics. In our case a Kripke structure consists of

- an (infinite) set of states $\mathcal{S}$. The states are first-order structures, providing interpretations of functions (including program variables) and predicates.
- a program input/output relation ρ fixing the meaning of programs occurring in modalities. This relation is dictated by the semantics of our programming language.

Analogously to the syntax definition, the semantics of JAVA CARD DL updates, terms, and formulas is mutually recursive. For better readability we still give separate definitions for the semantics of update, terms, and formulas, respectively.

2.4.1 Models: First-order and Kripke Structures

Definition 2.11 (First-order structure). A *first-order structure* is a triple $(\mathcal{D}, \delta, \mathcal{I})$ of

- a *domain* $\mathcal{D}$,
- a *dynamic type function* $\delta : \mathcal{D} \to \mathcal{T}_d$, and
- an *interpretation* $\mathcal{I}$,

such that, if we define the set of all domain elements that "fit" the type A

$$\mathcal{D}^A = \{d \in \mathcal{D} \mid \delta(d) \sqsubseteq A\} \quad ,$$

it holds that

- $\mathcal{D}^A$ is non-empty for all $A \in \mathcal{T}_d$,
- for any $f : A_1, \ldots, A_n \to A \in \text{FSym}$, $\mathcal{I}$ yields a function

$$\mathcal{I}(f) : \mathcal{D}^{A_1} \times \ldots \times \mathcal{D}^{A_n} \to \mathcal{D}^A \quad ,$$

- for any $p : A_1, \ldots, A_n \in \text{PSym}$, $\mathcal{I}$ yields a subset

$$\mathcal{I}(p) \subseteq \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n} \quad ,$$

- for type casts, $\mathcal{I}((A))(x) = x$ if $\delta(x) \sqsubseteq A$, otherwise $\mathcal{I}((A))(x)$ is an arbitrary but fixed[5] element of $\mathcal{D}^A$, and
- for equality, $\mathcal{I}(=) = \{(d, d) \mid d \in \mathcal{D}\}$,
- for type predicates, $\mathcal{I}(\sqsubseteq A) = \mathcal{D}^A$. $\lhd$

First-order structures are not quite sufficient to give a meaning to an arbitrary first-order term or formula: they say nothing about the variables. For this, we introduce the notion of a variable assignment.

Definition 2.12 (Variable assignment). Given a first-order structure $(\mathcal{D}, \delta, \mathcal{I})$, a *variable assignment* is a function β: VSym $\to \mathcal{D}$, such that

$$\beta(x) \in \mathcal{D}^A \quad \text{for all} \quad x{:}A \in \text{VSym} \quad .$$

We also define the *modification* β_x^d of a variable assignment β for any variable $x{:}A$ and any domain element $d \in \mathcal{D}^A$ by:

$$\beta_x^d(y) = \begin{cases} d & \text{if } y = x \\ \beta(y) & \text{otherwise.} \end{cases} \quad \lhd$$

[5] The chosen element may be different for different arguments, i.e., if $x \neq y$, then $\mathcal{I}((A))(x) \neq \mathcal{I}((A))(y)$ is allowed.

Definition 2.13 (Java Card DL Kripke structure). A *Java Card DL Kripke structure* $\mathcal{K}$ is a pair $(\mathcal{S}, \rho)$. $\mathcal{S}$ is the set of first-order structures over the given signature Σ, which serve as program states. ρ is the transition relation on $\mathcal{S}$, interpreting programs: $\rho(p) \subseteq \mathcal{S}^2$. The program relation ρ is, for all states $s_1, s_2 \in \mathcal{S}$ and any legal sequence p of Java Card DL program statements, defined by:

$$(s_1, s_2) \in \rho(p)$$
iff
p started in s_1 in a static context terminates normally in s_2
according to the Java language specification [Gosling et al., 2000].

$\mathcal{S}$ must also satisfy the following side conditions:

1. Rigid function and predicate symbols have a fixed interpretation for all states, while the interpretation of non-rigid symbols may differ from state to state.
2. The dynamic type function δ is the same for all states.
3. All states have the same domain $\mathcal{D}$ ("constant domain assumption"). We refer to $\mathcal{D}$ as *the* domain of $\mathcal{K}$.
4. The domain $\mathcal{D}$ must satisfy certain sanity properties not detailed here. In particular, the domain contains exactly the two elements *tt* and *ff* with dynamic type boolean and *null* as the only element with dynamic type Null. Moreover, we require that for each dynamic subtype A of type Object (except type Null) there is a countably infinite number of domain elements representing the Java Card objects of dynamic type A.
5. We demand that the set $\mathcal{S}$ of states of any Kripke structure consists of *all* first-order structures satisfying the above restrictions.

Furthermore, all Java Card DL Kripke structure must interpret certain function and predicate symbols that we have distinguished as *predefined* ($\Rightarrow$ Note 2.4) in a predefined way.[6] $\lhd$

2.4.2 Semantics of Java Card DL Updates

Similar to the first-order case we inductively define a valuation function val_s assigning meaning to updates, terms, and formulas. Since non-rigid function and predicate symbols can have different meanings in different states, the valuation function is parameterized with a Java Card DL state, i.e., for each state s in $\mathcal{S}$, there is a separate valuation function.

The intuitive meaning of updates is that the term or formula following the update is to be evaluated not in the current state but in the state described by the update. To be more precise, updates do not describe a state completely, but merely the difference between the current state and the target state. As we see later this is similar to the

[6] The semantics of each predefined symbol can be constrained completely or only partially. Such constraints are typically expressed as axioms, together with symbol declaration. An example of a partially constrained predefined function in Java Card DL is division on integers.

semantics of programs contained in modal operators and indeed updates are used to describe the effect of programs.

In parallel updates $u_1 \parallel u_2$ (as well as in quantified updates) clashes can occur, where u_1 and u_2 simultaneously modify a non-rigid function f for the same arguments in an inconsistent way, i.e., by assigning it different values. To handle this problem, we use a *last-win* semantics, i.e., the update that syntactically occurs last dominates earlier ones. In the more general situation of quantified (unbounded parallel) updates `for` x; ϕ; u, we assume that a fixed well-ordering $\leq$ on the universe $\mathcal{D}$ exists (i.e., a total ordering such that every non-empty subset $\mathcal{D}_{sub} \subseteq \mathcal{D}$ has a least element $min_{\leq}(\mathcal{D}_{sub})$). The parallel application of unbounded sets of updates can then be well-ordered as well, and clashes are resolved by giving precedence to the update with the smallest value of x.

As every set can be well-ordered [Zermelo, 1904], this does not restrict the range of possible domains. The particular order imposed on the domain of a Kripke structure is a parameter that can be chosen freely depending on the problem. The KeY system implements a certain "natural" order, which we do not describe here (but see [Beckert et al., 2007]).

The semantics of applying an update to a given JAVA CARD DL state $(\mathcal{D}, \delta, \mathcal{I})$ is defined as a new state $(\mathcal{D}, \delta, \mathcal{I}')$ that differs only in the interpretation of the updated function, and this only for the arguments specified in the update. But before we give the exact definition, we introduce semantic updates and discuss their consistency.

Definition 2.14 (Semantic update). A *semantic update* is a triple $(f, (d_1, \ldots, d_n), d)$ such that

- $f: A_1, \ldots, A_n \to A \in \mathrm{FSym}_{nr}$,
- $d_i \in \mathcal{D}^{A_i}$ $(1 \leq i \leq n)$, and
- $d \in \mathcal{D}^A$. ◁

Since updates in general modify more than one location (a location is a pair $(f, (d_1, \ldots, d_n))$), we define sets of consistent semantic updates.

Definition 2.15 (Consistent semantic updates). A set CU of semantic updates is called *consistent* if for all $(f, (d_1, \ldots, d_n), d), (f', (d'_1, \ldots, d'_m), d') \in CU$,

$$d = d' \text{ if } f = f', n = m, \text{ and } d_i = d'_i \ (1 \leq i \leq n) \ .$$

Let $\mathcal{CU}$ denote the set of consistent semantic updates. ◁

As we see in Def. 2.17, a syntactic update describes the modification of a state s as a set CU of consistent semantic updates. In order to obtain the state in which the terms, formulas, or updates following an update u are evaluated, CU is applied to s yielding a state s'.

Definition 2.16 (Application of semantic updates). Let $s = (\mathcal{D}, \delta, \mathcal{I})$ be a first-order structure. For any set $CU \in \mathcal{CU}$ of consistent semantics updates, the *application result* $CU(s)$ is defined as the structure $(\mathcal{D}', \delta', \mathcal{I}')$ with

$$\begin{aligned}
\mathcal{D}' &= \mathcal{D} \\
\delta' &= \delta \\
\mathcal{I}'(f)(d_1,\ldots,d_n) &= \begin{cases} d & \text{if } (f,(d_1,\ldots,d_n),d) \in CU \\ \mathcal{I}(f)(d_1,\ldots,d_n) & \text{otherwise} \end{cases}
\end{aligned}$$

for all $f\colon A_1,\ldots,A_n \to A \in \mathrm{FSym}_{nr}$ and $d_i \in \mathcal{D}^{A_i}$ $(1 \le i \le n)$. ◁

Intuitively, a set CU of consistent semantic updates modifies the interpretation of s for the locations that are contained in CU. The consistency condition in Def. 2.15 guarantees that the interpretation function $\mathcal{I}'$ in Def. 2.16 is well-defined.

Definition 2.17 (Semantics of Java Card DL updates). Let $\mathcal{K} = (\mathcal{S}, \rho)$ be a Kripke structure of Java Card DL with the domain ordered by a total order $\prec$ (explained above), and let β be a variable assignment.

For every state $s = (\mathcal{D}, \delta, \mathcal{I}) \in \mathcal{S}$, the valuation function $\mathrm{val}_s : \mathit{Updates} \to \mathcal{CU}$ for updates is inductively defined by

- $\mathrm{val}_{s,\beta}(f(t_1,\ldots,t_n) := t) = \{(f,(d_1,\ldots,d_n),d)\}$ where

$$\begin{aligned}
d_i &= \mathrm{val}_{s,\beta}(t_i) \qquad (1 \le i \le n) \\
d &= \mathrm{val}_{s,\beta}(t) \ ,
\end{aligned}$$

- $\mathrm{val}_{s,\beta}(u_1\,;u_2) = (U_1 \cup U_2) \setminus C$ where

$$\begin{aligned}
U_1 &= \mathrm{val}_{s,\beta}(u_1) \\
U_2 &= \mathrm{val}_{S',\beta}(u_2) \qquad \text{with } S' = \mathrm{val}_{s,\beta}(u_1)(S) \\
C &= \{(f,(d_1,\ldots,d_n),d) \mid (f,(d_1,\ldots,d_n),d) \in U_1 \text{ and} \\
&\qquad (f,(d_1,\ldots,d_n),d') \in U_2 \text{ for some } d' \neq d\} \ ,
\end{aligned}$$

- $\mathrm{val}_{s,\beta}(u_1 \,||\, u_2) = (U_1 \cup U_2) \setminus C$ where

$$\begin{aligned}
U_1 &= \mathrm{val}_{s,\beta}(u_1) \\
U_2 &= \mathrm{val}_{s,\beta}(u_2) \\
C &= \{(f,(d_1,\ldots,d_n),d) \mid (f,(d_1,\ldots,d_n),d) \in U_1 \text{ and} \\
&\qquad (f,(d_1,\ldots,d_n),d') \in U_2 \text{ for some } d' \neq d\} \ ,
\end{aligned}$$

- $\mathrm{val}_{s,\beta}(\texttt{for}\ x;\ \phi;\ u) = U$ where

$$\begin{aligned}
U = \{(f,(d_1,\ldots,d_n),d) \mid{} & ((f,(d_1,\ldots,d_n),d),a) \in \mathit{dom} \text{ for some } a \in \mathcal{D}^A \\
& \text{and } b \nprec a \text{ for all } ((f,(d_1,\ldots,d_n),d'),b) \in \mathit{dom}\}
\end{aligned}$$

with $\mathit{dom} = \bigcup\limits_{\substack{a \in \mathcal{D}^A \\ s,\beta_x^a \models \phi}} (\mathrm{val}_{s,\beta_x^a}(u) \times \{a\})$, and $x{:}A$,

- $\mathrm{val}_{s,\beta}(\{u_1\}\,u_2) = \mathrm{val}_{s',\beta}(u_2)$ with $s' = \mathrm{val}_{s,\beta}(u_1)(s)$.

For an update u without free variables we simply write $\mathrm{val}_s(u)$ since $\mathrm{val}_{s,\beta}(u)$ is independent of β. ◁

Example 2.18. Consider the quantified update

$$\texttt{for}\ x;\ x=0 \lor x=1;\ h(x):=0 \ .$$

It updates two different locations, so there is no clash. The result of applying this update is a state satisfying $h(0)=0$ and $h(1)=0$.

Now consider the update

$$\texttt{for}\ x;\ x=0 \lor x=1;\ h(0):=5-x \ .$$

It attempts to simultaneously assign the values 5 and 4 to the same location $h(0)$. Let's assume that x ranges over the positive integers, which allows us to choose the natural "less than or equal" ordering relation $\leq$ on the relevant part of the domain. Since $0 \leq 1$, we give preference to the update binding x to zero, and the result of applying the quantified update is a state satisfying $h(0)=5$. ◁

2.4.3 Semantics of Java Card DL Terms

The valuation function for Java Card DL terms is defined analogously to the one for first-order terms, though depending on the Java Card DL state.

Definition 2.19 (Semantics of Java Card DL terms). Let $\mathcal{K}=(\mathcal{S},\rho)$ be a Kripke structure of Java Card DL, and let β be a variable assignment.

For every state $s=(\mathcal{D},\delta,\mathcal{I})\in\mathcal{S}$, the valuation function val_s for terms is inductively defined by:

$$\begin{aligned}
\mathrm{val}_{s,\beta}(x) &= \beta(x) \quad \text{for variables } x\\
\mathrm{val}_{s,\beta}(f(t_1,\dots,t_n)) &= \mathcal{I}(f)(\mathrm{val}_{s,\beta}(t_1),\dots,\mathrm{val}_{s,\beta}(t_n))\\
\mathrm{val}_{s,\beta}(\text{if } \phi \text{ then } t_1 \text{ else } t_2) &= \begin{cases}\mathrm{val}_{s,\beta}(t_1) & \text{if } s,\beta\models\phi\\ \mathrm{val}_{s,\beta}(t_2) & \text{if } s,\beta\not\models\phi\end{cases}\\
\mathrm{val}_{s,\beta}(\{u\}(t)) &= \mathrm{val}_{s_1,\beta}(t) \quad \text{with } s_1=\mathrm{val}_{s,\beta}(u)(s)
\end{aligned}$$

Since $\mathrm{val}_{s,\beta}(t)$ does not depend on β if t is ground, we write $\mathrm{val}_s(t)$ in that case. ◁

The function and predicate symbols of a signature are divided into disjoint sets of rigid and non-rigid function and predicate symbols, respectively. By Def. 2.13, rigid symbols have the same meaning in all states of a given Kripke structure. The following syntactic criterion continues the notion of rigidness from function symbols to terms.

Definition 2.20. A Java Card DL term t is *rigid*

- if $t=x$ and $x\in\mathrm{VSym}$,
- if $t=f(t_1,\dots,t_n)$, $f\in\mathrm{FSym}_r$ and the sub-terms t_i are rigid ($1\leq i\leq n$),
- if $t=\{u\}(s)$ and s is rigid,
- if $t=(\text{if } \phi \text{ then } t_1 \text{ else } t_2)$ and the formula ϕ is rigid (Def. 2.22) and the sub-terms t_1, t_2 are rigid. ◁

Rigid terms have the same meaning in all Java Card DL states, whereas the meaning of non-rigid terms may differ from state to state.

2.4.4 Semantics of Java Card DL Formulas

Definition 2.21 (Semantics of Java Card DL formulas). Let $\mathcal{K} = (\mathcal{S}, \rho)$ be a Kripke structure of Java Card DL, and let β be a variable assignment.

For every state $s = (\mathcal{D}, \delta, \mathcal{I}) \in \mathcal{S}$ the validity relation $\models$ for Java Card DL formulas is inductively defined by:

- $s, \beta \models R(t_1, \ldots, t_n)$ iff $(\mathrm{val}_{s,\beta}(t_1), \ldots, \mathrm{val}_{s,\beta}(t_n)) \in \mathcal{I}(R)$
- $s, \beta \models \mathrm{true}$
- $s, \beta \not\models \mathrm{false}$
- $s, \beta \models \neg\phi$ iff $s, \beta \not\models \phi$
- $s, \beta \models (\phi \wedge \psi)$ iff $s, \beta \models \phi$ and $s, \beta \models \psi$
- $s, \beta \models (\phi \vee \psi)$ iff $s, \beta \models \phi$ or $s, \beta \models \psi$ (or both)
- $s, \beta \models (\phi \longrightarrow \psi)$ iff $s, \beta \not\models \phi$ or $s, \beta \models \psi$ (or both)
- $s, \beta \models \forall x.\phi$ iff $s, \beta_x^d \models \phi$ for every $d \in \mathcal{D}^A$ (if $x : A$)
- $s, \beta \models \exists x.\phi$ iff $s, \beta_x^d \models \phi$ for some $d \in \mathcal{D}^A$ (if $x : A$)
- $s, \beta \models \{u\}(\phi)$ iff $s_1, \beta \models \phi$ with $s_1 = \mathrm{val}_{s,\beta}(u)(s)$
- $s, \beta \models \langle p \rangle \phi$ iff there exists some state $s' \in \mathcal{S}$ such that $(s, s') \in \rho(p)$ and $s', \beta \models \phi$
- $s, \beta \models [p]\phi$ iff $s', \beta \models \phi$ for every state $s' \in \mathcal{S}$ with $(s, s') \in \rho(p)$

We write $S \models \phi$ for a closed formula ϕ, since β is then irrelevant. $\lhd$

Similar to rigidness of terms, we now define rigidness of formulas.

Definition 2.22. A Java Card DL formula ϕ is *rigid*

- if $\phi = p(t_1, \ldots, t_n)$, $p \in \mathrm{PSym}_r$ and the terms t_i are rigid ($1 \le i \le n$),
- if $\phi = \mathrm{true}$ or $\phi = \mathrm{false}$,
- if $\phi = \neg\psi$ and ψ is rigid,
- $\phi = (\psi_1 \vee \psi_2)$, $\phi = (\psi_1 \wedge \psi_2)$, or $\phi = (\psi_1 \longrightarrow \psi_2)$, and ψ_1, ψ_2 are rigid,
- if $\phi = \forall x.\psi$ or $\phi = \exists x.\psi$, and ψ is rigid,
- $\phi = \{u\}\psi$ and ψ is rigid. $\lhd$

Note 2.23. A formula $\langle p \rangle \psi$ or $[p]\psi$ is *not* rigid, even if ψ is rigid, since the truth value of such formulas depends, e.g., on the termination behavior of the program statements p in the modal operator. Whether a program terminates or not in general depends on the state the program is started in. $\lhd$

Rigid formulas—in contrast to non-rigid formulas—have the same meaning in all Java Card DL states.

Finally, we define what it means for a formula to be valid or satisfiable.

Definition 2.24 (Validity). We say that a Kripke structure $\mathcal{K} = (\mathcal{S}, \rho)$ is a *model* of a formula ϕ, or that ϕ is $\mathcal{K}$*-valid*, iff $s, \beta \models \phi$ for all $s \in \mathcal{S}$ and all variable assignments β (i.e., when ϕ is true in all states).

A formula ϕ is *valid* if all Kripke structures are a model of ϕ. $\lhd$

Example 2.25. We now check the formulas from Example 2.9 for validity. We assume that c is an integer non-rigid constant in the signature.

$\models \{c := 0\}\,(c = 0)$	since in the state in which $c = 0$ is evaluated, c is indeed 0 (due to the update).
$\not\models (\{c := 0\}\,c) = c$	since $(\{c := 0\}c)$ evaluates to 0 in any state but there are states in which c (the right side) is different from 0.
$\models \langle$`int i=0; v=i;`$\rangle$`v`$=0$	since the program always terminates in state with `v` $= 0$.
$\models [$`while(true){}`$]$*false*	since *false* has to hold *if* the program terminates, but the program never terminates.
$\not\models$ `x` $<$ `y` $\longrightarrow \langle$`x++;y++;`$\rangle$`x` $<$ `y`	since `y` may suffer integer overflow upon increment, but not `x`. The fomula would have been valid assuming a mathematical integer semantics. KeY actually offers the possibility to work with different integer semantics.

◁

2.4.5 Java Card-reachable States

Not all states of a Java Card DL Kripke structure can actually occur during an execution of a Java Card program. Indeed, a state is (only) Java Card-reachable if it satisfies the following conditions:

1. A finite number of objects are created.[7]
2. Reference type attributes of created objects are either null or point to some other created object. Similarly, all entries of created reference-type arrays are either null or point to some created object.
3. For any array a with dynamic type $\delta(a) = A[\,]$, the dynamic type of the array entries is a subtype of A (an assignment violating this condition throws an `ArrayStoreException` in Java).
4. Initialized classes are not erroneous and other conditions related to class initialization.
5. For multi-threaded programs, program counters of threads and lock states must be consistent with locking operations in the program.

Thus, there are formulas that are true in all Java Card-reachable states but that are not valid in Java Card DL. This problem can be overcome by adding a special predicate *inReachableState* (formalizing the above conditions) to the invariants of the program to be verified. Then, states that are not reachable by any Java Card program are excluded from consideration.

When a correctness proof is started, the KeY system automatically adds the predicate *inReachableState* to the precondition of the specification. In the majority of cases, proofs can be completed without considering *inReachableState*. There are however situations that require the use of *inReachableState*. To deal with such situations, the KeY calculus provides rules that allow the user to extract parts of *inReachableState* that are necessary to close the proof.

[7] In Java Card DL, objects are represented by domain elements, and the domain is defined to be constant. Whether an object is created or not is indicated by a ghost Boolean attribute.

2.5 The Calculus for JAVA CARD DL

2.5.1 Sequents, Rules, and Proofs

The KeY system's calculus for JAVA CARD DL is a Gentzen-style [Gentzen, 1935] sequent calculus. A *calculus* is formally a set of rules. Rules are used to derive sequents from other sequents.

Definition 2.26 (Sequents). A *sequent* is of the form $\Gamma \Longrightarrow \Delta$, where Γ, Δ are sets of closed JAVA CARD DL formulas.

The left-hand side Γ is called *antecedent* and the right-hand side Δ is called *succedent* of the sequent.

The semantics of a sequent

$$\phi_1, \dots, \phi_m \Longrightarrow \psi_1, \dots, \psi_n$$

is the same as that of the formula

$$(\phi_1 \wedge \dots \wedge \phi_m) \longrightarrow (\psi_1 \vee \dots \vee \psi_m) \ .$$

◁

Definition 2.27 (Rule, Derivability). A *rule R* is a binary relation between (a) the set of all tuples of sequents and (b) the set of all sequents.

If $R(\langle P_1, \dots, P_k \rangle,\ C)$ $(k \geq 0)$, then the *conclusion C* is *derivable* from the *premisses* $P_1, \dots, P_k$ using rule R.

The set of sequents that are *derivable* is the smallest set such that: If there is a rule in the calculus that allows to derive a sequent S from premisses that are all derivable, then S is derivable in C. ◁

Intuitively, a proof for a sequent S is a derivation of S written as a tree with root S, where the sequent in each node is derivable from the sequents in its child nodes.

Definition 2.28 (Proof tree, Proof). A *proof tree* is a finite tree, such that:

- each node of the tree is annotated with a sequent
- each inner node of the tree is additionally annotated with one of the calculus rules that have at least one premiss. The rule relates the node's sequent to the sequents of its descendants. In particular, the number of node's descendants is the same as the number of premisses of the rule.
- a leaf node may or may not be annotated with a rule. If it is, it is one of the rules that have no premisses, also known as closing rules.

A *proof tree for a formula* ϕ is a proof tree where the root sequent is annotated with $\Longrightarrow \phi$.

A *branch* of a proof tree is a path from the root to one of the leaves. A branch is *closed* if the leaf is annotated with one of the closing rules. A proof tree is *closed* if all its branches are closed, i.e., every leaf is annotated with a closing rule.

A closed proof tree (for a formula ϕ) is also called a *proof* (for ϕ). ◁

2.5.2 Soundness and Completeness of the Calculus

Soundness

The most important property of the Java Card DL calculus is soundness, i.e., only valid formulas are derivable.

Proposition 2.29 (Soundness). *If a sequent $\Gamma \Longrightarrow \Delta$ is derivable in the Java Card DL calculus (Def. 2.27), then it is valid, i.e., the formula $\bigwedge \Gamma \longrightarrow \bigvee \Delta$ is logically valid (Def. 2.24).* $\lhd$

It is easy to show that the whole calculus is sound if and only if all its rules are sound. That is, if the premisses of any rule application are valid sequents, then the conclusion is valid as well. Given the soundness of the existing core rules of the Java Card DL calculus, the user can add new rules, whose soundness must then be proven w.r.t. the existing rules. A bigger perspective on the issue of calculus soundness is given in Chapter 9.

Relative Completeness

Ideally, one would like a program verification calculus to be able to prove all statements about programs that are true, which means that all valid sequents should be derivable. That, however, is *impossible* because Java Card DL includes first-order arithmetic, which is already inherently incomplete as established by Gödel's Incompleteness Theorem [Gödel, 1931]. Another, equivalent, argument is that a complete calculus for Java Card DL would yield a decision procedure for the Halting Problem, which is well-known to be undecidable. Thus, a logic like Java Card DL cannot ever have a calculus that is both sound and complete.

Still, it is possible to define a notion of *relative completeness* [Cook, 1978], which intuitively states that the calculus is complete "up to" the inherent incompleteness in its first-order part. A relatively complete calculus contains all the rules that are necessary to prove valid program properties. It only may fail to prove such valid formulas whose proof would require the derivation of a non-provable first-order property (being purely first-order, its provability would be independent of the program part of the calculus).

Proposition 2.30 (Relative Completeness). *If a sequent $\Gamma \Longrightarrow \Delta$ is valid, i.e., the formula $\bigwedge \Gamma \longrightarrow \bigvee \Delta$ is logically valid (Def. 2.24), then there is a finite set Γ_{FOL} of logically valid first-order formulas such that the sequent*

$$\Gamma_{FOL}, \Gamma \Longrightarrow \Delta$$

is derivable in the Java Card DL calculus. $\lhd$

The standard technique for proving that a program verification calculus is relatively complete [Harel, 1979] hinges on a central lemma expressing that for all Java Card DL formulas there is an equivalent purely first-order formula. A completeness proof for the object-oriented dynamic logic ODL [Beckert and Platzer, 2006b], which captures the essence of Java Card DL, is given by Platzer [2004a].

2.5.3 Rule Schemata and Schema Variables

To attain a finite rule description we use *rule schemata*, i.e., rules containing *schema variables*. Schema variables represent concrete syntactical elements (e.g., terms, formulas or programs).

Definition 2.31 (Rule schema). A *rule schema* is of the form

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_k}{C} \qquad (k \geq 0)$$

where $P_1, \ldots, P_k$ and C are schematic sequents, i.e., sequents containing schema variables. $\lhd$

A rule schema $P_1 \cdots P_k \,/\, C$ represents a rule R if the following equivalence holds: a sequent C^* is derivable from premisses $P_1^*, \ldots, P_k^*$ iff $P_1^* \cdots P_k^* \,/\, C^*$ is an instance of the rule schema. Schema instances are constructed by instantiating the schema variables with syntactical constructs (terms, formulas, etc.) which are compliant to the kinds of the schema variables. One rule schema represents infinitely many rules, namely, its instances.

There are many cases, where a basic rule schema is not sufficient for describing a rule. Even if its general form adheres to a pattern that is describable in a schema, there may be details in a rule that cannot be expressed schematically. For example, in the rules for handling existential quantifiers, there is the restriction that (Skolem) constants introduced by a rule application must not already occur in the sequent. When a rule is described schematically, such constraints are added as a note to the schema.

All the rules of our calculus perform one (or more) of the following actions:

- A sequent is recognised as an axiom, and the corresponding proof branch is closed.
- A formula in a sequent is modified. A single formula (in the conclusion of the rule) is chosen to be in focus. It can be modified or deleted from the sequent. Note that we do not allow more than one formula to be modified by a rule application.
- Formulas are added to a sequent. The number of formulas that are added is finite and is the same for all possible applications of the same rule schema.
- The proof branches. The number of new branches is the same for all possible applications of the same rule schema.

Moreover, whether a rule is applicable and what the result of the application is, depends on the presence of certain formulas in the conclusion.

The above list of possible actions excludes, for example, rules performing changes on all formulas in a sequent or that delete all formulas with a certain property.

Thus, all our rules preserve the "context" in a sequent, i.e., the formulas that are not in the focus of the rule remain unchanged. Therefore, we can simplify the notation of rule schemata, and leave this context out. Similarly, an update that is common to all premisses can be left out (this is formalized in Def. 2.32). Intuitively, if a rule

"$\phi \Longrightarrow \psi \,/\, \phi' \Longrightarrow \psi'$" is correct, then $\phi' \Longrightarrow \psi'$ can be derived from $\phi \Longrightarrow \psi$ in all possible contexts. In particular, the rule then is correct in a context described by $\Gamma, \Delta, \mathcal{U}$, i.e., in all states s for which there is a state s_0 such that $\Gamma \Longrightarrow \Delta$ is true in s_0 and s is reached from s_0 by executing $\mathcal{U}$. Therefore, "$\Gamma,\ \mathcal{U}\phi \Longrightarrow \mathcal{U}\psi\ \Delta \,/\, \Gamma\ \mathcal{U}\phi' \Longrightarrow \mathcal{U}\psi',\ \Delta$" is a correct instance of "$\phi \Longrightarrow \psi \,/\, \phi' \Longrightarrow \psi'$", and $\Gamma, \Delta, \mathcal{U}$ do not have to be included in the schema. Instead we allow them to be added during application. Note, however, that the *same* $\Gamma, \Delta, \mathcal{U}$ have to be added to all premisses of a rule schema.

There are also a few rules (mostly invariant rules of different flavors) where the context cannot be omitted. Such rules are indicated with the $(*)$ symbol.

Definition 2.32. If

$$\frac{\begin{array}{c}\phi_1^1, \ldots, \phi_{m_1}^1 \Longrightarrow \psi_1^1, \ldots, \psi_{n_1}^1 \\ \vdots \\ \phi_1^k, \ldots, \phi_{m_k}^k \Longrightarrow \psi_1^k, \ldots, \psi_{n_k}^k\end{array}}{\phi_1, \ldots, \phi_m \Longrightarrow \psi_1, \ldots, \psi_n}$$

is an instance of a rule schema, then

$$\frac{\begin{array}{c}\Gamma,\ \mathcal{U}\phi_1^1, \ldots, \mathcal{U}\phi_{m_1}^1 \Longrightarrow \mathcal{U}\psi_1^1, \ldots, \mathcal{U}\psi_{n_1}^1,\ \Delta \\ \vdots \\ \Gamma,\ \mathcal{U}\phi_1^k, \ldots, \mathcal{U}\phi_{m_k}^k \Longrightarrow \mathcal{U}\psi_1^k, \ldots, \mathcal{U}\psi_{n_k}^k,\ \Delta\end{array}}{\Gamma,\ \mathcal{U}\phi_1, \ldots, \mathcal{U}\phi_m \Longrightarrow \mathcal{U}\psi_1, \ldots, \mathcal{U}\psi_n,\ \Delta}$$

is an inference rule of our DL calculus, where $\mathcal{U}$ is an arbitrary syntactic update (including the empty update), and Γ, Δ are finite sets of context formulas.

If, however, the symbol $(*)$ is added to the rule schema, the context $\Gamma, \Delta, \mathcal{U}$ must be empty, i.e., only instances of the schema itself are inference rules. $\lhd$

Example 2.33. Consider, for example, the rule impRight:

$$\text{impRight}\ \frac{\phi \Longrightarrow \psi}{\Longrightarrow \phi \longrightarrow \psi}$$

When this schema is instantiated, a context consisting of Γ, Δ and an update $\mathcal{U}$ can be added, and the schema variables ϕ, ψ can be instantiated with formulas that are not purely first-order. For example, the following is an instance of impRight:

$$\frac{x = 1,\ \{x := 0\}(x = y) \Longrightarrow \{x := 0\}\langle \texttt{m();} \rangle(y = 0)}{x = 1 \Longrightarrow \{x := 0\}(x = y \longrightarrow \langle \texttt{m();} \rangle(y = 0))}$$

where $\Gamma = (x = 1)$, Δ is empty, and the context update $\mathcal{U}$ is $\{x := 0\}$. $\lhd$

Schema variables and their kinds

The schema variables used in rule schemata are all assigned a kind that determines which class of concrete syntactic elements they represent. In the following sections, we often do not explicitly mention the kinds of schema variables but use the name of the variables to indicate their kind. Table 2.1 gives the correspondence between names of schema variables that represent pieces of JAVA code and their kinds. In addition, we use the schema variables ϕ, ψ to represent formulas and Γ, Δ to represent sets of formulas.

Table 2.1. Correspondence between names of schema variables and their kinds

π	non-active prefix of JAVA code (Sect. 2.5.4)
ω	"rest" of JAVA code after the active statement (Sect. 2.5.4)
p, q	JAVA code (arbitrary sequence of statements)
e	arbitrary JAVA expression
se	simple expression, i.e., any expression whose evaluation, a priori, does not have any side-effects. It is defined as one of the following: (a) a local variable (b) `this.`a, i.e., an access to an instance attribute via the target expression `this` (or, equivalently, no target expression) (c) an access to a static attribute of the form t`.`a, where the target expression t is a type name or a simple expression (d) a literal (e) a compile-time constant (f) an `instanceof` expression with a simple expression as the first argument (g) a `this` reference. An access to an instance attribute o`.`a is not considered simple because a `NullPointerException` may be thrown
nse	non-simple expression, i.e., any expression that is not simple (see above)
lhs	simple expression that can appear on the left-hand-side of an assignment. This amounts to the items (a)–(c) from above
$v, v_0, \dots$	local program variables (i.e., non-rigid constants)
a	attribute
l	label
$args$	argument tuple, i.e., a tuple of expressions
cs	sequence of catch clauses
$mname$	name of a method
T	type expression
C	name of a class or interface

If a schema variable T representing a type expression is indexed with the name of another schema variable, say e, then it only matches with the JAVA type of the expression with which e is instantiated. For example, "T_w v = w" matches the JAVA code "`int i = j`" if and only of the type of `j` is `int` (and not, e.g., `byte`).

2.5.4 The Active Statement in a Modality

The rules of our calculus operate on the first *active* statement p in a modality $\langle \pi p \omega \rangle$ or $[\pi p \omega]$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch-finally` blocks, and beginnings "`method-frame(...){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately.

The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on (in particular, ω contains closing braces corresponding to the opening braces in π). For example, if a rule is applied to the following Java block operating on its first active command "`i=0;`", then the non-active prefix π and the "rest" ω are the indicated parts of the block:

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

In versions of dynamic logic for simple programming languages, where no prefixes are needed, any formula of the form $\langle pq\rangle\phi$ can be replaced by $\langle p\rangle\langle q\rangle\phi$. In our calculus, decomposing of $\langle \pi pq\omega\rangle\phi$ into $\langle \pi p\rangle\langle q\omega\rangle\phi$ is not possible (unless the prefix π is empty) because πp is not a valid program; and the formula $\langle \pi p\omega\rangle\langle \pi q\omega\rangle\phi$ cannot be used either because its semantics is in general different from that of $\langle \pi pq\omega\rangle\phi$.

2.5.5 The Essence of Symbolic Execution

Our calculus works by reducing the question of a formula's validity to the question of the validity of several simpler formulas. Since Java Card DL formulas contain programs, the Java Card DL calculus has rules that reduce the meaning of programs to the meaning of simpler programs. For this reduction we employ the technique of *symbolic execution* [King, 1976]. Symbolic execution in Java Card DL resembles playing an accordion: you make the program longer (though simpler) before you can make it shorter.

For example, to find out whether the sequent

$$\Longrightarrow\langle\texttt{o.next.prev=o;}\rangle\texttt{o.next.prev}=\texttt{o}$$

is valid, we symbolically execute the Java code in the diamond modality. At first, the calculus rules transform it into an equivalent but longer (albeit in a sense simpler) sequence of statements:

$$\Longrightarrow\langle\texttt{ListEl v; v=o.next; v.prev=o;}\rangle\texttt{o.next.prev}=\texttt{o}\ .$$

This way, we have reduced reasoning about the complex expression `o.next.prev=o` to reasoning about several simpler expressions. We call this process *unfolding*, and it works by introducing fresh local variables to store intermediate computation results.

Now, when analyzing the first of the simpler assignments (after removing the variable declaration), one has to consider the possibility that evaluating the expression `o.next` may produce a side effect if `o` is `null` (in that case an exception is thrown). However, it is not possible to unfold `o.next` any further. Something else has to be done, namely a case distinction. This results in the following two new goals:

$\texttt{o}\neq\texttt{null} \Longrightarrow \{\texttt{v:=o.next}\}\langle\texttt{v.prev=o;}\rangle\texttt{o.next.prev=o}$

$\texttt{o=null} \Longrightarrow \langle\texttt{throw new NullPointerException();}\rangle\texttt{o.next.prev=o}$

Thus, we can state the essence of symbolic execution: the JAVA code in the formulas is step-wise unfolded and replaced by case distinctions and syntactic updates.

Of course, it is not a coincidence that these two ingredients (case distinctions and updates) correspond to two of the three basic programming constructs. The third basic construct are loops. These cannot in general be treated by symbolic execution, since using symbolic values (as opposed to concrete values) the number of loop iterations is unbounded. Symbolically executing a loop, which is called "unwinding", is useful and even necessary, but unwinding cannot eliminate a loop in the general case. To treat arbitrary loops, one needs to use induction (⇒ Sect. 2.6.4) or loop invariants (⇒ Sect. 2.8). Also, certain kinds of loops can be translated into quantified updates [Gedell and Hähnle, 2006].

Method invocations can be symbolically executed, replacing a method call by the method's implementation. However, it is often useful to instead use a method's contract so that it is only symbolically executed once—during the proof that the method satisfies its contract—instead of executing it for each invocation.

2.5.6 Components of the Calculus

Our JAVA CARD DL calculus has five major components, which are described in detail in the following sections. Since the calculus consists of hundreds of rules, however, we cannot list them all in this book. Instead, we give typical examples for the different rule types and classes (a complete list can be found on the KeY project website).

In particular, we usually only give the rule versions for the diamond modality $\langle\cdot\rangle$. The rules for box modality $[\cdot]$ are mostly the same—notable exceptions are the rules for handling loops (Sect. 2.8) and some of the rules for handling abrupt termination (Sect. 2.7.6).

The five components of the JAVA CARD DL calculus are:

1. Non-program rules, i.e., rules that are not related to particular program constructs. This includes first-order rules, rules for data types (in particular the integers), rules for modalities (e.g., rules for empty modalities), and the induction rule.
2. Rules that work towards reducing/simplifying the program and replacing it by a combination of case distinction (proof branches) and sequences of updates. These always (and only) apply to the first active statement. A "simpler" program may be syntactically longer; it is simpler in the sense that expressions are not as deeply nested or have less side-effects.
3. An invariant rule that handles loops for which no fixed upper bound on the number of iterations exists. (Another technique for doing this is induction, which is part of Component 1.)
4. Rules that replace a method invocation by the method's contract.
5. Update simplification.

Component 2 is the core for handling Java Card programs occurring in formulas. These rules can be applied automatically, and they can do everything needed for handling programs except evaluating loops and using method specifications.

The overall strategy is to use the rules in Component 2, interspersed with applications of rules in Component 3 and Component 4 for handling loops resp. methods, to step-wise eliminate the program and replace it by updates and case distinctions. After each step, Component 5 is used to simplify/eliminate updates. The final result of this process are sequents containing pure first-order formulas. These are then handled by Component 1.

The symbolic execution process is for the most part done automatically by the KeY system. Of course, this presupposes that loop invariants are given. In addition, the user can give the prover modularization hints such as method contracts, lemmas, etc. User interaction may also be necessary when solving the first-order problem that is left at the end of symbolic execution (e.g., quantifier instantiation). At this stage, the KeY system can request assistance from external decision procedures for first-order logic and basic data type theories.

2.6 Calculus Component 1: Non-program Rules

2.6.1 First-order Rules

This component includes:

- Standard first-order rules (⇒ Fig. 2.2)
- Almost standard equality rules (which we do not show). As we deal with a modal logic, an equality $t_1 = t_2$ may only be used for rewriting if
  - both t_1 and t_2 are rigid terms (Def. 2.20), or
  - the equality $t_1 = t_2$ and the occurrence of t_i that is being replaced are (a) not in the scope of two different program modalities and (b-1) not in the scope of two different updates or (b-2) in the scope of two updates with the same effect.

  Equality handling is further complicated by subtyping
- Rules for reasoning about type casts and type predicates (which we do not show)
- Standard arithmetical rules.

2.6.2 The Cut Rule and Lemma Introduction

The cut rule

$$\mathsf{cut}\ \frac{\Longrightarrow \phi \qquad \phi \Longrightarrow}{\Longrightarrow}$$

allows to introduce a lemma ϕ, which is an arbitrary Java Card DL formula. The lemma occurs in the succedent of the left premiss (where, intuitively speaking, the lemma has to be proved) and in the antecedent of the right premiss (where, intuitively

$$\mathsf{andLeft}\ \frac{\phi, \psi \Longrightarrow}{\phi \land \psi \Longrightarrow} \qquad \mathsf{andRight}\ \frac{\Longrightarrow \phi \qquad \Longrightarrow \psi}{\Longrightarrow \phi \land \psi}$$

$$\mathsf{orRight}\ \frac{\Longrightarrow \phi, \psi}{\Longrightarrow \phi \lor \psi} \qquad \mathsf{orLeft}\ \frac{\phi \Longrightarrow \qquad \psi \Longrightarrow}{\phi \lor \psi \Longrightarrow}$$

$$\mathsf{impRight}\ \frac{\phi \Longrightarrow \psi}{\Longrightarrow \phi \longrightarrow \psi} \qquad \mathsf{impLeft}\ \frac{\Longrightarrow \phi \qquad \psi \Longrightarrow}{\phi \longrightarrow \psi \Longrightarrow}$$

$$\mathsf{notLeft}\ \frac{\Longrightarrow \phi}{\neg \phi \Longrightarrow} \qquad \mathsf{notRight}\ \frac{\phi \Longrightarrow}{\Longrightarrow \neg \phi}$$

$$\mathsf{allRight}\ \frac{\Longrightarrow [x/c](\phi)}{\Longrightarrow \forall x.\phi} \qquad \mathsf{allLeft}\ \frac{\forall x.\phi, [x/t](\phi) \Longrightarrow}{\forall x.\phi \Longrightarrow}$$

with $c : \to A$ a new constant, if $x{:}A$ — with $t \in \mathrm{Trm}_{A'}$ rigid ground, $A' \sqsubseteq A$, if $x{:}A$

$$\mathsf{exLeft}\ \frac{[x/c](\phi) \Longrightarrow}{\exists x.\phi \Longrightarrow} \qquad \mathsf{exRight}\ \frac{\Longrightarrow \exists x.\phi, [x/t](\phi)}{\Longrightarrow \exists x.\phi}$$

with $c : \to A$ a new constant, if $x{:}A$ — with $t \in \mathrm{Trm}_{A'}$ rigid ground, $A' \sqsubseteq A$, if $x{:}A$

$$\mathsf{close}\ \frac{}{\phi \Longrightarrow \phi}$$

$$\mathsf{closeFalse}\ \frac{}{\mathrm{false} \Longrightarrow} \qquad \mathsf{closeTrue}\ \frac{}{\Longrightarrow \mathrm{true}}$$

Figure 2.2. Classical first-order rules

speaking, the lemma can be used). One can also view the cut rule as a case distinction on whether ϕ is true or not as the right premiss is equivalent to $\Longrightarrow \neg\phi$.

Using the cut rule in the right way can greatly reduce the length of proofs. However, since the cut formula ϕ is arbitrary, the cut rule is not analytic and non-deterministic. That is the reason why it is not included in the calculus for first-order logic (it is not needed for completeness). In the KeY system it is only applied interactively when the user can choose a useful cut formula based on their knowledge and intuition.

The cut rule introduces a lemma ϕ that is proved in the particular context in which it is introduced. Thus, it can only be used in that context. It can, for example, not be used in the context of an update $\mathcal{U}$ since ϕ does not imply $\{\mathcal{U}\}\,\phi$. Another way to introduce a lemma is to define a new calculus rule and prove its soundness. That way, a lemma ϕ can be introduced that can be used in any context (provided that ϕ is shown to be logically valid).

2.6.3 Non-program Rules for Modalities

The JAVA CARD DL calculus contains some rules that apply to modal operators and are, thus, not first-order rules but that are neither related to a particular JAVA construct. The most important representatives of this rule class are the following two rules for handling empty modalities:

$$\text{emptyDiamond}\ \frac{\Longrightarrow \phi}{\Longrightarrow \langle\rangle\phi} \qquad \text{emptyBox}\ \frac{\Longrightarrow \phi}{\Longrightarrow [\,]\phi}$$

The rule

$$\text{diamondToBox}\ \frac{\Longrightarrow [p]\phi \qquad \Longrightarrow \langle p\rangle\text{true}}{\Longrightarrow \langle p\rangle\phi}$$

relates the diamond modality to the box modality. It allows to split a total correctness proof into a partial correctness proof and a separate proof for termination. Note that this rule is only sound for deterministic programming languages.

2.6.4 Induction

This following simple Peano induction rule is used both to conclude that a formula holds for all (natural) numbers, and to use that conclusion as an assumption for other proof obligations:

$$\text{natInduction}\ \frac{\begin{array}{l}\Longrightarrow I(0)\\ \Longrightarrow \forall n.\,(I(n) \longrightarrow I(n+1))\\ \forall n.\,I(n) \Longrightarrow\end{array}}{\Longrightarrow}$$

where I is a formula with at most one free variable $n{:}\mathbb{N}$.

It has three premisses: (1) the *base case*, (2) the *step case*, and (3) the *use case*. The formula I is the *induction hypothesis* and n is the *induction variable*. Dynamic Logic makes it possible to use this rule to prove a wide range of program properties, since the induction hypothesis can contain programs.

2.7 Calculus Component 2: Reducing JAVA Programs

2.7.1 The Basic Assignment Rule

In JAVA—like in other object-oriented programming languages—different reference variables can refer to the same object. This phenomenon, called *aliasing*, causes difficulties for handling assignments in a calculus (a similar problem occurs with syntactically different array indices that may refer to the same array element).

For example, whether or not the formula `o1.a=1` still holds after the execution of the assignment "`o2.a = 2;`" depends on whether or not `o1` and `o2` refer to the same object. Therefore, JAVA assignments cannot be symbolically executed by syntactic substitution, as done, for instance, in classic Hoare Logic. Solving this problem naively—by doing a case split—is inefficient and leads to heavy branching of the proof tree.

In the JAVA CARD DL calculus we use a different solution. It is based on the notion of *updates*, which can be seen as "semantic substitutions". Evaluating $\{loc{:=}val\}\phi$ in a state is equivalent to evaluating ϕ in a modified state where *loc* evaluates to *val*, i.e., has been "semantically substituted" with *val*.

The KeY system uses special simplification rules to compute the result of applying an update to terms and formulas that do not contain programs (⇒ Sect. 2.10). Computing the effect of an update to a formula $\langle p\rangle\phi$ is delayed until p has been symbolically executed using other rules of the calculus. Thus, case distinctions are not only delayed but can often be avoided altogether, since (a) updates can be simplified *before* their effect has to be computed, and (b) their effect is computed when a maximal amount of information is available (namely *after* the symbolic execution of the whole program).

The basic assignment rule thus takes the following simple form:

$$\text{assignment}\ \frac{\Longrightarrow \{loc{:=}val\}\langle \pi\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \ loc\ \texttt{=}\ val\texttt{;}\ \ \omega\rangle\phi}$$

That is, it just turns the assignment into an update. Of course, this does not solve the problem of computing the effect of the assignment. This problem is postponed and solved later by the rules for simplifying updates.

Furthermore—and this is important—this "trivial" assignment rule is correct only if the expressions *loc* and *val* satisfy certain restrictions. The rule is only applicable if neither the evaluation of *loc* nor that of *val* can cause any side effects. Otherwise, other rules have to be applied first to analyze *loc* and *val*, check for possible exceptions, etc. For example, these other rules would replace the formula $\langle\texttt{x = ++i;}\rangle\phi$ with $\langle\texttt{i = i+1; x = i;}\rangle\phi$, before the assignment rule can be applied to derive first $\{\texttt{i}{:=}\texttt{i+1}\}\langle\texttt{x = i;}\rangle\phi$ and then $\{\texttt{i}{:=}\texttt{i+1}\}\{\texttt{x}{:=}\texttt{i}\}\langle\rangle\phi$. These rules are presented in the KeY book and we do not show them here.

2.7.2 Rules for Conditionals

Most `if-else` statements have a non-simple expression (i.e., one with potential side-effects) as their condition. In this case, we unfold it in the usual manner first. This is achieved by the rule

$$\text{ifElseUnfold}\ \frac{\Longrightarrow \langle \pi\ \texttt{boolean}\ v\ \texttt{=}\ nse\texttt{;}\ \texttt{if}\ \texttt{(}v\texttt{)}\ p\ \texttt{else}\ q\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{if}\ \texttt{(}nse\texttt{)}\ p\ \texttt{else}\ q\ \ \omega\rangle\phi}$$

where v is a fresh boolean variable.

After dealing with the non-simple condition, we will eventually get back to the `if-else` statement, this time with the condition being a variable and, thus, a simple expression. Now it is time to take on the case distinction inherent in the statement. That can be done using the following rule:

$$\mathsf{ifElseSplit}\ \frac{\begin{array}{c} se = \mathrm{TRUE} \Longrightarrow \langle \pi\ p\ \ \omega\rangle\phi \\ se = \mathrm{FALSE} \Longrightarrow \langle \pi\ q\ \ \omega\rangle\phi \end{array}}{\Longrightarrow \langle \pi\ \texttt{if (}se\texttt{)}\ p\ \texttt{else}\ q\ \ \omega\rangle\phi}$$

While perfectly functional, this rule has several drawbacks. First, it unconditionally splits the proof, even in the presence of additional information. However, the program or the sequent may contain the explicit knowledge that the condition is true (or false). In that case, we want to avoid the proof split altogether. Second, after the split, the condition *se* appears on both branches, and we then have to reason about the same formula twice.

A better solution is the following rule that translates a program with an `if-else` statement into a conditional formula:

$$\mathsf{ifElse}\ \frac{\Longrightarrow \mathrm{if}(se = \mathrm{TRUE})\ \mathrm{then}\ \langle \pi\ p\ \ \omega\rangle\phi\ \mathrm{else}\ \langle \pi\ q\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{if (}se\texttt{)}\ p\ \texttt{else}\ q\ \ \omega\rangle\phi}$$

Note that the if-then-else in the premiss of the rule is a logical and not a program language construct (⇒ Def. 2.8).

The ifElse rule solves the problems of the ifElseSplit rule described above. The condition *se* only has to be considered once. And if additional information about its truth value is available, splitting the proof can be avoided. If no such information is available, however, it is still possible to replace the propositional if-then-else operator with its definition, resulting in

$$(se = \mathrm{TRUE}) \longrightarrow \langle \pi\ p\ \ \omega\rangle\phi \quad \wedge \quad (se \neq \mathrm{TRUE}) \longrightarrow \langle \pi\ q\ \ \omega\rangle\phi$$

and carry out a case distinction in the usual manner.

A problem that the above rule does not eliminate is the duplication of the code part ω. Its double appearance in the premiss means that we may have to reason about the same piece of code twice. Leino [2005] proposes a solution for this problem within a verification condition generator system. However, to preserve the advantages of a symbolic execution, the KeY system here sacrifices some efficiency for the sake of usability. Fortunately, this issue is hardly ever limiting in practice.

The rule for the `switch` statement, which also is conditional and leads to case distinctions in proofs, is not shown here. It transforms a `switch` statement into a sequence of `if` statements.

2.7.3 Unwinding Loops

The following rule "unwinds" `while` loops. Its application is the prerequisite for symbolically executing the loop body. Unfortunately, just unwinding a loop repeatedly is only sufficient for its verification if the number of loop iterations has a known upper bound.

If the number of loop iterations is not bounded, the loop has to be verified using either induction (⇒ Sect. 2.6.4) or an invariant rule (⇒ Sect. 2.8). If induction is used, the unwind rule is also needed as the loop has to be unwound once in the step case of the induction.

In case the loop body does not contain `break` or `continue` statements (which is the common case), the following simple version of the unwind rule can be applied:

$$\text{loopUnwind}\ \frac{\Longrightarrow \langle \pi\ \texttt{if}\ (e)\ \{\ p\ \texttt{while}\ (e)\ p\ \}\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{while}\ (e)\ p\ \omega\rangle\phi}$$

2.7.4 Replacing Method Calls by their Implementation

Symbolic execution deals with method invocations by syntactically replacing the call by the called implementation (verification via contracts is described in Sect. 2.9). To obtain an efficient calculus we have conservatively extended the programming language with two additional constructs: a method body statement, which allows us to precisely identify an implementation, and a `method-frame` block, which records the receiver of the invocation result and marks the boundaries of the inlined implementation.

2.7.5 Instance Creation and Initialization

JAVA CARD DL, like many modal logics, operates under the technically useful constant domain semantics (all program states have the same universe). This means, however, that all instances that are ever created in a program have to exist a priori. To resolve this seeming paradox, we use ghost fields that allow to change and query the program-visible instance state (created, initialized, etc.).

To handle instance initialization, we use an approach that is based on program transformation. The transformation reduces a JAVA program p to a program p' such that the behavior of p (with initialization) is the same as that of p' when initialization is disregarded. This is done by inserting code into p that explicitly executes the initialization.

2.7.6 Handling Abrupt Termination

In JAVA, the execution of a statement can terminate *abruptly* (besides terminating normally and not terminating at all). Possible reasons for an abrupt termination are (a) that an exception has been thrown, (b) that a loop or a `switch` statement is terminated with `break`, (c) that a single loop iteration is terminated with the `continue` statement, and (d) that the execution of a method is terminated with a `return` statement. Abrupt termination of a statement either leads to a redirection of the control flow after which the program execution resumes (for example if the exception is caught), or the whole program terminates abruptly (if the exception is not caught).

If the Whole Program Terminates Abruptly

In Java Card DL, an *abruptly* terminating statement—where the abrupt termination does not just change the control flow but actually terminates the whole program p in a modal operator $\langle p\rangle$ or $[p]$—has the same semantics as a *non-terminating* statement (Def. 2.13). For that case rules such as the following are provided in the Java Card DL calculus for all abruptly terminating statements:

$$\text{throwDiamond}\ \frac{\Longrightarrow \text{false}}{\Longrightarrow \langle \texttt{throw}\ se\texttt{;}\ \omega\rangle\phi} \qquad \text{throwBox}\ \frac{\Longrightarrow \text{true}}{\Longrightarrow [\texttt{throw}\ se\texttt{;}\ \omega]\phi}$$

Note that in these rules, there is no inactive prefix π in front of the `throw` statement. Such a π could contain a `try` with accompanying `catch` clause that would catch the thrown exception. However, the rules throwDiamond, throwBox etc. must only be applied to uncaught exceptions. If there is a prefix π, other rules described below must be applied first.

If the Control Flow is Redirected

The case where an abruptly terminating statement does not terminate the whole program in the modal operator but only changes the control flow is more difficult to handle. The basic idea for handling this case in the calculus is to execute the change in control flow symbolically by syntactically rearranging the affected program parts. The calculus rules have to consider the different combinations of prefix-context (beginning of a block, method-frame, or `try`) and abruptly terminating statement (`break`, `continue`, `return`, or `throw`). We omit these rules here.

2.8 Calculus Component 3: Invariant Rule for Loops

There are two techniques for handling loops in KeY: induction and using an invariant rule. For the sake of clarity, we show here a "classical" invariant rule. In particular, we assume that there is no abrupt termination and that loop guard expressions do not have side-effects. In reality, the KeY calculus implements a much more involved version of the rule.

$$\text{invRuleClassical}\ \frac{\begin{array}{c}\Gamma \Longrightarrow \mathcal{U}Inv,\ \Delta \\ Inv,\ \ se \Longrightarrow [p]Inv \\ Inv,\ \neg se \Longrightarrow \phi\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\texttt{while (}se\texttt{) \{ }p\texttt{ \}}]\phi,\ \Delta}\ (*)$$

This rule states that, if one can find a formula *Inv* such that the three premisses hold requiring that

(a) *Inv* holds in the beginning,
(b) *Inv* is indeed an invariant, and

(c) the conclusion ϕ follows from *Inv* and the negated loop condition $\neg se$,

then ϕ holds after executing the loop (provided it terminates).

Remember that the symbol $(*)$ in the rule schema means, that the context $\Gamma, \Delta, \mathcal{U}$ must be empty unless its presence is stated explicitly (as in the first premiss), i.e., only instances of the schema itself are inference rules.

2.9 Calculus Component 4: Using Method Contracts

There are basically two possibilities to deal with method calls in program verification: inlining the body of the invoked method (⇒ Sect. 2.7.4) or using the specification (which then, of course, has to be verified). The specification of a method is called *method contract* and is defined as follows.

Definition 2.34 (Method contract). A method contract for a method or constructor *op* declared in a class or interface $C \in P$ is a quadruple

$$(Pre, Post, Mod, term)$$

where:

- $Pre \in Formulas$ is the precondition that may contain the following program variables:
  - *self* for the receiver object (the object which a caller invokes the method on); if *op* refers to a static method or a constructor the receiver object variable is not allowed;
  - $p_1 \ldots, p_n$ for the parameters.
- $Post \in Formulas$ is the postcondition of the form

$$(exc \doteq \texttt{null} \longrightarrow \phi) \wedge (exc \neg \doteq \texttt{null} \longrightarrow \psi)$$

  where ϕ is the postcondition for the case that the method terminates normally and ψ specifies the case where the method terminates abruptly with an exception. The formulas ϕ and ψ may contain the following program variables:
  - *self* for the receiver object; again the receiver object variable is not allowed for static methods;
  - $p_1, \ldots, p_n$ for the parameters;
  - *result* for the returned value;
- *Mod* is a modifier set for the method, i.e., an upper bound on state changed by *op*.
- The termination marker *term* is an element from the set $\{partial, total\}$; the marker is set to *total* if and only if the method contract requires the method or constructor to terminate, otherwise *term* is set to *partial*. ◁

The formulas *Pre* and *Post* are JAVA CARD DL formulas. However, in most cases they do not contain modal operators. This is in particular true if they are automatically generated translations of JML or OCL specifications.

The KeY calculus contains rules both for replacing method invocations with contracts and establishing their correctness.

2.10 Calculus Component 5: Update Simplification

The process of update simplification comprises (a) update normalisation and (b) update application. Update normalisation transforms single updates into a certain normal form, while update application computes the effect of applying an update to a term, a formula, or another update. Note that both normalisation and application of updates is done automatically; there are no interactive rules for that purpose.

We do not give the rules for updates here. We just note that updates can be distributed over logical operators (except modal operators) as (a) the semantics of logical operators is not affected by a state change (b) the state change affected by an update is deterministic. The application of an update u to a formula with a modal operator, such as $\{u\}\,\langle p\rangle\phi$ and $\{u\}\,[p]\phi$, cannot be simplified any further. In such a situation, instead of using update simplification, the program p must be handled first by symbolic execution. Only when the whole program has disappeared, the resulting updates can be applied to the formula ϕ.

Example 2.35 (Update application). As an example, consider the term

$$\{\mathtt{a(o)}{:=}\mathtt{t}\}\,\mathtt{a(p)}\ .$$

The update $\mathtt{a(o)}{:=}\mathtt{t}$ affects the term $\mathtt{a(p)}$ iff $\mathtt{o}$ and $\mathtt{p}$ evaluate to the same domain element. In this case, the result is $\mathtt{t}$, otherwise the update is simply propagated giving $a(\{\mathtt{a(o)}{:=}\mathtt{t}\}\,\mathtt{p})$. The latter simplifies to $\mathtt{p}$, since it can be excluded syntactically that an update to a unary function $\mathtt{a}$ can affect the non-rigid nullary function $\mathtt{p}$.

Thus, the result of applying the update in the original term is the conditional term

$$\text{if } \mathtt{p}=\mathtt{o} \text{ then } \mathtt{t} \text{ else } \mathtt{a(p)}\ ,$$

which coincides with our intuition. ◁

Part II

A Novel Approach to Verification of Multi-threaded JAVA Programs

Typographic Conventions

To keep the introductory material compatible with [Beckert et al., 2007] on one hand, but to use "natural" notation in the Part dedicated to multi-threading on the other hand, we in the following overload certain symbols. From now on:

α	denotes in MODL a part of the program before the emphasized position. Was: typing function (⇒ Def. 2.3).
π_i	denotes in MODL the thread choice function ("permutation") at position i. Was: the non-active prefix of a statement sequence (⇒ Sect. 2.5.4). MODL generalizes the non-active prefix to the irrelevant program part α (above).
$\mathcal{T}$	denotes in MODL the carrier set of thread ids. Was: the set of types in a type hierarchy (⇒ Def. 2.1).
$\langle\cdot\rangle$	denotes in MODL the concurrent diamond modality. Was: the sequential JAVA CARD modality (⇒ Def. 2.8). Sequential diamond modality in MODL is denoted by $\langle\underline{\cdot}\rangle$. The same applies for the box modality $[\cdot]$.

3

Introducing MODL—A Multi-threaded Object-oriented Dynamic Logic

Multithreading is just one damn thing after, before, or simultaneous with another.

Andrei Alexandrescu

3.1 Main Idea of the Proposed Logic and Proof System

Our aim has been to design a program logic that

- reflects the properties of Java concurrency in an intuitive manner
- has a sound and (relatively) complete calculus
- employs only sound and transparent abstractions
- poses no bounds on the state space or thread number
- allows reasoning about properties of the scheduler, but does not require such reasoning for program verification.

In parallel to Object-oriented Dynamic Logic (ODL) [Beckert and Platzer, 2006b], which captures the essence of object-orientation in a small language, we have called our logic MODL—Multi-threaded Object-oriented Dynamic Logic.

The logic MODL

Unsurprisingly, MODL is a close relative of Java Card DL, the sequential KeY logic. It has the familiar modal operators $\langle p\rangle\phi$ and $[p]\phi$ referring, this time, to the total and partial correctness of a multi-threaded program p. The biggest difference lies in the programs: multi-threaded programs require a different representation than sequential ones. Conceptually, we follow the CFG-style program model of Keller [1976], who has defined "parallel programs" as

> a bipartite directed graph, the nodes of which are divided into
> - place nodes: representing points at which an instruction pointer of a processor may dwell,

- transitions nodes: representing a class of transitions, each denoting an event which corresponds to the execution of a particular instruction.

In our case, the role of place nodes is played by set-valued control variables, which are part of the state and contain thread ids (collectively we also call them a thread configuration). The transition nodes are given by Java-like statements, which appear as "program text" inside the modal operator (⇒ Fig. 3.1).

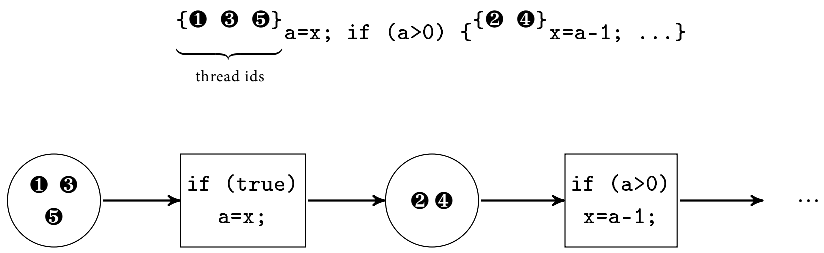


Figure 3.1. A textual and a graph representation of a multi-threaded program (together with thread configuration). The exact definitions are given in Section 4.1

Execution of a program corresponds to the movement of thread id "tokens", while the program text remains unaltered. The movement is accompanied by a corresponding change in data state. It is clear that programs can behave differently depending on the thread scheduling. The natural question is how to model the scheduler?

With a purely indeterministic scheduling, we have no choice but to perform (a prohibitively large number of) case distinctions in the calculus. Unsightly meta-level efforts would then be necessary to prune the proof search space and get a grip on the complexity. Instead, we opt for an underspecified deterministic scheduler, and express its decisions explicitly on the object level by means of a partially specified scheduling function.

Such a design gives our concurrent programs (surprisingly maybe) a deterministic semantics, just as is the case with sequential Java programs (⇒ Sect. 2.2). The main advantage is the much stronger control over granularity of reasoning. We can tackle simple problems with relatively little effort, but still have the power to get into the "gory details" for demanding cases. Furthermore, we retain beneficial logical properties, like $\langle p\rangle\phi \longrightarrow [p]\phi$.

A calculus for MODL

To prove theorems of MODL, we have developed a sequent-style calculus. The calculus performs symbolic execution of programs—a method, which goes back to [King, 1976] and ensures good understandability of the process for the user. As far as we know, this work is the first application of symbolic execution to full functional verification of multi-threaded programs.

In principle, the calculus explores the behavior of a concurrent program by building all possible thread interleavings. Done naively, such an approach is doomed to failure due to state explosion; it is also inapplicable to systems with an unbounded number of threads. Our calculus can effectively perform such exploration by employing symmetry reductions that merge many interleavings that are not significantly different. This is efficiently possible for the considered language fragment and produces a feasible number of cases (even in presence of unbounded concurrency). Further efficiency gains are possible from appropriate program and proof modularization.

By means of symbolic execution, the calculus reduces assertions about programs to assertions about data types and permutations, which encapsulate the scheduler decisions and hide symmetric schedulings. In the desirable case that the program is scheduling-independent[1] the permutations can be removed from the correctness assertions by application of standard algebraic lemmas. When also the remaining assertions (now without permutations) can be discharged, then the program is fully correct w.r.t. its functional specification.

Plan of attack

Chapter	Content
this chapter	continues with a discussion of which features of Java concurrency are supported and surveys related work.
Chapter 4	defines the basic version of MODL, introducing the concepts of threads, deterministic scheduling and thread-local data.
Chapter 5	refines the basic version of the logic with a more verification-friendly scheduler formulation. The refined scheduler model avoids explicit thread enumeration, allows unbounded thread configurations and symmetry reduction.
Chapter 6	presents the symbolic execution calculus used for verification, describes how Java programs are normalized ("unfolded") before verifying.
Chapter 7	shows further extensions and refinements: how to prove atomicity with invariants, verify condition variables, establishing program correctness w.r.t. the Java Memory Model; discusses future work.
Chapter 8	describes the implementation of the calculus in the KeY system and presents case studies.

[1] Scheduling independence means here that the program's final state always satisfies the specification, in spite of possibly different intermediate states taken in different runs.

3.2 Modeling JAVA Concurrency

Threads and Shared Memory

Concurrent programming in JAVA is based on shared-memory multi-threading. A thread is a single flow of control that can execute program instructions independently of other threads. A thread, in essence, consists of a program counter and a call stack. Threads can exchange data via references to the same objects on the shared heap.

In JAVA, threads are created and (to some extent) controlled via instances of the `java.lang.Thread` class. Such instances can be obtained in two ways: (1) by declaring and instantiating a class that extends `Thread` or (2) by passing a `Runnable` instance to the standard constructor of the `Thread` class. Typical code for creating and starting a thread looks like this:

```
Thread t = new MyThread();
t.start();
// run() method of MyThread executes asynchronously now
```
JAVA

The use of objects to create and control threads sometimes obnubilates the fact that threads and objects are, actually, two orthogonal concepts. JAVA objects are mostly passive data entities coming to life when threads execute their methods. On the other hand, objects have only limited means to prevent undesired access.

No thread identities in programs

In MODL threads are currently identified not with `java.lang.Thread` instances, but with elements (thread ids or tids) of a not further structured type Thread. Currently, we do not support thread identities in programs. This means that the programmer may not make use of the reference `t` shown in the listing above.

It is, thus, not allowed to invoke thread-controlling methods of the `Thread` instance, the most important being `t.interrupt()` and `t.join()`. We believe that this limitation prevents us from verifying only a small fraction of interesting code. In particular, it does not forbid the use of synchronized blocks or condition variables with `wait()/notify()`.

Furthermore, we conjecture that it is possible to extend our logic and calculus with thread identities in programs, since thread identities are completely exposed through the scheduler function. In this case we would indeed identify the type Thread with `java.lang.Thread`. In general, using thread identities in programs breaks thread symmetry and would degrade the performance of the proof system. This approach may still be useful in certain cases though.

No dynamic thread creation (but unbounded multi-threading)

The only thread creation mechanism we currently provide is a possibility to specify the initial thread configuration of a program (together with the initial local variable assignment). Note that the configuration values can be symbolic ("k threads"). While this limitation is indeed unfortunate, it does not impair the usefulness of the calculus much. It is in the nature of concurrent Java applications that most objects are passive entities. They are unaware of thread creation and can (and indeed have to) be verified for an arbitrary number of threads accessing them. The most prominent expression of this fact is library code, which has to be thread-safe for any number of client threads.

Sequential Coverage

On the sequential side, we benefit from the 100% Java Card coverage of the KeY calculus. This includes full support for dynamic object creation (with static initialization), efficient aliasing treatment, Java-faithful arithmetics, etc. All of these features can be used in verification of concurrent programs.

Exceptions cannot be caught

One area where there is currently a gap between the concurrent and the sequential calculus is exception handling. The concurrent proof system is sound but incomplete in this regard. Exceptions are always detected, but once thrown they cannot be caught. The calculus treats the whole program as non-terminating in this case. A possible approach to overcoming this limitation is sketched in the section on future work (⇒ Sect. 7.4).

No non-atomic loops

Finally, we require all loops to be atomic. The programmer has to ensure that no (significant) interleavings occur while the loop runs. This property can be checked by our method as described later on (⇒ Sect. 7.1). An approach for working around this limitation as well as some remarks about developing a more elaborated model of the scheduler that does not have this restriction are given in the section on future work (⇒ Sect. 7.4).

Mutual Exclusion

Mutual exclusion of threads in critical regions is achieved by means of synchronized methods and blocks. Every such block includes a reference to a locked object (for synchronized instance methods it is the object referenced by `this`, for static methods—the class object). Locks are binary semaphores, which can be acquired or released by a thread. Every object has one such lock. At most one thread can possess any given lock at the same time. Threads trying to enter a synchronized block where the lock is held by another thread are blocked until the lock becomes available.

Locks can only be acquired and released in block-structured manner, meaning that when the control flow leaves a synchronized block—whether normally or abruptly—the involved lock is automatically released. Locks are also reentrant: if a thread already possesses a certain lock, a repeated acquisition of the same lock succeeds immediately. In this case we say that the lock depth has increased.

Declaring a method synchronized does not mean that it becomes atomic, i.e., free from harmful interference. Atomicity is only guaranteed if all threads in the system potentially accessing the same data acquire the same lock(s). Threads not adhering to an appropriate locking protocol can observe inconsistent state or perform harmful updates, destroying the assumptions of other threads.

A particular problem with synchronized blocks is that the value of the lock expression, which may be as simple as a field

```
synchronized(lock) {...}
```

or as complex as a method

```
synchronized(lock()) {...}
```

can change over time. Threads seeing different values of the lock expression in this case are no longer guaranteed mutual exclusion. This subtle issue is a source of hard-to-find errors.

The main problem with locking—or, more general, with JAVA concurrency—is that it is a major source of non-compositionality. There is no single point where a correct policy for accessing a shared resource is fixed in JAVA. Each thread must voluntarily adhere to the programmer-designed locking protocol in order for the whole application to be correct.

Modeling locking in our programming language

To make lock acquisition and release explicit, we extend the `Object` class with two "ghost" methods:

1. `public void <lock>()`
2. `public void <unlock>()` .

Code marked as synchronized is automatically surrounded by invocations of these methods during the unfolding stage (⇒ Sect. 6.2). To keep track of locking state we also declare two ghost fields per object:

1. `<lockedby>` of type tid (identity of the thread holding the object's lock)
2. `int <lockcount>` (locking depth).

Condition Variables

An important feature of JAVA's concurrency mechanism is condition variables. It allows threads to suspend execution until an external signal is received. The signaling does not involve thread identities, but works via a shared reference to an arbitrary object.

The waiting thread must acquire the object's lock first. Calling `wait()` on the object releases the lock and suspends thread execution. When a wake-up signal is received, the thread leaves the suspended state but does not yet continue execution. It must now compete for the acquisition of the lock with other threads. When it succeeds, the depth of the lock is restored to the state before the wait.

The notifying thread must possess the object lock as well. Sending a wake-up signal to one (randomly chosen) suspended thread requires calling `notify()` on the corresponding object. Waking up all threads waiting is possible by calling `notifyAll()`. Again, the waiting threads will be able to proceed *in the earliest* when the notifying thread has released the lock.

Since other threads can intervene and destroy the condition between the wake-up signal and lock re-acquisition (a phenomenon known as "barging"), it is in most cases compulsory to re-test the condition upon wake up and return to the suspended state if it is not satisfied. This practice is advocated by all programming guidelines and followed by most of the programs.

The Java Memory Model

The Java Memory Model (JMM) is a part of the Java Language Specification, which describes how threads interact via shared memory. Many programmers assume that Java multi-threading operates under an intuitive, sequentially consistent memory model. Sequential consistency [Lamport, 1979] means that updates to shared state are immediately visible to all threads, and concurrent program behavior can be described by thread interleavings. In reality, the Java Memory Model provides much weaker guarantees: updates to shared state performed by one thread need not become immediately visible to other threads. Even worse, updates may become visible to other threads in an order different from the one in which they have been carried out.

The JMM has undergone greater revisions within [JSR-133]. The latest, most comprehensive accounts from the responsible authors are [Manson, 2004; Manson et al., 2005a]. In them the JMM designers make three promises to the users:

1. A promise for programmers. Programs without data races (also known as properly or fully synchronized programs) shall have sequentially consistent semantics. This is also known as the DRF guarantee.
2. A promise of security. Programs *with* data races shall still enjoy certain minimal security guarantees. The JMM promises that a program with a data race will never divulge—due to the race alone—potentially sensitive information contained in program parts unrelated to the race.
   In other words, the JMM promises that variables can only assume values that are in some sense "justifiable" by the program at hand. Unjustifiable out-of-thin-air values (OoTA), which could breach security, should be prevented. What constitutes an OoTA value is a controversial issue and is currently specified by means of an example catalog.
   To keep this promise the JMM defines a (complicated) policy as to what constitutes an allowed behavior in presence of a data race.
3. A promise for compilers. Common compiler and VM optimizations shall be allowed.

Current research [Aspinall and Ševčík, 2007; Huisman and Petri, 2007] shows that the latest JMM formulation still suffers from quite severe deficiencies. In particular, the promises 2 and 3 are not fulfilled. The promise 1, in contrast, is fulfilled and it is consensus in the field that the only way programmers can achieve well-defined program behavior is by staying within the fully-synchronized fragment of Java. Calculus rules for checking this in our proof system are presented in Section 7.3.

Finalization and Other Concurrency Primitives

Among the things that we do not consider is finalization, even though finalizers introduce concurrency into an application. The use of finalizers is further complicated by intricate interactions with the Java Memory Model. Experts estimate that most uses of finalizers in practice are subtly incorrect [Boehm, 2005]. Still, in our logic, we have to disregard finalization as we do not model garbage collection.

Furthermore, since our logic lacks any notion of time, we do not treat primitives that involve timing, such as `wait(long millisecs)`.

The atomicity of assignments to non-volatile `long` or `double` variables is implementation-specific according to the JLS. A JVM is allowed to implement a single write to such a variable as two separate writes: one to each 32-bit half. For this reason, we currently demand that all `long` or `double` variables are declared volatile.

3.3 Related Work

Westheimer's Discovery: A couple of months in the laboratory can frequently save a couple of hours in the library.

Classical approaches to deductive verification of concurrent programs

One of the first deductive verification methods was the partial correctness proof method of Ashcroft [1975] and Keller [1976], incorporating a CFG-like program formalism and an induction principle. The principle is to show that every atomic statement preserves a global invariant. Of course, such global invariants can quickly become unwieldy without modularization. Nonetheless, these early works contain many seminal insights into the inner working of concurrent programs

Another classical method is due to Owicki and Gries [1976b] and builds on Hoare Logic for sequential programs. The method combines a proof of local (i.e., sequential) correctness with a non-interference check. The latter establishes that assumptions used throughout the proof of local correctness are not destroyed if the scheduler chooses to interleave execution with other threads. This leads to proof size that is quadratic in the number of statements. The method is not compositional. We have implemented an Owicki-Gries-style proof system for a fragment of JAVA in KeY [Klebanov, 2004]. Further modern adaptations of the method are described in the next section.

A revolutionary step towards compositional verification of concurrent programs was the rely-guarantee method of Jones [1981]. The method introduces for each thread two predicates: *rely* and *guarantee*. In contrast to assertions or postconditions these predicates range not over states but over pairs of consecutive states. The proof method consists in showing that every step of a thread satisfies its guarantee obligation assuming that every step of the environment satisfies the rely assumption. The rely assumption in its turn is composed from the guarantee obligations of other threads. The method is compositional and the proof size is linear in the number of threads. The difficulty resides in summarizing the behavior of a thread in one transitive predicate.

Deductive verification of multi-threaded JAVA programs

Several deductive calculi for (different fragments of) sequential Java exist [Jacobs and Poll, 2001c; Poetzsch-Heffter and Müller, 1999b; von Oheimb, 2001a; Zee et al., 2008; Marché et al., 2004]. In contrast, the only implemented deductive verification system for multi-threaded JAVA existing to date is—to our knowledge—Verger [Ábrahám et al., 2005]. The calculus is an adaptation of the Owicki-Gries method to JAVA, incorporating a proof method for CSP in order to reason about method calls as message passing. The system generates verification conditions from programs augmented with auxiliary variables and annotated with Hoare-style assertions. The verification conditions are subsequently discharged in PVS. The system has a good concurrent language coverage.

A recent and more accessible formulation is [de Boer, 2007], which replaces the CSP calculus with proof theory of recursive procedures.

Separation Logic is another extension of Hoare Logic with operators for reasoning about resource access, which allows for greater modularity of reasoning. The logic has also been extended to handle JAVA and concurrency, and the latest development is a "marriage" of rely-guarantee and Separation Logic [Vafeiadis and Parkinson, 2007]. The latter promises better modularity in dealing with rely and guarantee predicates.

Temporal logics

A huge body of work is available on verifying temporal properties of concurrent software. This includes model checkers and even deductive proof systems (e.g., by Manna and Pnueli [1991]). In contrast to using temporal logic, though, a proof system for Dynamic Logic allows functional verification, i.e., full reasoning about data. This way verification tasks can be tackled where not only safety or liveness but the input-output relation of a concurrent program is of interest.

Concurrent Dynamic Logic

The only Dynamic Logic for a programming language incorporating concurrency is—to our knowledge—the Concurrent Dynamic Logic (CDL) by Peleg [1987b]. He notes, however, that this particular logic "suffers from the absence of any communication mechanisms; processes of CDL are totally independent and mutually ignorant". Peleg [1987a] gives two extensions of CDL with interprocess communication: one with channels and one with shared variables. In both works cited, the focus is on studying concerns of expressivity and decidability of the logics (communication renders the logic highly undecidable). The issue of a calculus or program verification in general is not touched.

Model checkers

Verification of *concurrent* systems has traditionally been—with a few exceptions—the domain of model checking tools. This holds also for Java program verification, where several very successful model checking frameworks have been established. Prominent model checkers for Java programs are Bogor [Robby et al., 2003b] and Java PathFinder [Havelund and Pressburger, 2000a].

These tools can check not only temporal but also functional properties. They employ very clever optimizations (abstractions) and can thus verify programs of substantial size. Many of these abstractions—like symmetry reduction—are sound and do not come at the price of missed errors. Still, to guarantee termination of the model checking process, a finite system model is required. Most of the time, this is achieved by unsound abstraction, such as bounding the length of explored executions, number of threads, number of loop iterations, size of initial heap configurations, etc. In this setup model checking is very useful for detecting bugs, but provides no indication of correct behavior under all circumstances.

A sound way to overcome the finite-model barrier is to use abstraction refinement. Counter-Example-Guided Abstraction Refinement (CEGAR) is a relatively recent technique that does so. It allows checking strong properties but must resort to iterated manual model refinement in order to eliminate spurious counter-examples appearing due to overapproximation. While CEGAR has been successfully used in the verification of sequential C programs, to our knowledge, this technique has not been applied to verification of programs in Java-like languages.

A comprehensive control flow model of Java concurrency is given in [Delzanno et al., 2002]. The authors use a variant of Petri nets to model the control flow of concurrent programs. The nets are specifically tailored to treat the "partially non-blocking rendez-vous" nature of Java's `wait()`/`notify()` mechanism. The authors do not perform functional verification but have built a model checker that can check safety properties expressed in terms of control flow. Their Petri net representation is conceptually close to ours, though we use full programs as transitions.

Yahav [2001] describes a system for verifying safety properties of multi-threaded Java-like programs. The system (implemented in the TVLA framework) is an instance of symbolic on-the-fly model checking, where first-order logical structures are used to represent states of the program. It can cope with an unbounded number of allocated objects by building conservative abstract descriptions of (multiple) program states in 3-valued logic. Also, in the above paper, symmetry reduction is mentioned and the author reports having obtained interesting results for an unbounded number of threads.

Static verifiers

Another broad category of verification systems for concurrent programs are static verifiers. Static verifiers are tools that can automatically check program properties by sufficiently approximating the program semantics. Many static verifiers allow the users to improve the quality of the approximation by adding annotations to the code.

Per design, static verifiers are not geared towards input-output reasoning. They are—in most cases—also neither sound nor complete. Still, such tools are very useful for automated detection of concurrency-related problems in practice. There is also a big potential in combining static verification systems with systems for full-functional verification.

A prominent representative of this class of tools is ESC/Java [Flanagan et al., 2002], an extended static checker for many types of properties. On the concurrency side this includes inter-thread escape analysis, race condition detection, deadlock detection, etc. There are also a number of dedicated static analysis tools for race condition detection. One of them is Houdini/rcc [Abadi et al., 2006], which is based on an elaborate type system.

Such tools are aimed to check that access to object fields is guarded by locks and that all threads adhere to a consistent locking policy. This check can be easy if the object fields are protected by the lock associated with the object itself or a dedicated object referenced by a final static field. More elaborated locking schemes might require user annotation or are beyond the scope of the tools.

A class of its own in this category is the SPEC# system, which (in its derivative SpecLeuven) incorporates a "static verifier" for a concurrent object-oriented language [Jacobs et al., 2006]. For one, verification with SPEC# guarantees the absence of data races and deadlocks. It also guarantees compliance of the program with programmer-provided method contracts and object invariants. The approach is sound but not complete.

A very interesting body of research has been produced by Greenhouse and Scherlis [2002b]. The authors have developed an annotation language to specify many important aspect of multi-threaded programs together with a tool suite to statically check them. The annotations include:

- effects (an upper bound on state a method reads and writes)
- aliasing intent. Unaliased data can be reasoned about sequentially
- locking intent. Programmers can associate locks with regions of state; the tool verifies that state is accessed only when the appropriate lock is held. Programmers can also declare that a method requires that a particular lock be held by the caller
- concurrency policy. Programmers can specify methods that can be safely executed concurrently.

The authors also make it plausible that for lock-based programs, concurrency policy combined with models of locking intent is a suitable surrogate for representation invariants.

Alternatives to multi-threading

There are countless models for concurrent computation. We would like to mention some alternatives to multi-threading that are interesting in our context.

One approach trying to overcome the difficulties inherent to JAVA threads is JCSP [Welch et al., 2007; Welch and Austin]. JCSP is a JAVA library for concurrent programming by means of Communicating Sequential Processes (CSP). CSP is a process algebra developed by Hoare [1985]. It has a precise and—in contrast to JAVA threads—compositional semantics. The programmer can reap these benefits by implementing the sequential process parts in regular JAVA and composing them concurrently using the CSP operators provided by the JCSP library. A calculus for verifying JCSP programs has been developed and implemented in the KeY system by Philipp Rümmer [Klebanov et al., 2005].

Some of the most massively concurrent applications available today are programmed in Erlang. Erlang is a functional programming language designed at the Ericsson Computer Science Laboratory and popular in telecommunications. Industrial Erlang programs may contain thousands of processes communicating by message passing. A deductive Erlang Verification Tool (EVT) based on modal μ-calculus was built by Arts et al. [2003].

Concurrent objects offer an alternative concurrency model for object-orientation, which has advantages over multi-threading, especially in highly parallel or distributed architectures. Two examples of languages for programming with concurrent objects are SCOOP [Arslan et al., 2006] and Creol [Johnsen et al., 2006]. A verification calculus for Creol in KeY is currently being incepted.

Today, multi-threading remains the predominant concurrency programming paradigm in spite of its problems. Seriously facing its issues, though, the only prudent advice to application developers is to avoid rolling own multi-threaded solutions whenever possible. If concurrency is necessary, it is recommended to rely on patterns and architectures developed by experts (and potentially verified), such as those in the `java.util.concurrent` package of the standard JAVA library.

4

MODL—A General but Overly Concrete Version

We start with a very general formalism which is quite close to the "machine" semantics. The usefulness of this logic is not so much in its suitability for verification (this will be addressed in the subsequent chapter), but in formalizing basic concepts of thread-based concurrency. We define the syntax and semantics of a multi-threaded JAVA-like programming language and a Dynamic Logic for reasoning about it. Along the way we introduce such concepts as thread configurations, shared and thread-local data, and a deterministic scheduler model.

4.1 Syntax of MODL

4.1.1 Threads and Multi-threaded Programs

The concurrent programming language that we consider is very close to a fragment of multi-threaded JAVA. Its basic constructs are assignments, if-then-else statements, while-loops, JAVA-like concurrency primitives, but also atomic blocks. Several threads can execute a program concurrently. Thus, in contrast to the sequential programs in KeY, a concurrent program is a passive template "without life", until a thread configuration is added. A thread configuration is a part of the program state describing which threads are executing the program. Threads are given a unique identifier, conventionally called *thread id* (tid), which is a term of type Thread; they are in fact identified with this identifier. In the following, we will denote $\mathcal{D}^{\text{Thread}}$, the carrier set of Thread, as $\mathcal{T}$.

In addition to concurrent programs, we also use sequential MODL programs. A sequential program is, roughly, a concurrent program executed by a single thread. The executing thread is explicitly identified in thread-local variables of the program. This explicit thread identifier is also the major difference between sequential MODL programs and sequential programs of JAVA CARD DL. In practice, we see sequential MODL programs as sequential programs of JAVA CARD DL operating on non-standard variables.

4.1.2 Signature: Heap- and Stack-Allocated Data

We maintain the type hierarchy (⇒ Sect. 2.3.1) and the signature (⇒ Sect. 2.3.2) definitions of the sequential JAVA CARD DL. In particular, everything that is subject to assignment during program execution (variables, object attributes, arrays) is modeled by non-rigid functions. We call these functions *program variables*. The full details of this modeling in JAVA CARD DL were given in Note 2.4, though we summarize them again in Table 4.1a.

JAVA CARD DL does not distinguish between heap- and stack-allocated data. In MODL this distinction becomes important. A variable on the heap refers to a single value and assignments changing it are immediately visible to all threads.[1] On the other hand, every thread has its own copy of each local variable (allocated on the thread's stack). An assignment to a local variable within one thread is not visible to other threads.

Table 4.1b shows how program variables are handled in MODL. The difference to JAVA CARD DL is in how (thread-)local variables are modeled (first line). The thread-local variable v in a concurrent program refers to a *series* of values. When the program executes, the unique value is identified by the context of the currently running thread. In the logic, we can talk about the local variable values in different threads by using a combination of variable name and thread id. All other variables (lines 2–4) are considered heap-allocated and are modeled exactly as in JAVA CARD DL.

This way the appearance of thread-local variables depends on the context. A thread-local variable appears:

- in concurrent programs as: `v`
- in sequential programs as: $\mathtt{v}(t)$
- in the logic (incl. updates) as: $\mathtt{v}(t)$

where t is a thread identifier.

Example 4.1 (Arity of thread-local variables). Consider the concurrent MODL program

```
if (a>0)...
```

where `a` is a local variable. This thread-local variable `a` is modeled by a non-rigid function of arity 1. In the program, however, it appears without parameters, i.e, has the arity 0. Symbolic execution of this statement by a thread with id t will lead to the branch condition formula $\mathtt{a}(t) > 0$ appearing in the proof. At this point, the symbol `a` appears with its full arity. ◁

Note 4.2 (Predefined symbols). As with JAVA CARD DL (⇒ Note 2.4, Def. 2.13), we expect that a signature of MODL always contains certain predefined symbols (scheduler function symbols, enabledness predicate symbols, etc.). These symbols will be introduced in following definitions, usually together with axioms constraining their semantics. ◁

[1] The cross-thread visibility is actually subject to conditions of the Java Memory Model, which we discuss in detail in Section 7.3.

(a) in JAVA CARD DL

Program entity	modeled by	notation in logics
local variable v	constant	v
static field access *Class.a*	constant	*Class.a*
instance field access $o.a$	unary function	$a(o)$ or $o.a$
array access o`[`i`]`	binary function	`[]`(o,i) or o`[`i`]`

(b) in MODL

Program entity	modeled by	notation in logics	
local variable v (of thread t)	**unary function**	$\boldsymbol{v(t)}$	
static field access *Class.a*	constant	*Class.a*	heap access
instance field access $o.a$	unary function	$a(o)$ or $o.a$	heap access
array access o`[`i`]`	binary function	`[]`(o,i) or o`[`i`]`	heap access

Table 4.1. How program variables are modeled

4.1.3 Terms and Updates

Terms and updates are defined exactly as in JAVA CARD DL (⇒ Sect. 2.3).

4.1.4 Syntax of Programs

First, we define sequential programs, which later serve as building blocks for concurrent programs.

Our sequential programs have several peculiarities:

- There is a `stop` statement, which does nothing and is never enabled. This statement is of little use in the sequential case, but is used to model concurrent programs with several thread classes.
- There is an atomic block construct, which, again, only becomes useful when the programming language is extended with concurrency.
- Every sequential program is identified with some thread executing it. The thread id appears in all local variables as an (additional) argument.
- Assignments must not contain more than one heap access. This restriction is necessary to faithfully model the semantics of concurrent JAVA assignments. We consider assignments to be atomic in our language, while they indeed can be non-atomic in JAVA. A program with more than one heap access in an assignment can easily be transformed into a program satisfying the above condition by adding assignments that store the value of heap-allocated variables in fresh local variables.
- Conditions of if-then-else statements must be local variables not occurring in the then- or else-part of the statement. This restriction is similarly easy to satisfy by

adding assignments with fresh local variables. The fact that these variables—once set—cannot change their value eliminates technical difficulties when specifying execution path conditions.

Definition 4.3 (Sequential programs). The set of sequential programs is recursively defined as follows. For all thread ids τ:

(Stop)
`stop` is a program.

(Assignment)
f`(`t_1`, ... ,`t_n`)=`t`;` is a program if
1. f is a non-rigid function symbol of arity n
2. $t_1, \dots, t_n$, as well as t are terms correctly typed w.r.t. f
3. the assignment contains at most one heap access (⇒Table 4.1b).

(Sequential composition)
pq is a program if p and q are programs.

(Conditional)
`if (`$v(\tau)$`) {`p`} else {`q`}` is a program if p and q are programs and $v(\tau)$ is a thread-local boolean variable not appearing in p or q.

(Loop)
`while (`$v(\tau)$`) {`p`}` is a program if p is a program, and $v(\tau)$ is a thread-local boolean variable.

(Atomic block)
$\ll p \gg$ is a program if p is a program.

(Lock acquire)
$o(\tau)$`.<lock>();` is a program if $o(\tau)$ is a thread-local reference-valued variable.

(Lock release)
$o(\tau)$`.<unlock>();` is a program if $o(\tau)$ is a thread-local reference-valued variable. ◁

Example 4.4 (Sequential program syntax). The following is an example of a concrete sequential program executed by thread `t`:

```
o(t).<lock>();
a(t)=o(t).sum;
o(t).sum=a(t)+e(t);
o(t).<unlock>();
```
◁

We now use the sequential programming language to define concurrent programs. Conversely, the verification calculus breaks concurrent programs down into sequential fragments. The part of this process that builds a sequential program from a part of a concurrent one is called *sequential instantiation* (⇒ Def. 4.7).

Definition 4.5 (Concurrent programs). The set of concurrent programs is defined as follows. Every sequential program is a concurrent program under the following transformation/conditions:

- all occurrences of loops must be within atomic blocks
- atomic blocks may not be nested
- atomic blocks may not contain locking operations
- thread indices are stripped from statements
- all function symbols representing local variables are stripped of thread identity (number of arguments is one less than actual arity)
- the last statement of the program must be `stop`
- `stop` may only occur at the top level in a program. ◁

Example 4.6 (Concurrent program syntax). The following is an example of a concrete concurrent program with one thread class:

```
o.<lock>(); a=o.sum; o.sum=a+e; o.<unlock>(); stop; .
```

The following is an example of a concrete concurrent program with two thread classes:

```
x=1; stop; x=2; stop; .
```

◁

The purpose of the final `stop` statement is to provide a "parking position" for the threads that have run to completion. It also allows us to model parallel composition as sequential composition. The latter program in the above example is conventionally written as

`x=1;` || `x=2; .`

We will omit the final `stop` statement from concurrent programs whenever clarity is not sacrificed.

Definition 4.7 (Sequential instantiation). If p is a concurrent program and τ is a thread id, then the *sequential instantiation* $p^{*(\tau)}$ is a sequential program built by augmenting every thread-local variable v in p by the thread id, giving $v(\tau)$.

We define a sequential instantiation in an analogous manner also for terms. ◁

4.1.5 Program Positions, Control Variables, and Thread Configurations

Until now, we have dealt with syntactic programs, which are just templates for threads to execute. Now we introduce means to describe which threads are executing a program, and where exactly each thread is at any given moment. For this, we number all atomic sub-programs in a program (statements and atomic blocks) from left to right, starting with one. We call these numbers the *positions* of the program. Their intuitive meaning is that if a thread is at a certain position, it is about to execute the corresponding atomic statement when it is next scheduled to run. We will refer to the statement at position i in a program p as $p(i)$.

Every position i is associated with a *control variable* $pos(i)$, which is a set-valued variable, not occurring in programs. The control variable lists exactly the tids waiting to be scheduled at the resp. position. Together, the control variables specify the *thread configuration.*

Definition 4.8 (Thread configuration). A thread configuration for a program p is a non-rigid function symbol pos:

$$pos^p : \{1, \ldots, size(p)\} \to 2^{\mathcal{T}} \ .$$

In order not to clutter notation, we will omit the program index and just write pos. The program it refers to is always clear from the context. ◁

Example 4.9 (Thread configuration notation). In this example we assume that thread ids are integers. Then, $(\{3,17,5\}, \{\}, \{2\})$ is an example of a configuration of size 3. A configuration of size n is compatible with programs that have n positions.

We write (compatible) pairs of thread configurations and programs by inlining the values of the control variables within the program. For example, the program

```
v=(x<10); if (v) {a=10; x=a+1}
```

together with the configuration $(\{5\}, \{3,4\}, \{1\}, \{2\})$, where four threads are active and one has already terminated, is written as

$$^{\{5\}}\texttt{v=(x<10); if (v) \{}^{\{3,4\}}\texttt{a=x;}^{\{1\}}\texttt{x=a+1;\}}^{\{2\}} \ .$$

On the formula level, if $\mathcal{U}$ is an update and $\bar{c}|p$ is a program with an inlined thread configuration, the formula

$$\mathcal{U}\langle \bar{c}|p\rangle\phi$$

is shorthand for

$$\{pos(1) := c_1 \,\|\, \ldots \,\|\, pos(n) := c_n\}\mathcal{U}\langle p\rangle\phi \ .$$

◁

Note 4.10 (Disjointness of control variable values). In general, we expect to deal only with disjoint values of control variables: every thread can be at only one place at the same time. Nonetheless, a formula can describe a state where this is not true. To avoid complications, we assume the following semantics for this case. If two control variables $pos(i)$ and $pos(j)$ (for $i \neq j$) have overlapping values A and B, i.e., $A \cap B \neq \emptyset$, then the semantics of a program with this configuration is the same as of a program in a state where $pos(i)$ has the value $A \setminus B$ and $pos(j)$ has the value $B \setminus A$. ◁

Definition 4.11 (Threads in a program). The set of threads in a program $Tids(p)$ is

$$Tids(p) = \bigcup_{i=1}^{n} pos(i)$$

for a program p with n atomic positions. Technically, this set is state-dependent, but our programs can neither create nor destroy threads. ◁

4.1.6 The Scheduler

Definition 4.12 (Scheduler). For each program p, the *scheduler* is (modeled by) the predefined (⇒ Note 4.2) non-rigid function (constant) symbol

$$sched^p : \to \mathcal{T} \cup \{\bot\} \ ,$$

which says which thread is to run next in a given state. In order not to clutter notation, we will omit the program index and just write *sched* in the future. The program to which the scheduler function refers is always clear from the context. ◁

The interpretation of *sched* depends, in general, on program state, even though we try to minimize this dependency in our program semantics. Different models, furthermore, may interpret this function differently and, thus, have different schedulers. The value that *sched* returns must, of course, be compatible with other components of the state, i.e., the program variables and the control variables. To express this we first define what it means for a thread to be enabled.

Definition 4.13 (Statement Enabledness). We introduce a non-rigid predicate symbol $enabled(s,t)$ capturing when a thread t is enabled to execute a concurrent statement s. We declare the predicate predefined (⇒ Note 4.2)), and its values are given by the following table:

Statement s	Enabledness condition $enabled(s,t)$
`stop`	*false*
assignment	*true*
atomic block	*true*
o`.<lock>()`	$o(t)$`.<lockcount>=0` $\vee$ $o(t)$`.<lockedby>=`t
o`.<unlock>()`	*true*

◁

Definition 4.14 (Thread Enabledness). The following non-rigid predicate symbol captures when a thread t is enabled in a program p (we will, again, omit this program index in the future). The predicate is predefined (⇒ Note 4.2) with the semantics constrained by the axiom:

$$enabled^p(t) = \begin{cases} enabled(s,t), & \text{if } t \in Tids(p), \\ false, & \text{otherwise,} \end{cases}$$

where s is the statement at which t is waiting to be scheduled. Per Note 4.10 there is at most one such statement. If there is none, the predicate evaluates to false. ◁

We now state the scheduler axioms.

1. *The scheduler may only schedule existing threads.* Which threads "exist" is given by the control variables of the state for the program at hand:

$$sched \in Tids(p) \ . \tag{4.1}$$

2. *The scheduled thread must be enabled.* When a thread is enabled is defined in Def. 4.14. At this point the scheduler depends upon actual program variables.

$$sched = t \wedge t \neq \bot \longrightarrow enabled(t) \tag{4.2}$$

3. *If no thread is enabled, the scheduler must return* $\bot$. This is the case when the program has terminated or entered deadlock.

$$\begin{gathered} sched = \bot \longleftrightarrow \\ \forall t.\, t \in Tids(p) \longrightarrow \neg enabled(t) \end{gathered} \tag{4.3}$$

4. *The scheduler is dependent upon a scheduling seed.* A problem for a deterministic scheduler model is the possibility that a program returns to a previously visited state (a kind of déjà vu). In this case, it would be unreasonable to expect that the scheduler run the same thread as last time. This situation could occur, for instance, if our programming language allowed non-atomic loops.
   To keep our model general, we introduce yet another control variable: the *scheduling seed* σ. The semantics of the programming language would use σ to guarantee that as long as a program is running, it never passes the same state twice. For non-atomic loops this would mean making σ a ghost loop counter. Please note, that our programming language is already restricted in such a way as to have the no-déjà vu property without resorting to an explicit seed.
   There is yet another potential reason to have an explicit seed. The seed makes it possible to relate two different runs of the same program. An example of this is the atomicity criterion from Section 7.1:

$$\forall v. \big(\langle \alpha \quad \beta \quad \omega \rangle (\mathrm{x} = v) \longrightarrow \exists s. \{\sigma := s\} \langle \alpha \quad \ll\beta\gg \quad \omega \rangle (\mathrm{x} = v) \big) \ ,$$

   Here the seed is existentially quantified, and we take this opportunity to further explicate its semantics. The scheduler behavior is not only dependent on the seed, but it is dependent in such a way that it is possible to induce *any* legal schedule by selecting the right seed. In other words, the seed variable is a substitute for quantifying over schedulers, which is not possible directly in our logic.
   In practice, however, the seed feature is rather esoteric. In the following, we omit the seed from our further considerations.

In general, this is already everything we assume about a scheduler. Fairness[2] or other scheduler properties are not built into our model. Such properties can, however, be specified by adding further axioms restricting the function *sched*.

4.1.7 Formulas

The set of formulas is defined similar to Java Card DL (⇒ Def. 2.8). The only difference concerns modalities. MODL defines two concurrent and two sequential kinds of modal operators.

[2] It should be noted that Java itself is only "statistically fair".

Definition 4.15 (Formulas of MODL). We amend the definition of formulas from Java Card DL (⇒ Def. 2.8) as follows:

For each concurrent program p and every formula ϕ, $\langle p\rangle\phi$ (the concurrent "diamond" modality) and $[p]\phi$ (the concurrent "box" modality, which is a shorthand for $\neg\langle p\rangle\neg\phi$) are formulas.

If p is a sequential program and ϕ a formula, then $\langle \underline{p}\rangle\phi$ (the sequential "diamond" modality) and $[\underline{p}]\phi$ (the sequential "box" modality, which is a shorthand for $\neg\langle \underline{p}\rangle\neg\phi$) are formulas. ◁

Intuitively, a diamond formula $\langle \underline{p}\rangle\phi$ (resp. its concurrent counterpart $\langle p\rangle\phi$) means that the program p in the diamond must terminate (resp. all threads must terminate) and afterwards ϕ has to hold. The meaning of a box formula is the same, but termination is not required, i.e., ϕ must only hold *if* p terminates. The formula $\psi \longrightarrow [\underline{p}]\phi$ has the same meaning as the Hoare triple $\{\psi\}p\{\phi\}$.

4.2 Semantics of MODL

Unsurprisingly, we use *Kripke structures* (introduced in Sect. 2.4) as semantic domains to interpret MODL formulas. The semantics of terms and updates remains unchanged (Sect. 2.4.3 and 2.4.2), while the semantics of formulas is modified from Sect. 2.4.4 to introduce new concurrent modalities. The major part of this section concentrates on defining the semantics (transition relation ρ) of the concurrent and sequential programming languages of MODL.

Definition 4.16 (State variation). If $s \in \mathcal{S}$ is a state and $u \in \mathit{Updates}$ is an update, then $s' = s[\![u]\!]$ is a *state variation* (i.e., also a state). Formally, $s' = (\mathrm{val}_s u)(s)$. This means that $s' = (\mathcal{D}, \delta, \mathcal{I}')$ is identical to $s = (\mathcal{D}, \delta, \mathcal{I})$ except for the interpretation mapping, which is changed according to the update u. ◁

4.2.1 Semantics of Sequential Programs

As in Java Card DL (⇒ Def. 2.13), the semantics of sequential programs is given by a transition relation on states $\rho_1(p) \subseteq \mathcal{S}^2$, for any valid sequential program p. Since programs are deterministic, the relation is actually a partial function: $\rho_1(p): \mathcal{S} \to \mathcal{S}$.

Definition 4.17 (Semantics of sequential programs). The semantics of sequential programs $\rho_1(p)$ is the smallest relation satisfying the following conditions. It does not depend on the scheduler.

(Stop)

$\rho_1(\texttt{stop}) = \mathit{id}$

(Atomic block)

$\rho_1(\ll p \gg) = \rho_1(p)$

(Assignment)
$(s,s') \in \rho_1(f(t_1,\dots,t_n)=t)$ iff the statement $f(t_1,\dots,t_n):=t$ interpreted as a JAVA assignment does not throw an exception[3] and $s' = s[\![f(t_1,\dots,t_n):=t]\!]$.

(Sequential composition)
$(s,s') \in \rho_1(pq)$ iff $(s,s'') \in \rho_1(p)$ and $(s'',s') \in \rho_1(q)$ for some state s''.

(Conditional)
$(s,s') \in \rho_1(\texttt{if (}v(t)\texttt{) \{}p\texttt{\} else \{}q\texttt{\}})$ iff either
(1) $val_s(v(t)) = \mathit{TRUE}$ and $(s,s') \in \rho_1(p)$, or
(2) $val_s(v(t)) = \mathit{FALSE}$ and $(s,s') \in \rho_1(q)$.

(Loop)
$(s,s') \in \rho_1(\texttt{while (}v(t)\texttt{) \{}p\texttt{\}})$ iff there is an $n \in \mathbb{N}$ and there are states $s_0,\dots,s_n$ with $s=s_0$ and $s'=s_n$ such that
(1) for $0 \le i < n$, $val_{s_i}(v(t)) = \mathit{TRUE}$ and $(s_i,s_{i+1}) \in \rho_1(p)$, and
(2) $val_{s_n}(v(t)) = \mathit{FALSE}$.

(Lock acquire)
$(s,s') \in \rho_1(o(t)\texttt{.<lock>()})$ iff either
(Case 1: the lock is free)

$$val_s(o(t)\texttt{.<lockcount>}) = 0 \text{ and } val_s(o(t)\texttt{.<lockedby>}) = \bot$$

or

$$val_s(o(t)\texttt{.<lockcount>}) > 0 \text{ and } val_s(o(t)\texttt{.<lockedby>}) = val_s(t)$$

and, in either case,

$$s' = s\left[\!\!\left[\begin{array}{c} o(t)\texttt{.<lockcount>} := o(t)\texttt{.<lockcount>} + 1 \,\|\, \\ o(t)\texttt{.<lockedby>} := t \end{array}\right]\!\!\right]$$

or
(Case 2: the lock is taken)
$val_s(o(t)\texttt{.<lockcount>}) > 0$ and $val_s(o(t)\texttt{.<lockedby>}) \neq val_s(t)$ while $s' = s$.

(Lock release)
$(s,s') \in \rho_1(o(t)\texttt{.<unlock>()})$ iff either
(Case 1: lock depth not yet exhausted)

$$\begin{array}{l} val_s(o(t)\texttt{.<lockcount>}) > 1 \\ val_s(o(t)\texttt{.<lockedby>}) = val_s(t) \\ s' = s[\![o(t)\texttt{.<lockcount>} := o(t)\texttt{.<lockcount>} - 1]\!] \end{array}$$

or

[3] We do not give a formal definition, since we want to avoid formalizing here major portions of the JLS. In practice, when an exception is thrown is exhaustively formalized by the sequential KeY calculus.

(Case 2: lock depth exhausted)

$$\begin{array}{l} val_s(o(t)\texttt{.<lockcount>})=1 \\ val_s(o(t)\texttt{.<lockedby>})=val_s(t) \\ s'=s\left[\!\left[\begin{array}{c} o(t)\texttt{.<lockcount>}:=0 \,\|\, \\ o(t)\texttt{.<lockedby>}:=\bot \end{array}\right]\!\right] \end{array}$$

◁

4.2.2 Semantics of Concurrent Programs

The semantics of concurrent programs is given by a transition relation on states $\rho(p)\subseteq\mathcal{S}^2$, for any valid concurrent program p. As explained previously, even our concurrent programs are deterministic (by means of an underspecified deterministic scheduler). Thus, this relation is a partial function: $\rho(p):\mathcal{S}\to\mathcal{S}$. We will define ρ below.

To make specifying the semantics of if-statements easier we assume that every thread steps through both the then- and the else-part of all if-statements. Yet the thread can only change the state if it is in the "right" part and executes NOPs otherwise. The *path condition* tells us if we are in the right part.

Definition 4.18 (Path condition of position in program). Let k be a position of an atomic sub-program in a non-atomic program p. Let this position occur within the scope of $n\geq 0$ (nested) if-statements in their then- or else part. Let $v_1,\ldots,v_n$ be the conditions of these if-statements.

Since, by definition, the local variable v_i does not occur in the then- or else-part of the ith if-statement, its value is not changed during the execution of the if-statement after it has been evaluated.

We define the *path condition* of k in p as the formula:

$$path(k,p,tid)=B_1\wedge\ldots\wedge B_n \quad ,$$

where

$$B_i=\begin{cases}\left(v_i(tid)=TRUE\right), & \text{if } k \text{ is in the then-part of the } i\text{th if-statement}\\ \left(v_i(tid)=FALSE\right), & \text{if } k \text{ is in the else-part.}\end{cases}$$

◁

Thus, a thread t will execute the atomic program at k within p iff $path(k,p,t)$ holds.

Example 4.19. The path condition of the statement `l=r;` in the program

```
if (a) {if (b) {} else {l=r;}} else {}
```

for a thread t is

$$\texttt{a}(t)=TRUE\wedge\texttt{b}(t)=FALSE \quad .$$

◁

Our next goal is to define the semantics of concurrent programs $\rho(p)$. The base for this is the semantics of sequential programs $\rho_1(p)$. We use it to describe the first step in the execution of a concurrent program, which is identified by the scheduler function. All further steps of the concurrent program are handled by recursively repeating the process.

Definition 4.20 (Semantics of concurrent programs). The semantics $\rho(p)$ of a concurrent program p is inductively defined as the smallest relation such that:

- $(s,s) \in \rho(p)$ if no thread of p is enabled in s, i.e., $sched = \perp$ in s.
- $(s,s') \in \rho(p)$ if some thread of p is enabled in s , and
  (1) $sched = tid$ in s
  (2) $tid \in pos(i)$ in s (there is always exactly one such i, cf. Note 4.10)
  (3) q is the atomic sub-program at position $pos(i)$ in p
  (4) $s \vDash path(\mathrm{i}, p, tid)$,
  (5) $(s,s'') \in \rho_1(q^{*(tid)})$ for some state s''
  (6) $val_{s''}(pos(i)) = val_s(pos(i)) \setminus \{tid\}$ and
  $val_{s''}(pos(i+1)) = val_s(pos(i+1)) \cup \{tid\}$
  (7) $(s'', s') \in \rho(p)$
- $(s,s') \in \rho(p)$ if some thread of p is enabled in s , and
  (1)-(3) as above
  (4) $s \nvDash path(i, p, tid)$,
  (5) there is a state $s'' = s[\![pos(i) := pos(i) \setminus \{tid\} \,||\, pos(i+1) := pos(i+1) \cup \{tid\}]\!]$
  (6) $(s'', s') \in \rho(p)$ $\lhd$

4.2.3 Semantics of Formulas

Now, we can define the semantics of formulas with modalities in a way similar to Def. 2.21.

Definition 4.21 (Semantics of formulas).

(Modalities with concurrent programs)
$s \vDash \langle p \rangle \phi$ iff $(s,s') \in \rho(p)$ for some state s' with $s' \vDash \phi$.

(Modalities with sequential programs)
$s \vDash \langle \underline{u} \rangle \phi$ iff $(s,s') \in \rho_1(u)$ for some state s' with $s' \vDash \phi$.

We say that a Kripke structure is a *model* of a formula ϕ iff $s \vDash \phi$ is true in all states s of that structure. A formula ϕ is *valid* if all Kripke structures are a model of ϕ. $\lhd$

5

MODL—A More Verification-Friendly Version

"If it can be programmed, it can be verified." This adage was not only a constant encouragement in the development of this proof system, but also a quite concrete guidance. Practice makes it clear that programmers do not reason about all possible interleavings when programming a multi-threaded application. They rather consider equivalence classes, since many interleavings are not significantly different. A verification system can and should make use of this circumstance, and a crucial factor in building the equivalence classes is thread symmetry.

In this chapter we refine the notion of thread configuration and the corresponding scheduler model given in the previous chapter. The refined model allows us to summarize many symmetric program executions in classes, to reason about an unbounded number of threads, and altogether to verify multi-threaded programs with a feasible effort. The chapter is concluded by combinatorial results relevant to multi-threading.

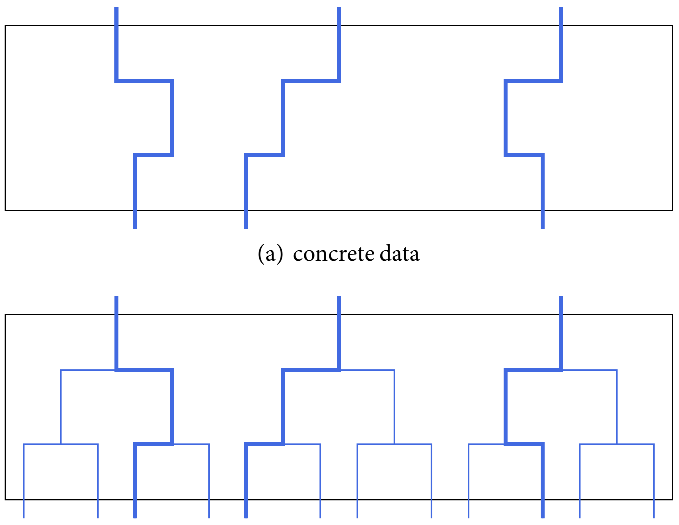
5.1 Do Not Enumerate—Describe!

The logic presented in the previous chapter already gives a complete account of multi-threading for the chosen language fragment and even allows symbolic execution of programs. It has two deficiencies though:

- The threads involved are explicitly enumerated, even when the concrete ids are actually not important. This circumstance makes it impossible to make statements about an unbounded (fixed but unknown) number of threads.
- Transitions are always totally ordered (resulting in proof branching), even if they are independent. Consider two threads τ_1 and τ_2 that are ready to be scheduled at the same position. Under the enumeration scheme, we have to perform a case distinction, which thread will run first, even if this distinction is not important later in the proof. Since we are dealing with symbolic data, this distinction is almost never important. The up-front distinction is inefficient and—again—prevents us from verifying programs with an unbounded number of threads.

To overcome these obstacles, we have developed a more refined logic where configurations are not enumerated but described algebraically. Efficient laws for reasoning about these descriptions complete the picture. The basis for the efficiency gain is symbolic thread symmetry.

Extending Symmetry Reduction


(a) concrete data

(b) symbolic data

Figure 5.1. Explored thread trajectories in a program

Symmetry reduction is a well-known idea that different threads with the same properties need not be distinguished. Most model-checking frameworks implement some sort of symmetry reduction to prune the state space. This feature is described prominently, for instance, in [Robby et al., 2003c] (the Bogor tool) and [Yahav, 2001] (on-the-fly model-checking with TVLA). However, detecting symmetries can be expensive, and most tools used in practice only detect symmetry when several threads have exactly the same concrete local data and program counter. Such a situation is as well as not present in the scenario depicted in Figure 5.1(a).

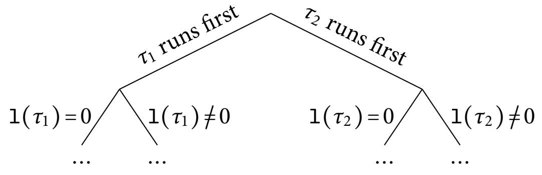
In a deductive verification system we can give the idea of symmetry reduction a new twist. We want to identify not just threads with the same local data, but threads with similar proof shapes. Indeed, when executed symbolically, most threads of the same thread class have similar proof shapes (Figure 5.1(b)), as symbolic execution explores all possible paths through the code. The number of such paths is finite and relatively small; it is bounded by the shape of the program text.[1]

Having paid the price of sequential symbolic execution in case distinctions, we might now as well reap the benefits in the concurrent case. We can—to a large extent—eliminate the necessity to consider different orderings of threads that have reached the same position within the program.

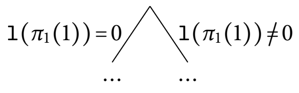
Example 5.1 (Symbolic symmetry reduction). Consider two threads τ_1 and τ_2 that are ready to execute the statement

$$\texttt{if (l==0) } \alpha \texttt{ else } \beta \text{ ,}$$

where `l` is a thread local variable. In the proof schematically shown in Figure 5.2(a), the distinction whether τ_1 or τ_2 runs first is performed up-front. Figure 5.2(b) shows how symmetry reduction allows to postpone this choice by hiding it in the scheduler function. Here $\pi_1(1)$ is the id of the thread to run first, and the following proof is implicitly quantified over all possible values of $\pi_1(1)$. In most cases this quantification



(a) without symmetry reduction



(b) with symmetry reduction

Figure 5.2. Symbolic execution trees for two threads τ_1 and τ_2 ready to execute the statement `if (l==0)` α `else` β

can be easily eliminated by applying algebraic laws about permutations and similar reasoning. In these cases symbolic symmetry reduction is successful. Otherwise, one or several case distinctions have to be performed on $\pi_1(1)$. Since no information about thread ids is ever removed, no unsoundness is introduced in the process. $\lhd$

Symmetry reduction eliminates proof complexity caused by different possible orderings of threads at one interference point. To deal with the number of interference points, one applies standard techniques for identifying atomic regions based on locking and data encapsulation. To deal with the possibly unbounded number of threads in the system ("unbounded concurrency"), one applies induction. Together, the three components (symmetry reduction, atomicity, induction) make deductive verification of concurrent systems feasible.

[1] Remember that we only consider atomic loops, which can be compressed into a single (complex) computation step.

Expressing Unbounded Concurrency

In our program model we pretend that each thread linearly traverses the program: There is no jumping back (except within an atomic loop), and each thread visits each position at most once (never, if it gets stuck on its way in an atomic loop or trying to acquire a lock). This means, however, that threads can end up in the "wrong" branch of an if-then-else statement. To preserve the original semantics of the program, we arrange that the state is not changed by the program while its control flow is in the wrong place.

We have now "forced" each thread to visit each program position at most once. Assuming threads with tids $1, \ldots, n$, it is clear that for every position i, there is a permutation $\pi_i \colon \{1 \ldots n\} \to \{1 \ldots n\}$ that describes the order in which the threads are scheduled at this position (should they reach it).

Given these permutations, it is sufficient to know *how many* threads are at each position. This fixes the exact configuration as well and allows writing configurations with m positions as $(\pi_0, \pi_1{:}k_1, \ldots, \pi_m{:}k_m)$, where $\pi_0, \ldots, \pi_m$ are terms representing the permutations and $k_1, \ldots, k_m$ are terms representing the number of threads.

Describing Thread Configurations

Definition 5.2 (Thread configuration). Configurations with explicit tids were introduced in Def. 4.8. We now overload this term with the following definition. Unless explicitly stated otherwise, in the following, all configurations refer to the following formulation.

A thread configuration for a program p is a family of non-rigid function symbols

$$\pi_i^p \colon \mathbb{N} \to \mathcal{T} \quad \text{for } i \in \{0, \ldots, size(p)\}$$

together with a non-rigid function symbol

$$pos^p \colon \{1, \ldots, size(p)\} \to \mathbb{N} \ .$$

π_i is a permutation of the set of tids $\mathcal{T}$, encapsulating the scheduler decisions at position i. $pos(i)$ is the number of threads currently available for scheduling at position i. ◁

In order not to clutter notation, we will omit the program index and just write π_i and *pos*. The program they refer to is always clear from the context. As before, we also often present configurations as inlined within programs. This time we limit ourselves to the values of *pos*. Since we never deal with concrete values of π_i, we omit them when stating configurations and simply imply their existence.

Example 5.3. Consider a program of size four with 2, 3, 5 and 7 threads waiting at each position respectively. The thread configuration of this program consists of the non-rigid function *pos* (with $pos(1) = 2, pos(2) = 3, pos(3) = 5, pos(4) = 7$), and the five non-rigid "permutation" functions $\pi_0, \ldots, \pi_4$ (whose values we do not know). Altogether there are 17 threads, which we can represent as $\{\pi_0(1), \ldots, \pi_0(17)\}$.

If we concentrate on position 1, we can see that $3 + 5 + 7 = 15$ threads have already passed this position and the next one to execute will be the 16th in count. If we now concentrate on position 2, we can see that $5 + 7 = 12$ threads have already passed this position and the next one to execute will be the 13th in count. ◁

Definition 5.4 ($Post(\cdot)$). For a given program p of size n (implied) and a position $i \leq n$, we define a predefined (⇒ Note 4.2) non-rigid function symbol $Post(i)$ with the semantics fixed by:

$$Post(i) = \begin{cases} pos(i) & \text{if } i = n \text{, or if the statement at position } i \text{ in } p \text{ is } \texttt{stop} \\ pos(i) + Post(i+1) & \text{otherwise.} \end{cases}$$

This is the number of threads of one thread class in p, which have already passed position i in the current state. In absence of `stop` statements in p (i.e., if there is only one thread class), the situation is simpler:

$$Post(i) = pos(i+1) + \ldots + pos(n) \quad .$$

◁

Example 5.5 (Example 5.3 continued). So, in our example $Post(2) = 5 + 7 = 12$. The next thread scheduled at position 2 is the $(Post(2) + 1) = 13$th thread. But exactly which one is the 13th? Here the permutation functions come into play. The exact tid of the thread scheduled to run next at position 2 is given by $\pi_2(Post(2) + 1) = \pi_2(13)$. This way we can talk concisely about thread orderings even if we don't know them exactly. ◁

The same way we can write configurations where the number of threads is not a concrete number but a variable. This very expressive form of writing allows us to formulate rules that do not take the scheduling order into account, as it is hidden inside the permutation functions. What we need for a complete calculus are then the usual algebraic properties of permutations and axioms of their interplay.

As mentioned above, the pos and the π_i functions completely fix the thread lineup. We now state exactly how, by defining the function pos_γ which in any given state produces an enumerative configuration in the sense of Def. 4.8.

Definition 5.6 (Configuration concretization). A *concretization function* (of size n) is a predefined non-rigid function symbol

$$pos_\gamma : \{1, \ldots, n\} \to 2^{\mathcal{T}}$$

with the semantics fixed by

$$pos_\gamma(i) = \left\{ \pi_{i-1}(1), \ldots, \pi_{i-1}(Post(i-1)) \right\} \setminus \left\{ \pi_i(1), \ldots, \pi_i(Post(i)) \right\} \quad .$$

◁

The intuition behind this definition is the following. The threads waiting at position i are exactly those that have already passed the position $i-1$, but excluding those that have already moved on past i.

Example 5.7 (Example 5.3 continued). We now translate the four integers and the five permutations from above into an enumerative 4-set configuration:

$$\begin{pmatrix} pos_y(1), \\ pos_y(2), \\ pos_y(3), \\ pos_y(4) \end{pmatrix} = \begin{pmatrix} \left\{\pi_0(1),\dots,\pi_0(17)\right\} \setminus \left\{\pi_1(1),\dots,\pi_1(15)\right\}, \\ \left\{\pi_1(1),\dots,\pi_1(15)\right\} \setminus \left\{\pi_2(1),\dots,\pi_2(12)\right\}, \\ \left\{\pi_2(1),\dots,\pi_2(12)\right\} \setminus \left\{\pi_3(1),\dots,\pi_3(7)\right\}, \\ \left\{\pi_3(1),\dots,\pi_3(7)\right\} \end{pmatrix}$$

◁

Note 5.8 (Configurations in physics). Different ways to formalize thread configurations have their parallels in statistical mechanics, which studies configurations of particles in discrete energy states. Under Maxwell-Boltzmann assumptions, the particles are always distinguishable, while the Bose-Einstein configurations do not distinguish between particles in the same state. Thread configurations with explicit tids have their counterpart in the former, while the abstract configurations correspond to the the latter. ◁

5.2 New Scheduler Formalization

Since we are aiming towards identifying all threads that have reached a certain position within the program, we wish to decompose the scheduling function into two components: the position choice function $\mathcal{P}$ and the thread choice functions π_i. In the following we will be restating the important definitions of concrete MODL primarily in terms of positions instead of in terms of threads.

The main component of the new scheduler formalization is the *position choice* function $\mathcal{P}$. It returns the position from which the next thread will be scheduled in the current state—or 0, if no enabled positions (⇒ Def. 5.9) remain.

Putting $\mathcal{P}$ together with the permutations introduced in the previous section, we obtain the following decomposition of the scheduler function (for non-disabled configurations):

$$sched = \pi_{\mathcal{P}}(Post(\mathcal{P}) + 1) \quad . \tag{5.1}$$

Position choice function characterization

In this section we state axioms for the position choice function, but first we need to define when a position is enabled. A position i is *enabled* in a configuration iff its tid set is not empty and its statement is enabled (⇒ Def. 4.13) for some thread at this position.

Definition 5.9 (Position enabledness). We introduce a non-rigid predicate symbol $enabled^{p}(i)$ capturing when a position i is enabled in a program p (which we will omit as it is clear from the context). We declare the predicate predefined (⇒ Note 4.2), and its semantics is constrained by the following axiom:

$$enabled(i) = \exists t. \left(t \in pos_{\gamma}(i) \wedge \left(path(i, p, t) \longrightarrow enabled(p(i), t)\right)\right) \ .$$

◁

Note 5.10. Note that for all statements except lock acquire the quantifier can be trivially eliminated. The same applies in the common case that all threads try to acquire the same lock (in absence of reentrant locking). These are exactly the cases of full symmetry between threads.

For instance, considering an assignment, the enabledness condition becomes simply:

$$pos(i) > 0 \ .$$

◁

Having defined position enabledness, we now axiomatize the position choice function. To achieve an adequate scheduler representation, the position choice function is subject to the following axioms:

- Only valid positions (or zero) are returned:

$$0 \leq \mathcal{P} < size(p). \tag{5.2}$$

  This axiom effectively amounts to a disjunction over the positions of p, which during the proof gives rise to a case distinction. Note that $size(p)$ is never returned, since the last position must be a `stop`, which is never enabled.
- The non-zero values of $\mathcal{P}$ are further restricted to the positions enabled in a given configuration:

$$\mathcal{P} \neq 0 \longrightarrow enabled(\mathcal{P}) \ . \tag{5.3}$$

- $\mathcal{P}$ may only return 0 if no position is enabled:

$$\begin{gathered}\mathcal{P} = 0 \longrightarrow \\ \forall i. \left(1 \leq i < size(p) \longrightarrow \neg enabled(i)\right) \ .\end{gathered} \tag{5.4}$$

Thread choice function characterization

Each thread choice function π_i is in every state an injective mapping from $\mathbb{N}$ to the set of tids $\mathcal{T}$ (we assume there are infinitely many thread ids). The injectivity is based on the fact that no thread can pass the same position twice as we have ruled out non-atomic loops. Formally:

$$\pi_k(i) = \pi_k(j) \text{ iff } i = j \ . \tag{5.5}$$

While it is our goal to assume as little about the thread choice functions as possible, in reality they are not completely arbitrary. The thread choice function at position i can only choose threads available at this position|

$$\pi_i(Post(i)+1) \in pos_\gamma(i) \ . \tag{5.6}$$

This constraint ties the choice at position i to the choices made at previous positions. Our calculus uses the axioms presented here to gather a constraint on the scheduler while exploring the behavior of the program.

Note 5.11. For efficient reasoning, the formula $\pi_i(Post(i)+1) \in pos_\gamma(i)$, which is shorthand for

$$\pi_i(Post(i)+1) \in \left\{\pi_{i-1}(1), \dots, \pi_{i-1}(Post(i-1))\right\} \setminus \left\{\pi_i(1), \dots, \pi_i(Post(i))\right\} \ ,$$

can be simplified. The subtracted term can be safely dropped if we recall injectivity of π_i. Together with (5.5) it is sufficient to demand that:

$$\pi_i(Post(i)+1) \in \left\{\pi_{i-1}(1), \dots, \pi_{i-1}(Post(i-1))\right\} \ .$$

$\triangleleft$

Finally, the threads of different thread classes are never confused. If there is a `stop` statement at position b in a program, then

$$\forall i, j, x, y. \left(i < b \land j \geq b \longrightarrow \pi_i(x) \neq \pi_j(y)\right) \ .$$

New definition of program semantics

In parallel to Def. 4.20, we now state a new definition of program semantics.

Definition 5.12 (Semantics of concurrent programs). The semantics $\rho(p)$ of a concurrent program p is inductively defined as the smallest relation such that:

- $(s,s) \in \rho(p)$ if no position of p is enabled in s, i.e., $\mathcal{P}=0$ in s.
- $(s,s') \in \rho(p)$ if some position of p is enabled in s , and
  (1) $tid = \pi_{\mathcal{P}}(Post(\mathcal{P})+1)$ in s
  (2) q is the atomic sub-program at position $\mathcal{P}$ in p
  (3) $s \models path(\mathcal{P}, p, tid)$,
  (4) $(s,s'') \in \rho_1(q^{*(tid)})$ for some state s''
  (5) $val_{s''}(pos(\mathcal{P})) = val_s(pos(\mathcal{P})) - 1$ and
  $val_{s''}(pos(\mathcal{P}+1)) = val_s(pos(\mathcal{P}+1)) + 1$
  (6) $(s'',s') \in \rho(p)$
- $(s,s') \in \rho(p)$ if some position of p is enabled in s , and
  (1)-(2) as above
  (3) $s \not\models path(\mathcal{P}, p, tid)$,
  (4) there is a state $s'' = s[\![pos(\mathcal{P}) := pos(\mathcal{P}) - 1 \| pos(\mathcal{P}+1) := pos(\mathcal{P}+1) + 1]\!]$
  (5) $(s'',s') \in \rho(p)$ $\triangleleft$

5.3 Combinatorial Effects of Symmetry Reduction

A calculus based on symbolic execution uses case distinctions to explore different program schedules. The number of cases is one measure of the calculus' efficiency. This number is roughly equal to the number of thread interleavings considered.

Our employment of extended symmetry reduction allows us not to distinguish between threads with the same program counter and thus reduce the number of interleavings considered (at the price of having to reason about permutations). One question to be asked is how high a reduction we achieve. In the following we survey combinatorial results for multi-threaded programs and provide statistics on the number of interleavings (without symmetry reduction), the number of configurations and the number of interleavings when employing symmetry reduction.

Counting Thread Interleavings

How many different interleavings can a multi-threaded program exhibit? A total of T threads: $t_1, \ldots, t_T$, each with $s_i, 1 \le i \le T$ atomic statements produces:

$$I(s_1, \ldots, s_T) = \binom{s_1 + \ldots + s_T}{s_1, \ldots, s_T}$$

interleavings. This *multinomial coefficient* [Cohen, 1978] counts the number of ways to put $\sum s_i$ unique balls into T boxes, where each box can hold s_i balls. Boxes correspond to threads in our model, while balls are numbered 1 to $\sum s_i$ and represent all the steps in the common schedule.

This number can be computed in the following way, which models successive filling of the boxes:

$$I(s_1, \ldots, s_T) =$$
$$\binom{s_1 + s_2 + \ldots + s_T}{s_T} \cdots \binom{s_1 + s_2 + s_3}{s_3} \binom{s_1 + s_2}{s_2} = \frac{(s_1 + \ldots + s_T)!}{s_1! \cdots s_T!} \quad .$$

In the uniform case, where each of T threads has S statements, the number of interleavings is:

$$I(\underbrace{S, \ldots, S}_{T}) = \frac{(ST)!}{(S!)^T} \quad .$$

Counting the Number of Thread Configurations

First, we want to count configurations where threads are always distinguishable. A program with S atomic statements requires a thread configuration of size $S + 1$. The number of such thread configurations with identity is:

$$(S + 1)^T \quad ,$$

since each unique thread can choose from $S+1$ positions independently. This is known as the number of *redundant permutations* in combinatorial theory.

Now, we turn to the case where we do not distinguish threads with the same program counter. For this we define $\left\langle {S \atop T} \right\rangle$ as the number of ways to put T indistinguishable balls into S labeled boxes of unlimited size (*redundant combinations*). It is known [Cohen, 1978, Theorem 15] that

$$\left\langle {S \atop T} \right\rangle = \binom{S+T-1}{T} .$$

The total number of thread configurations without identity of size $S+1$ is thus:

$$\left\langle {S+1 \atop T} \right\rangle = \binom{S+T}{T} .$$

Furthermore, for any given $k \leqslant S$, the number of configurations with k enabled positions is:

$$\left| {S, T \atop k} \right| = \binom{S}{k} \left\langle {k+1 \atop T-k} \right\rangle = \binom{S}{k}\binom{T}{k} .$$

The key to this calculation is selecting k positions and putting a ball into each of them first. The rest of the balls can be distributed between these k selected and the last position.

Counting the Cases Considered by the MODL Calculus

Finally, we consider what effect symmetry reduction has on the number of interleavings and thus on the case distinctions performed by the calculus. This counting problem is closely related to counting problems over other domains such as lattice walks, Dyck paths, Young tableaux, etc. [OEIS A060854, 2008].

Let us consider a program of S statements and T threads without identity. Furthermore, we abstract from data, i.e., assume that all path conditions are true. In this case, program executions (interleavings) can be modeled as S-dimensional lattice walks (with positive unit steps) from $(0, \ldots, 0)$ to $(T, \ldots, T)$ such that all the points on the walk satisfy $x_1 \geqslant x_2 \geqslant \cdots \geqslant x_S \geqslant 0$.

The number of such lattice walks is known to be equal to the multidimensional Catalan number [OEIS A060854, 2008] $C_{S,T}$, where

$$C_{s,t} = \frac{0!1! \cdots (t-1)!}{s!(s+1)! \cdots (s+t-1)!} (st)! .$$

In particular, for $s=2$ (or $t=2$, as $C_{s,t} = C_{t,s}$), $C_{2,t}$ corresponds to the "regular" Catalan numbers [OEIS A000108, 2008].

Asymptotically, the Catalan numbers grow "only" exponentially:

$$C_{2,n} \sim \frac{4^n}{\sqrt{\pi n}(n+1)} .$$

This can be derived from the Stirling approximation of $n!$.

The first few multidimensional Catalan numbers are charted in Table 5.1.

$s \backslash t$	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	1	2	5	14	42	132	429	1430	
3	1	5	42	462	6006	87516	1385670		
4	1	14	462	24024	1662804	140229804			
5	1	42	6006	1662804	701149020				
6	1	132	87516	140229804					
7	1	429	1385670						
8	1	1430							
9	1								

Table 5.1. Multidimensional Catalan numbers $C_{s,t}$

Conclusion

The above calculations show that symmetry reduction lowers the number of interleavings that have to be considered from superexponential to "just" exponential. The number of case distinctions to be made remains after all too high for using the current MODL calculus as a model checking tool. This, however, is not detrimental to our goals, as the main advantage of symmetry reduction remains. It opens the possibility to make logical assertions that are parametric on the number of threads (without requiring explicit thread enumeration), thus paving the way to using induction on this parameter.

6

A Calculus for MODL

Most of our lives are about proving something,
either to ourselves or to someone else.

6.1 Calculus Overview

The MODL calculus is built from the following new components:

A. rules for symbolic execution of concurrent programs (interleaving and symmetry reduction engine) (⇒ Sect. 6.3)
B. rules for reasoning about scheduling functions (permutations) produced by Component A (presented as axioms in Sect. 5.2)
C. concurrent invariant rule (not needed for completeness) (⇒ Sect. 6.5)
D. unfolding rules for translating JAVA to MODL (⇒ Sect. 6.2)

as well as the following pre-existing components of the sequential calculus (presented in Sect. 2.5.6):

1. FOL rules, reasoning about equality and arithmetics, induction
2. rules for symbolic execution of atomic sequential program fragments produced by Component A
3. invariant rule for sequential loops
4. method contract rules (further modularization)
5. rules for simplification and application of updates, which are produced by Component 2 (efficient aliasing treatment)

The Components 1–5 have been borrowed (with very minor modifications) from the stock KeY system.

6.2 Program Unfolding: Translating Java to MODL

The calculus given later in this chapter operates on MODL programs. Here we describe a mapping from Java programs satisfying the requirements given in the introduction to MODL. The mapping is such that the Java program and its counterpart in MODL perform the same state transition: if started in the same state, both will either terminate in the same state or not terminate at all. Thus, if the MODL program can be verified, then the original Java program is correct as well.

Translating Java to MODL is a two-step process. First, we completely *unfold* the Java program using a special rule set in KeY. The result is a more fine-grained Java program that is semantically equivalent to the original. Then, we use a simple transformation from the unfolded program into MODL.

Unfolding the Java program

The basis for the unfolding process is the calculus for sequential Java programs described in Sect. 2.5.5.

For a concurrent Java program α, the unfolded Java program α' satisfies the following conditions:

1. α' is trace-equivalent to α (w.r.t. vocabulary of α)
2. all occurring expressions are in normal form, i.e., it is no longer possible to factor out subexpressions by means of fresh local variables
3. each assignment is atomic (i.e., updates at most one heap location)
4. the conditions of if-statements and loops are fresh local variables
5. the conditions of if-statements do not occur in the then- or else-part of the statement
6. method calls are inlined, if necessary together with extra conditionals to simulate dynamic binding.

To achieve this, we utilize the rules that are already a part of the sequential KeY calculus. Examples of sequential unfolding are given in Sect. 2.5.5. These rules introduce fresh local variables and additional assignments. Examples of concurrent program unfolding are given in Table 6.1. Furthermore, instance creation is already modeled by assignment to ghost fields in the KeY calculus, and method implementations are inlined.

We have manually inspected the KeY rule base identifying the rules that perform the unfolding. Syntactically, these rules can be very closely approximated as the rules that match programs, but do not produce updates or case distinctions. For instance, the unfolding rules include the rule ifElseUnfold, but not the rule ifElseSplit (⇒ Sect. 2.7.2). Minor fine-tuning was subsequently performed to ensure the atomicity condition of assignments. We have also checked that no rules "swallow" intermediate states, i.e., perform optimizations like replacing `i++;i--;` by a NOP. The resulting rules were then pooled in a special *unfolding strategy* of the prover.

JAVA statement	unfolded form
`o.a=u.a++;`	`v=u.a; u.a=v+1; o.a=v;`
`if (o.a>1)` $\{\alpha\}$ `else` $\{\beta\}$	`v=o.a>1; if (v)` $\{\alpha'\}$ `else` $\{\beta'\}$
`while (o.a>1)` $\{\alpha\}$	`v=o.a>1; while (v)` $\{\alpha'$ `v=o.a>1;}`
`synchronized(`o`) {` α `}`[†]	o`.<lock>();` α' o`.<unlock>();`

`v` is in each case a fresh local variable of appropriate type

[†] The correct way to unfold a synchronized block is actually
`try {`o`.<lock>();` α`} finally {`o`.<unlock>();}`,
but since we do not allow catching exceptions at the moment,
we are using a simpler version.

Table 6.1. Examples of unfolding JAVA programs

Translating unfolded JAVA into MODL

After the program has been completely unfolded, it almost satisfies the syntax requirements of MODL. The biggest missing piece is the atomicity requirement for loops. The user must declare code sections containing loops as atomic. More atomic sections can be introduced in order to improve proof performance. In both cases, one needs to carry out further justification (⇒ Sect. 7.1). Finally, it remains to compose different thread classes by means of `stop` statements, add initial thread configurations and, in general, formulate the proof obligation.

6.3 The Basic Rules of Concurrent Execution

The calculus presented in the following makes extensive use of the axioms given in the previous two chapters. The axioms are the constraints on the interpretation of predefined functions and predicates given in their definitions. These axioms can be added to the antecedent of a proof goal at any time. Among symbols subject to axioms are enabledness predicates (⇒ Def. 5.9), path conditions (⇒ Def. 4.18), scheduler functions (⇒ Sect. 5.2), etc.

Figure 6.1 shows the main rule of the MODL calculus. The rule shows how to symbolically execute any statement that is not a concurrency primitive. In the rule, α and ω denote unchanged program parts, and i is the position of the executed atomic statement S (in the overall program p). $S^{*(tid)}$ is the sequential instantiation (⇒ Def. 4.7) of S for the currently running thread tid, which is an abbreviation for:

$$tid = \pi_i(Post(i) + 1) \ .$$

The formula $path(i, p, tid)$ is the path condition (⇒ Def. 4.18) of the statement S in p for thread tid. $\mathcal{P}$ is the position choice function, and the first premiss encodes the

$$\mathsf{step}\ \frac{\begin{array}{r} \Longrightarrow \mathcal{P}=i \\ path(i,p,tid) \Longrightarrow \langle\!\lbrack \underline{S^{*(tid)}} \rbrack\!\rangle \langle\!\lbrack \alpha^{\{n-1\}} S^{\{k+1\}}\,\omega \rbrack\!\rangle \phi \\ \neg path(i,p,tid)\Longrightarrow \qquad\qquad \langle\!\lbrack \alpha^{\{n-1\}} S^{\{k+1\}}\,\omega \rbrack\!\rangle \phi \end{array}}{\Longrightarrow \langle\!\lbrack \alpha^{\{n\}} \underset{\substack{\uparrow\\ \text{position } i \text{ in } p}}{S^{\{k\}}}\, \omega\rbrack\!\rangle\phi}$$

Figure 6.1. The concurrent symbolic execution rule

scheduler decision to schedule a thread at position i next. Since the scheduler behavior is in general unknown, this rule is usually applied after a case distinction over possible values of $\mathcal{P}$. These are, in turn, dictated by the scheduler axioms (⇒ Sect. 5.2).

After applying the step rule, the sequential program $S^{*(tid)}$ has to be tackled by the rules of the sequential KeY calculus. Eventually, it will be reduced to a series of updates and case distinctions.

Finally, if no position is enabled in a configuration, the program does nothing and the modality can be removed altogether. The following rule applies:

$$\mathsf{empty\text{-}program}\ \frac{\Longrightarrow \mathcal{P}=0 \qquad \Longrightarrow \phi}{\Longrightarrow \langle\!\lbrack p \rbrack\!\rangle \phi}$$

A Simple but Complete Verification Example

The following example is popular in the field (e.g., [Abadi et al., 2006]), since it already exhibits a large part of issues inherent to thread-based concurrency.

Example 6.1. Consider a financial transaction system that processes concurrent incoming payments for an account. We wish to establish that all payments end up deposited, regardless of their number and the order in which the threads are scheduled. This can be expressed by the following proof obligation, where `sum` is a static variable and `e` is a thread-local variable containing the payment amount.

$$\forall n.\{\mathtt{sum}:=0\}\langle^{\{n\}}\ll\mathtt{sum=sum+e;}\gg^{\{\}}\rangle(\mathtt{sum}=\sum_{i=1}^{n}\mathtt{e}(\pi_0(i))) \tag{6.1}$$

Note that for presentation purposes we have abused the programming language by writing an assignment with two heap accesses. This shorthand is permissible here, since the assignment is protected by an atomic block. This protection ensures that the assignment `a=sum+e;sum=a;` (as the above is properly written) does not lead to an atomicity failure (sometimes known as “race”).

As the first step of the proof, we eliminate the universal quantifier from the conjecture, replacing n by a Skolem constant n_0. Then we apply the induction rule natInduction (⇒ Sect. 2.6.4). The induction hypothesis is that $n_0 - k$ transactions have been completed, while k remain (k is the induction variable, $0 \le k \le n_0$):

$$\{\mathtt{sum}:=\sum_{i=1}^{n_0-k}\mathtt{e}(\pi_1(i))\}\langle^{\{k\}}\ll\mathtt{sum=sum+e;}\gg^{\{n_0-k\}}\rangle(\mathtt{sum}=\sum_{i=1}^{n_0}\mathtt{e}(\pi_0(i)))$$

Step case

Now we have to prove that the above holds for $k+1$ transactions, i.e.:

$$\{\mathtt{sum}:=\sum_{i=1}^{n_0-k-1}\mathtt{e}(\pi_1(i))\}\langle^{\{k+1\}}\ll\mathtt{sum=sum+e;}\gg^{\{n_0-k-1\}}\rangle(\mathtt{sum}=\sum_{i=1}^{n_0}\mathtt{e}(\pi_0(i)))$$

We apply the step rule once. There is only one position and thus one relevant permutation, namely π_1. The position is enabled (as $k+1>0$), and there is indeed only one possible choice $\mathcal{P}=1$ (per Axioms (5.2) and (5.3) on page 73). Since there are no if-statements, the path condition is simply *true*. The only remaining goal is thus:

$$\{\mathtt{sum}:=\sum_{i=1}^{n_0-k-1}\mathtt{e}(\pi_1(i))\}\langle\underline{\mathtt{sum=sum+e;}^{*(\pi_1(n_0-k))}}\rangle \langle^{\{k\}}\ll\mathtt{sum=sum+e;}\gg^{\{n_0-k\}}\rangle(\mathtt{sum}=\sum_{i=1}^{n_0}\mathtt{e}(\pi_0(i)))$$

We expand the definition of sequential instantiation. Only the thread-local variable e is affected:

$$\{\mathtt{sum}:=\sum_{i=1}^{n_0-k-1}\mathtt{e}(\pi_1(i))\}\langle\underline{\mathtt{sum=sum+e(}\pi_1(n_0-k)\mathtt{);}}\rangle \langle^{\{k\}}\ll\mathtt{sum=sum+e;}\gg^{\{n_0-k\}}\rangle(\mathtt{sum}=\sum_{i=1}^{n_0}\mathtt{e}(\pi_0(i)))$$

We execute the sequential instantiation of the assignment symbolically using the sequential assignment rule. This generates the update $\{\mathtt{sum}:=\mathtt{sum}+\mathtt{e}(\pi_1(n_0-k))\}$. We have:

$$\{\mathtt{sum}:=\sum_{i=1}^{n_0-k-1}\mathtt{e}(\pi_1(i))\}\{\mathtt{sum}:=\mathtt{sum}+\mathtt{e}(\pi_1(n_0-k))\} \langle^{\{k\}}\ll\mathtt{sum=sum+e;}\gg^{\{n_0-k\}}\rangle(\mathtt{sum}=\sum_{i=1}^{n_0}\mathtt{e}(\pi_0(i)))$$

Update simplification yields:

$$\{\mathtt{sum}:=\sum_{i=1}^{n_0-k}\mathtt{e}(\pi_1(i))\}\langle^{\{k\}}\ll\mathtt{sum=sum+e;}\gg^{\{n_0-k\}}\rangle(\mathtt{sum}=\sum_{i=1}^{n_0}\mathtt{e}(\pi_0(i)))$$

Now, the induction hypothesis for k applies, and the step case of the induction is closed.

Base case

The base case $k=0$ looks like this:

$$\{\mathtt{sum}:=\sum_{i=1}^{n_0}\mathtt{e}(\pi_1(i))\}\langle^{\{0\}}\ll\mathtt{sum=sum+e;}\gg^{\{n_0\}}\rangle(\mathtt{sum}=\sum_{i=1}^{n_0}\mathtt{e}(\pi_0(i)))$$

There are no enabled threads left, so the modality with the program can be removed (rule empty-program), leaving to prove:

$$\{pos(1):=0\,\|\,pos(2):=n_0\}\{\mathtt{sum}:=\sum_{i=1}^{n_0}\mathtt{e}(\pi_1(i))\}(\mathtt{sum}=\sum_{i=1}^{n_0}\mathtt{e}(\pi_0(i)))$$

After applying the inner update the goal is:

$$\{pos(1):=0\,\|\,pos(2):=n_0\}(\sum_{i=1}^{n_0}\mathtt{e}(\pi_1(i))=\sum_{i=1}^{n_0}\mathtt{e}(\pi_0(i)))$$

The sum equality follows from commutativity of addition, the injectivity of π_i (Axiom 5.5), and the fact that $\{\pi_0(1),\dots,\pi_0(n_0)\}=\{\pi_1(1),\dots,\pi_1(n_0)\}$. The latter follows from the definition of position concretization for position 1 (⇒ Def. 5.6):

$$pos_\gamma(1)=\left\{\pi_0(1),\dots,\pi_0(n_0)\right\}\setminus\left\{\pi_1(1),\dots,\pi_1(n_0)\right\} .$$

Taking into account that $pos_\gamma(1)=\varnothing$ (as $pos(1)=0$), we obtain the desired set equality:

$$\left\{\pi_0(1),\dots,\pi_0(n_0)\right\}=\left\{\pi_1(1),\dots,\pi_1(n_0)\right\} .$$

This completes the base case proof.

Use case

By this argument we have established the hypothesis for any $k\le n_0$. Instantiating k with n_0 yields:

$$\{\mathtt{sum}:=\sum_{i=1}^{0}\mathtt{e}(\pi_1(i))\}\langle^{\{n_0\}}\ll\mathtt{sum=sum+e;}\gg^{\{0\}}\rangle(\mathtt{sum}=\sum_{i=1}^{n_0}\mathtt{e}(\pi_0(i)))$$

The sum in the update collapses yielding the Skolemized version of the original conjecture (6.1).

The lessons learned from the example are: We have verified the transaction mechanism for an arbitrary number of threads. This is important, since it is easy to devise code that works for n but not for $n+1$ threads. The state explosion caused by the potentially different ordering of transactions is efficiently controlled, even without further knowledge of concrete data. The scheduling-independence of the system does not require a separate proof before the functional properties can be addressed. Furthermore, it is possible to apply the full power of deductive reasoning about unbounded data and its implementations (e.g., overflow control for the integer variables [Beckert and Schlager, 2005]). ◁

6.4 Treating Locking Primitives

The lock acquisition method is symbolically executed by applying the rule shown in Figure 6.2. The structure of this rule is similar to the step rule for handling normal assignments. Execution is successful if the path condition is satisfied and the statement is enabled (remember, $\mathcal{P}=i$ implies $enabled(i)$).As before, the thread performing the acquire has the id $tid=\pi_i(Post(i)+1)$.

lock

$$\frac{\begin{array}{l} \Longrightarrow \mathcal{P}=i \\[1ex] o^{*(tid)}\texttt{.<lockcount>}=0 \vee o^{*(tid)}\texttt{.<lockedby>}=tid, \\ \quad path(i,p,tid) \Longrightarrow \{o^{*(tid)}\texttt{.<lockcount>} := o^{*(tid)}\texttt{.<lockcount>}+1\} \\ \qquad\qquad\qquad\qquad \{o^{*(tid)}\texttt{.<lockedby>} := tid\} \\ \qquad\qquad\qquad\qquad \langle[\alpha\ ^{\{n-1\}}o\texttt{.<lock>()}^{\{k+1\}}\ \omega]\rangle\phi \\[1ex] \neg path(i,p,tid) \Longrightarrow \langle[\alpha\ ^{\{n-1\}}o\texttt{.<lock>()}^{\{k+1\}}\ \omega]\rangle\phi \end{array}}{\Longrightarrow \langle[\alpha\ \underbrace{^{\{n\}}o\texttt{.<lock>()}^{\{k\}}}_{\text{at position } i \text{ in } p}\ \omega]\rangle\phi}$$

Figure 6.2. The rule for lock acquisition

Note that the mutual-exclusion semantics of locking does not appear in the rule *directly*. Rather, it is hidden in the definition of enabledness (⇒ Def. 5.9, 4.13), which in its turn is part of the axiomatization of position choice $\mathcal{P}$.

A similar rule exists for the `<unlock>()` method (⇒ Fig. 6.3), which decreases the lock count and clears the locked-by status when the count reaches zero. For simplicity we do not clear the `<lockedby>` flag in the calculus, since it does not prevent further acquisition of the lock once `<lockcount>` has reached zero.

Programmers use locking protocols (besides thread-local data) to enforce atomicity of code sections. The easiest way to prove lock-based atomicity with our calculus is by using the invariant rule. We describe this in detail in Section 7.1.

Recognizing Deadlock

The presence of locking opens a possibility for deadlock. Just as the sequential KeY calculus maps abrupt termination onto non-termination, we have decided to model deadlock in the logic as termination. It is still easy to discern a deadlocked state from normal termination by considering the final program configuration. Besides, the desired postcondition would still hold, even if the program becomes prematurely disabled.

unlock

$$\Longrightarrow \mathcal{P}=i$$

$$path(i,p,tid) \Longrightarrow \{o^{*(tid)}.\texttt{<lockcount>} := o^{*(tid)}.\texttt{<lockcount>} - 1\}$$
$$\langle\!\!\langle [\alpha \ {}^{\{n-1\}} o.\texttt{<unlock>()}^{\{k+1\}} \ \omega]\!\!]\rangle\!\!\rangle \phi$$

$$\frac{\neg path(i,p,tid) \Longrightarrow \langle\!\!\langle [\alpha \ {}^{\{n-1\}} o.\texttt{<unlock>()}^{\{k+1\}} \ \omega]\!\!]\rangle\!\!\rangle \phi}{\Longrightarrow \langle\!\!\langle [\alpha \ \underbrace{{}^{\{n\}} o.\texttt{<unlock>()}^{\{k\}}}_{\text{at position } i \text{ in } p} \ \omega]\!\!]\rangle\!\!\rangle \phi}$$

Figure 6.3. The rule for lock release

6.5 An Invariant Rule

So far, we have used induction for verifying full programs. In the following we present a complementary rule invariant, which allows tackling each potentially enabled statement separately. Instead of an induction hypothesis, the user has to state (and then prove) a suitable invariant INV of the system. The rule is:

invariant

$$\Gamma \Longrightarrow \mathcal{U} INV, \Delta$$

$$INV, \ \mathcal{P}=0 \Longrightarrow \phi$$

$$INV, \ path(1,p,tid(1)), \ enabled(1) \Longrightarrow$$
$$\langle\!\!\langle [\underline{p_1^{*(tid(1))}}]\!\!]\rangle\!\!\rangle \{pos(1) := pos(1) - 1\}\{pos(2) := pos(2) + 1\} INV$$
$$\vdots$$
$$INV, \ path(q,p,tid(q)), \ enabled(q) \Longrightarrow$$
$$\frac{\langle\!\!\langle [\underline{p_q^{*(tid(q))}}]\!\!]\rangle\!\!\rangle \{pos(q) := pos(q) - 1\}\{pos(q+1) := pos(q+1) + 1\} INV}{\Gamma \Longrightarrow \mathcal{U} \langle\!\!\langle [p]\!\!]\rangle\!\!\rangle \phi, \Delta} \quad (*)$$

We assume that the program p has q positions, and $p_i^{*(tid(i))}$ is the sequential instantiation (⇒ Def. 4.7) of the atomic program at position i in p. The id of the thread executing the instantiation is as usual: $tid(i) = \pi_i(Post(i) + 1)$.

The first premiss of the rule states that the system satisfies the invariant in its initial configuration. The second premiss states that the invariant implies the desired property, once no thread is longer enabled. What follows are q premisses—one for each position in the program—stating that the "sequential" execution of the atomic statement at this position preserves the invariant. For each position we can assume its enabledness predicate and the corresponding path condition.

Comparison to loop invariants

At this point it is natural to compare the above invariant rule to the standard loop invariant rule (⇒ Sect. 2.8). First, while a loop only has one degree of freedom (the execution of the loop body), a concurrent program has one degree of freedom for each potentially enabled position. Every executed statement brings the system into a new state, and, thus, has to be shown as invariant-preserving. Second, the concurrent invariant formula can—and most probably will—contain control variables, which correspond to the loop counter. Third, our invariant rule is sound for the diamond modality even without a special termination argument. The only potential sources of non-termination are loops, which we assume as atomic, and the sequential calculus fragment is sound and complete for these. For this reason, the above invariant rule is also not needed for the completeness of the concurrent calculus.

6.6 Remarks on Calculus Soundness

The soundness of a verification calculus—together with the adequacy of the underlying programming language theory—is an issue of great importance. We have validated our calculus (and its implementation) by extensive testing. As with the sequential calculus of KeY we have not performed a formal/mechanized soundness proof. The reason for this decision is a resource trade-off, and Chapter 9 is dedicated to explaining this trade-off in detail.

We did though state in the previous chapters a formal semantics of the logic. Among other things, the semantics defines the scheduler axioms, which are used by the calculus. In fact, we did state two versions of semantics: one with explicit thread ids (⇒ Chap. 4) and one with permutations (⇒ Chap. 5). This approach has helped us to separate concerns present in developing a general program logic with a deterministic scheduler and later one with symmetry reduction. It is of course an interesting question why the latter logic correctly simulates the former.

The key to answering this question is in the configuration concretization function (⇒ Def. 5.6) and the scheduler decomposition equality (5.1). The configuration concretization function explains how every configuration with permutations can be translated into a configuration with concrete tids. The scheduler decomposition explains the same for the scheduler function. Both translations are quite simple, and allow us to fall back on many common definitions in both logics.

Ultimately, of course, it is impossible to relate formal and informal artifacts formally. Thus, there can be no formal proof that any of these semantics are adequate w.r.t. the Java Language Specification. It is also impracticable to prove that they conform with the implementation of any given compiler and JVM. In this light, Chapter 9 explains why testing is necessary for obtaining a reliable reasoning system. The sequential KeY calculus is automatically tested with the compiler test suite [Jacks] on a regular basis. For reasons probably related to nondeterminism, such test suites do not include concurrent programs. This problem does not arise when "running" a program in a verification system though, as one can usually make assertions over all threads

$$\begin{tikzcd}
\dots \\
\text{compiler and JVM} \arrow[u] \\
\text{JLS} \arrow[u, "\text{correctly describes the language implemented by}"'] \\
\text{MODL semantics with explicit thread ids (Ch. 4)} \arrow[u, "\text{adequate}"'] \\
\text{MODL semantics with permutations (Ch. 5)} \arrow[u, "\text{simulates correctly}"'] \\
\text{calculus} \arrow[u, "\text{sound}"'] \\
\text{prover} \arrow[u, "\text{implements correctly}"'] \\
\dots \arrow[u]
\end{tikzcd}$$

Figure 6.4. Reliability of reasoning: artifacts and their relations

and/or interleavings. Thus, a suite of small programs (and reference results) for testing how verification systems treat concurrency primitives is indeed feasible. Such a test suite would be of great benefit to the field.[1]

[1] We have started building such a benchmark/test suite in the context of the COST Action IC0701 "Formal Verification of Object-oriented Software" (`http://www.cost-ic0701.org`).

7

Extensions and Refinements

7.1 Proving Atomicity with Invariants

A method or code block is *atomic* if its execution is not affected by and does not interfere with concurrently-executing threads [Flanagan and Freund, 2004]. There are two main reasons for wanting to establish atomicity of code sections:

- One reason roots in the limitation of our calculus that all loops must be atomic (i.e., appear within atomic blocks). In real Java programs, atomicity of code sections is implemented implicitly with locking or thread-local data encapsulation. Thus, it is necessary to prove that every such implementation is indeed correct, and no unsoundness is introduced by putting loops into explicit atomic blocks.
- The second reason is to coarsen the interference granularity of programs and simplify reasoning about their concurrent behavior. It is often useful to separate concerns, i.e., to establish atomicity of code sections first, and then use this fact in further proof of functional correctness.

So far we have relied on atomicity as a proviso imported into the proof (possibly established by some other tool). Here's how we can prove atomicity in a deductive verification framework.

To restate the definition more formally, a code block β is atomic if for every program execution with final state s there is some equivalent (i.e., also ending in s) execution, where β is executed without interruption. It is actually possible to specify the atomicity definition of β as a formula of our logic:

$$\forall v. \big(\langle \alpha \;\; \beta \;\; \omega \rangle (\mathtt{x} = v) \longrightarrow \exists s. \{ \sigma := s \} \langle \alpha \;\; \ll\!\beta\!\gg \;\; \omega \rangle (\mathtt{x} = v) \big) \ ,$$

where σ is the scheduling seed (cf. scheduler axioms on page 61), and x is the only location that β modifies (assumed here without loss of generality).

In theory, we could establish atomicity by proving the above formula. There are two hurdles though. First, we have not stated a calculus for scheduling seeds, so there is (currently) no way to eliminate the existential quantifier. Second, this method does not work for proving atomicity of loops, since we cannot reason about the program in the antecedent formula, as it is then not syntactically valid. To sidestep these problems we now show how to check sufficient conditions for atomicity of code sections guarded by locking.

A sufficient condition for atomicity of β is:

$$\sum_{i \in C(\beta)} pos(i) \leq 1 \ ,$$

where $C(\beta)$ is the set containing at least (a) all program positions of β and (b) all positions that access the same shared state as β[1]. This condition ensures that whenever some thread could execute a statement potentially interfering with β, β has either not yet started or has already finished. The condition can be proven with help of a simple invariant as illustrated by the following example.

Example 7.1. We want to use the invariant rule to establish atomicity of the following code section protected by locking[2]:

```
o.<lock>(); a=o.sum+e; o.sum=a; o.<unlock>();
```

Since this is the only critical section in the system, the atomicity condition is:

$$N \leq 1, \text{ where } N = \sum_{i=2}^{q} pos(i) \ ,$$

which states that the configuration never has more than one thread between its second and the last position. q is here the number of statements; for the above code, $q = 4$. Before the atomicity proof can succeed, the above invariant has to be strengthened to

$$INV = N \leq 1 \wedge (N = \texttt{o.<lockcount>}) \ .$$

Applying the invariant rule produces 6 premisses:

(1a) This premiss states that the invariant holds in the initial state. We assume that the initial state satisfies `o.<lockcount>=0` and that the initial configuration is:

$$^{\{n\}}\texttt{o.<lock>();}^{\{0\}}\texttt{a=o.sum+e;}^{\{0\}}\texttt{o.sum=a;}^{\{0\}}\texttt{o.<unlock>();}^{\{0\}} \ .$$

The invariant clearly holds then, since both $N = \sum_{i=2}^{4} pos(i)$ and `o.<lockcount>` are zero.

(1b) We ignore the premiss $INV \longrightarrow \phi$, since we are only interested in the maintenance of the invariant.

(1) This premiss has to show that the locking statement at position 1 preserves the invariant. In order for the statement to be enabled at all, the lock must be available before execution. Then, $N = 0$ before the locking per the second conjunct of the invariant, which we can assume in the pre-state. After the execution, both N and `o.<lockcount>` are equal to 1 (⇒ Fig. 7.1 for a more detailed proof of invariance).

[1] This information may be available in form of modifies/reads clauses.

[2] To achieve proper mutual exclusion without complicating the proof obligation we assume in this example that the variable `o` is static. In general, this variable would have to be local, and the calculus would check that all threads lock the same object.

The premiss of the rule for position 1 is the sequent:

$$INV,\ path(1, p, tid(1)),\ enabled(1) \Longrightarrow$$
$$\langle \underline{p_1^{*(tid(1))}} \rangle \{pos(1) := pos(1) - 1\}\{pos(2) := pos(2) + 1\} INV$$

Here $tid(1) = \pi_1(\sum_{i=2}^{5} pos(i) + 1)$, but this is irrelevant, since no thread-local data is involved. Expanding the definitions yields:

$$(N \leq 1) \wedge (N = \texttt{o.<lockcount>}),$$
$$true,\ (pos(1) > 0) \wedge (\texttt{o.<lockcount>} = 0) \Longrightarrow$$
$$\langle \underline{\texttt{o.<lock>();}} \rangle \{pos(1) := pos(1) - 1\}\{pos(2) := pos(2) + 1\}\big((N \leq 1)$$
$$\wedge (N = \texttt{o.<lockcount>})\big)$$

After using $\texttt{o.<lockcount>} = 0$ from the enabledness condition for rewriting the invariant in the antecedent we obtain:

$$N \leq 1,\ N = 0,\ pos(1) > 0,\ \texttt{o.<lockcount>} = 0 \Longrightarrow$$
$$\langle \underline{\texttt{o.<lock>();}} \rangle \{pos(1) := pos(1) - 1\}\{pos(2) := pos(2) + 1\}\big((N \leq 1)$$
$$\wedge (N = \texttt{o.<lockcount>})\big)$$

Symbolic execution and update application increases both N and `o.<lockcount>` by one, while the update to *pos* is irrelevant:

$$N \leq 1,\ N = 0,\ pos(1) > 0,\ \texttt{o.<lockcount>} = 0 \Longrightarrow$$
$$\big((1 + N \leq 1) \wedge (1 + N = \texttt{o.<lockcount>} + 1)\big)$$

After rewriting the succedent with both equalities from the antecedent, we have:

$$0 \leq 1,\ N = 0,\ pos(1) > 0,\ \texttt{o.<lockcount>} = 0 \Longrightarrow (1 + 0 \leq 1) \wedge (1 + 0 = 0 + 1)\ ,$$

which clearly holds.

Figure 7.1. Atomicity proof with invariant, premiss for position 1

(2) The statements at position 2 preserves the invariant, since it can neither change the value of N nor of `o.<lockcount>`. Only the statements at positions 1 and 4 change these (⇒ Fig. 7.2 for a more detailed proof of invariance).
(3) Position 3: same as premiss (2).
(4) Position 4: analogical to premiss (1).

Thus, the locking works correctly, and the code section is atomic. Once a thread has entered the section it will run to completion without interference. We can use this fact to simplify further reasoning. ◁

The above method allows us to prove atomicity of β in

$$o.\texttt{<lock>();}\ \beta;\ o.\texttt{<unlock>();}$$

regardless of what β is, as long as it does not contain locking operations. This, in turn, can be established by an easy syntactic check. We can even prove that a loop is atomic, if we ignore for a moment that a loop without an enclosing atomic block is not a syntactically valid program.

The premiss of the rule produces the sequent:

$$INV,\ path(2,p,tid(2)),\ enabled(2) \Longrightarrow \langle \underline{p_2^{*(tid(2))}} \rangle \{pos(2) := pos(2) - 1\}\{pos(3) := pos(3) + 1\} INV$$

Here $tid(2) = \pi_2(pos(3) + pos(4) + pos(5) + 1)$, but we will not need this for the further proof. Expanding other definitions yields:

$$(N \le 1) \wedge (N = \texttt{o.<lockcount>}),\ true,\ pos(2) > 0 \Longrightarrow \langle \underline{\texttt{a=o.sum+e;}^{*(tid(2))}} \rangle \{pos(2) := pos(2) - 1\}\{pos(3) := pos(3) + 1\}\big((N \le 1) \wedge (N = \texttt{o.<lockcount>})\big)$$

which simplifies to:

$$(N \le 1) \wedge (N = \texttt{o.<lockcount>}),\ pos(2) > 0 \Longrightarrow \langle \underline{\texttt{a=o.sum+e;}^{*(tid(2))}} \rangle \{pos(2) := pos(2) - 1\}\{pos(3) := pos(3) + 1\}\big((N \le 1) \wedge (N = \texttt{o.<lockcount>})\big)$$

The symbolic execution of the sequential program in the diamond does not affect the postcondition (as it does not contain a). The update has no effect either, since both $pos(2)$ and $pos(3)$ are summands in N. Altogether:

$$(N \le 1) \wedge (N = \texttt{o.<lockcount>}),\ pos(2) > 0 \Longrightarrow (N \le 1) \wedge (N = \texttt{o.<lockcount>})\ ,$$

which trivially holds.

Figure 7.2. Atomicity proof with invariant, premiss for position 2

7.2 Treating Condition Variables

Introduction

In the following we present a calculus extension for verifying programs with condition variables. This is the most complex part of our calculus and its status must still be considered experimental. We also make a number of simplifications and restrictions.

The biggest restriction is due to the fact that a correct implementation of a condition variable in Java requires a non-atomic loop, which we cannot (yet) treat in our framework. On the other hand, for conditions that are atomic, we can consider the whole wait-in-loop idiom as one atomic statement. Many programs in practice satisfy these requirements. Such programs can be verified with this calculus. We actually make an even stronger restriction, demanding that evaluating the condition does not change the state.

To simplify matters further, we demand that all threads synchronize on the same object and that the condition is uniform (i.e., if one thread satisfies it, then all do). This is the case when the condition is expressed in terms of a shared data structure.

Since we do not allow thread identities in programs, one cannot `interrupt()` a thread. Thus, we do not model the case when a thread exits a condition variable with an `InterruptedException`. Unsurprisingly, we also do not allow the use of the `wait(long timeout)` method, since our framework has no notion of real time.

Additional Means of Expression

We package the common implementation of a condition variable in a special ghost method `void <waitUntil>(boolean b)`, which we make part of the `Object` class. The actual Java implementation to be verified is replaced by this method during the unfolding stage of the verification process. The method has as parameter a boolean condition, which must evaluate to true for a thread to proceed (it is the negated condition of the condition-testing while loop in the original program).

An example of the unfolding is given in Figures 7.3 and 7.4.

We need some means to differentiate between threads that are ready to execute `<waitUntil>(b)` and threads that have suspended their execution until a notification arrives. We employ the ghost field `<waiting>` present in every object to keep track of the number of suspended threads. If the program has more than one `wait()` on (potentially) the same object then position-indexed `<waiting>` fields have to be used.

An important question is when a position with `<waitUntil>(b)` is enabled. We recall that

$$enabled(i) = \exists t. \left(t \in pos_{\gamma}(i) \wedge \left(path(i,p,t) \longrightarrow enabled(p(i),t)\right)\right)$$

and proceed to extend the predicate $enabled(s,t)$.

Definition 7.2 (Enabledness of `<waitUntil>()`). We extend Def. 4.13 of statement enabledness as follows. For the o.`<waitUntil>(b)` statement, we define the predicate to be:

$$enabled(o.\texttt{<waitUntil>(b)}, t) = \\ (o(t).\texttt{<lockcount>} = 0 \vee o(t).\texttt{<lockedby>} = t) \ ,$$

where t is the executing thread id. ◁

```
public synchronized Object take()
                              throws InterruptedException {
    try {
        while (count == 0)
            wait();
    } catch (InterruptedException ie) {
        // return without removing
    }
    Object x = extract(); // decreases count
    return x;
}
```

Figure 7.3. Element removal method from a blocking concurrent queue (slightly adapted from `java.util.concurrent.ArrayBlockingQueue`). If the queue is empty, a consumer thread blocks until an element is put into the queue

```
q.<lock>();
waitUntil( count!=0 );
items=this.items;
x_1=items[takeIndex];
items[takeIndex]=null;
i=takeIndex;
j_4=i+1;
i=j_4;
j_2=i;
j_3=items.length;
b=j_2==j_3;
if (b) {
  j_1=0;
} else {
  j_1=i;
}
takeIndex=j_1;
j_5=count-1;
count=j_5;
q.notifyAll();
res=x_1;
q.<unlock>();
```

Figure 7.4. Element removal method after unfolding

Incidentally, this is the same condition as for lock acquisition. The difference (i.e., the fact that some threads may have suspended execution) is hidden in the definition of pos_γ (⇒ Def. 5.6), which we will adapt next.

First though, we need to introduce yet another "permutation" function. For each position i with a `<waitUntil>(b)`, there is a non-rigid function symbol

$$\pi_{i'} : \mathbb{N} \to \mathcal{T} \enspace .$$

The function stores the ids of the threads that have suspended execution since the last `notifyAll()`.

Definition 7.3 (Configuration concretization for condition variables). We amend the Definition 5.6 as follows. If the statement o`.<waitUntil>(b)` occupies position i in a program, then

$$\begin{aligned} pos_\gamma(i) = \Bigg(\Bigg\{ & \pi_{i-1}(1), \dots, \pi_{i-1}(Post(i-1)) \Bigg\} \setminus \\ & \Bigg\{ \pi_i(1), \dots, \pi_i(Post(i)) \Bigg\} \Bigg) \setminus \\ & \Bigg\{ \pi_{i'}(1), \dots, \pi_{i'}(o(t)\texttt{.<waiting>}) \Bigg\} \enspace . \end{aligned}$$

◁

The third subterm is new and expresses the fact that suspended threads are unavailable for scheduling.

The usual axioms still hold, in particular Axiom (5.6):

$$\pi_i(Post(i) + 1) \in pos_\gamma(i) \enspace .$$

There is now one additional axiom

$$\pi_{i'}(o(t)\texttt{.<waiting>} + 1) \in pos_\gamma(i) \tag{7.1}$$

constraining the id of a thread next to suspend execution.

Please note that $\pi_{i'}$ is non-rigid and depends on the state in which it is evaluated. In particular, its interpretation depends on the value of $pos(j)$, where j is the position of `notifyAll()`. One can think of $pos(j)$ as an "invisible parameter" to $\pi_{i'}$. Performing a `notifyAll()` cleans the slate and gives us a "fresh" $\pi_{i'}$, with a possibly different ordering of threads suspending.

The Rules for Symbolic Execution

We start with a rule for `notifyAll()`, which is shown in Figure 7.5. If this statement is enabled (first premiss), and the path condition is satisfied, the rule wakes up all suspended threads by setting the `<waiting>` counter to zero (second premiss). If the path condition is not satisfied, the statement is a no-op.

$$\Longrightarrow \mathcal{P} = i$$

$$path(i,p,tid) \Longrightarrow \{o^{*(tid)}\texttt{.<waiting>} := 0\} \langle\!\![\alpha \ ^{\{n-1\}} o\texttt{.notifyAll()}^{\{k+1\}} \ \omega]\!\!\rangle\phi$$

$$\text{notifyAll} \ \frac{\neg path(i,p,tid) \Longrightarrow \langle\!\![\alpha \ ^{\{n-1\}} o\texttt{.notifyAll()}^{\{k+1\}} \ \omega]\!\!\rangle\phi}{\Longrightarrow \langle\!\![\alpha \ ^{\{n\}} \ \underbrace{o\texttt{.notifyAll()}}_{\text{at position } i \text{ in } p} \ ^{\{k\}} \ \omega]\!\!\rangle\phi}$$

Figure 7.5. The rule for notification

$$\Longrightarrow \mathcal{P} = i$$

$$\Longrightarrow \Phi \longleftrightarrow \langle \underline{\texttt{boolean x = } b^{*(tid)}\texttt{;}} \rangle \texttt{x} = true$$

$$path(i,p,tid), \quad \Phi, \ o^{*(tid)}\texttt{.<lockcount>} > 0, \ o^{*(tid)}\texttt{.<lockedby>} \in pos_\gamma(i) \Longrightarrow \langle\!\![\alpha \ ^{\{n-1\}} o\texttt{.<waitUntil>(}b\texttt{)}^{\{k+1\}} \ \omega]\!\!\rangle\phi$$

$$path(i,p,tid), \quad \Phi, \ o^{*(tid)}\texttt{.<lockcount>} = 0 \Longrightarrow \{o^{*(tid)}\texttt{.<lockcount>} := depth(tid)\} \{o^{*(tid)}\texttt{.<lockedby>} := tid\} \langle\!\![\alpha \ ^{\{n-1\}} o\texttt{.<waitUntil>(}b\texttt{)}^{\{k+1\}} \ \omega]\!\!\rangle\phi$$

$$path(i,p,tid), \ \neg\Phi, \ o^{*(tid)}\texttt{.<lockcount>} > 0, \ o^{*(tid)}\texttt{.<lockedby>} \in pos_\gamma(i) \Longrightarrow \{o^{*(tid)}\texttt{.<waiting>} := o^{*(tid)}\texttt{.<waiting>} + 1\} \{depth(tid) := o^{*(tid)}\texttt{.<lockcount>}\} \{o^{*(tid)}\texttt{.<lockcount>} := 0\} \langle\!\![\alpha \ ^{\{n\}} o\texttt{.<waitUntil>(}b\texttt{)}^{\{k\}} \ \omega]\!\!\rangle\phi$$

$$path(i,p,tid), \ \neg\Phi, \ o^{*(tid)}\texttt{.<lockcount>} = 0 \Longrightarrow \{o^{*(tid)}\texttt{.<waiting>} := o^{*(tid)}\texttt{.<waiting>} + 1\} \langle\!\![\alpha \ ^{\{n\}} o\texttt{.<waitUntil>(}b\texttt{)}^{\{k\}} \ \omega]\!\!\rangle\phi$$

$$\text{waitUntil} \ \frac{\neg path(i,p,tid) \Longrightarrow \langle\!\![\alpha \ ^{\{n-1\}} o\texttt{.<waitUntil>(}b\texttt{)}^{\{k+1\}} \ \omega]\!\!\rangle\phi}{\Longrightarrow \langle\!\![\alpha \ ^{\{n\}} \underbrace{o\texttt{.<waitUntil>(}b\texttt{)}^{\{k\}}}_{\text{position } i \text{ in } p} \ \omega]\!\!\rangle\phi}$$

Figure 7.6. The rule for `<waitUntil>()`

Now we look at the rule for symbolic execution of `<waitUntil>()` given in Figure 7.6. The first premiss demands, among other things, that the position in question is enabled: $\mathit{enabled}(i)$ must hold.

The second premiss captures the condition Φ of the condition variable. Φ can be $\langle \underline{\texttt{boolean x =}b^{*(tid)}\texttt{;}} \rangle \texttt{x} = \mathit{true}$ or its first-order equivalent. Note that the diamond formula is purely sequential and $b^{*(tid)}$ is the sequential instantiation of b for the next thread to run at position i (i.e., thread with id $\pi_i(\mathit{Post}(i) + 1)$). In the case of the blocking queue, Φ is simply $\texttt{count} \neq 0$.

The third premiss assumes that the condition Φ is satisfied and there is one non-suspended thread (which holds the lock for `o`). This thread then proceeds past the `<waitUntil>()`.

The fourth premiss assumes that Φ holds, but no thread holds the lock of `o`. In this case one of the recently awakened threads contending for the lock (there must be at least one, otherwise $\mathcal{P} \neq i$) is successful and proceeds.

The fifth premiss assumes that the condition Φ does not hold, while one thread holds the lock. In this case there is no thread movement (the configuration does not change), but the number of suspended threads o.`<waiting>` increases by one. The lock is released.

The sixth premiss assumes that the condition Φ does not hold, while no thread currently holds the lock. In this case one of the recently awakened threads returns to suspended state.

The seventh and final premiss deals with the negative path condition. In this case, just as with other rules, the thread executes a no-op.

7.3 Proving Absence of Data Races and JMM-Safety

According to the Java Memory Model, updates to shared state performed by one thread need not become immediately visible to other threads. Even worse, updates need not become visible to other threads in the order they have been made. Since this state of affairs puts a high burden on the programmer, the JMM describes a sufficient condition for attaining sequentially consistent program behavior. This condition, known as the DRF guarantee, can be stated as follows:

> If every sequentially consistent execution of a program is free of data races, these are all the executions allowed for that program. [Huisman and Petri, 2007]

Sequentially consistent executions are exactly executions built by thread interleaving. It is our goal to check that the program has no other executions, since (a) this is what programmers usually expect and (b) these are exactly the executions considered by our verification calculus. To achieve this we need to establish that all interleaving executions are free of data races.

In the following, we extend our calculus with explicit checks for data races. Without them, a verified program could still contain "benign" data races or data races in parts not covered by the specification. In the light of the JMM, these races become a problem. For the sake of completeness it should be noted though that the absence of data races does not entail the correctness of the program.

We use a definition of data race slightly adapted from [Ševčík, 2008].

Definition 7.4 (Data race). An interleaving contains a data race iff it contains two actions a_1 and a_2 such that:

1. a_1 and a_2 are performed by two different threads on the same shared location
2. a_1 is a write
3. there is no synchronization link between the two, i.e., there are no actions b_1 and b_2 between a_1 and a_2 (it is allowed that $a_1 = b_1$ and $a_2 = b_2$) such that either
   (a) b_1 is an unlock on object m and b_2 is a lock on object m or
   (b) b_1 is a write to some volatile location v and b_2 is a read from v $\lhd$

All currently available functional verifications systems for multi-threaded JAVA-like languages are either unsound (they assume a sequentially consistent semantics) or incomplete (they can only detect synchronization links of type 3(a)). The calculus extension presented below is able to detect synchronization links of both types.

Consider the code in Figure 7.7. One thread executes the method `one()` while another thread concurrently executes `two()`. In a sequentially consistent model this is perfectly safe. Under the JMM, this code can throw a `NullPointerException` due to `two()` seeing a partially constructed object. This happens when the update to `instance` by thread one has already propagated to thread two but not yet the update to `name`.

One way to avoid this is to declare `instance` volatile. This would create a synchronization edge between writing `instance` in `one()` and reading it in `two()`. The edge would ensure that all updates made during the process of initialization by one thread before it writes `instance` are visible to the other thread after the other reads `instance`. `two()` will either see `instance` as `null` or pointing to a completely constructed object. This pattern is also known as *safe one-time publication* [Goetz et al., 2006].

Note 7.5 (Data races vs. functional correctness). Concurrency texts and programmers are often preoccupied with data races (or "race conditions"). Inside the JMM domain, this term is used in a strict and "technical" sense; outside it is often a catch-all phrase for concurrency problem. The latter view of a data race is problematic for two reasons:

- There are correct programs with data races. Representatives of this program class usually deal with a stream of data values entering from the environment.
- There are incorrect programs without race conditions. Incorrect in the concurrency part, that is. Also, any program can be automatically made data-race-free by enclosing every single access to shared data in a synchronized block.

The above "fix" makes it clear that the problem lies in the granularity of access. For this reason, we advocate—outside of the JMM domain—the term *atomicity failure* as a replacement for "data race". Furthermore, it should be pointed out that the

```
class Foo {

    private String name = "Foo";
    static Foo instance;

    static void one() {
        if (instance==null) instance = new Foo(); }

    static void two() {
        if (instance!=null) System.out.println(instance.name.length());
    }

}
```
JAVA

Figure 7.7. Code, surprisingly prone to failure under the JMM

correct level of atomicity cannot be established without considering a particular application domain. There can be no universal, application-independent atomicity failure checker. What can be checked meaningfully is whether a program conforms to its specification, or to a certain convention, or that it accesses shared state in an internally consistent manner. ◁

Calculus extension to prove JMM-safety

Definition 7.6 (Heap locations, synchronization edges). Let v be a local reference variable, C a class name and a an attribute name in a given program. The set of heap locations Loc is the set of program-compatible pairs:

- (v, a), also written $v.a$, for every instance field
- (C, a), also written $C.a$, for every static field.

The set of synchronization edges $Edge$ is the set of program-compatible tuples of the form:

- (v), also written v, called lock/unlock
- (v, a), also written $v.a$, called instance volatile read/write
- (C, a), also written $C.a$, called static volatile read/write. ◁

Note 7.7 (Quantifying over locations). In order to quantify over locations (and edges) we have to resort to a trick, since attribute and class names are not first-class citizens of the underlying KeY logic. We encode heap locations $v.a$ as pairs $(v, a^\sharp)$, where v is the reference to the object and $a^\sharp$ is a natural-number hash of the attribute name a. Thus, quantifying over such locations amounts to quantifying over pairs of objects and natural numbers. A similar scheme applies to static locations. ◁

To detect data races (or more precisely, absence of synchronization edges), we define two auxiliary non-rigid function symbols:

$$\mathit{lastwrittenby}\colon \mathit{Loc} \to \mathcal{T} \quad ,$$

which keeps track of which thread was the last to update a heap location, and

$$\mathit{mem}\colon \mathit{Loc}, \mathcal{T}, \mathcal{T}, \mathit{Edge} \to \{\mathit{dirty}, \mathit{flushed}, \mathit{visible}\} \quad ,$$

which tracks the visibility status of a heap location between two threads—the writer and a reader.

$\mathit{mem}(l, t_1, t_2, e) = \mathit{dirty}$ means that the last update of the location l performed by t_1 need not be visible to t_2 via synchronization edge e (i.e., t_1 has updated l after its last release on e). The value *flushed* means the same, but in addition we know that t_1 has performed the release part of the edge e and has not changed l afterwards. t_2 will see the newest value of a flushed location after performing a corresponding acquire on e. Finally, the value *visible* means that the latest update by t_1 is visible to t_2 via e.

The JMM-safe rules for symbolic execution of assignments are presented in Figure 7.8. These rules supersede the basic step rule presented in Section 6.3. Their structure is quite similar to step, but there are separate rules for reading and writing volatile locations as well as for non-volatile. The presented rules only treat instance fields, but the rules for static fields are a direct analogon.

The most interesting part is the second premiss of the rule step_read_normal. This premiss becomes relevant when a thread is trying to access a non-volatile heap location that has been previously updated by another thread. To discharge this premiss it is then necessary to show that there has been a synchronization edge between the two threads in-between. The presence of a synchronization edge (resp. its source and sink) is recorded by the rules step_write_volatile and step_read_volatile. The rule step_write_normal, on the other hand, excludes the location it updates from the scope of preceding synchronization edges.

Finally, the following modifications complete the calculus:

- The rules unlock and lock (⇒ Sect. 6.4) are modified to include the updates similar to rules step_write_volatile and step_read_volatile.
- As part of the instance initialization process, KeY executes code that assigns default values to object fields. These assignments must be executed by a special rule that (a) does not check visibility w.r.t. a previous write to the field (there are none) and (b) establishes a visibility relation with all subsequent accesses to the field. The JLS guarantees that default values are always visible without further action by the programmer.

7.4 Further Extensions and Future Work

Arbitrary systems, pl.n.: Systems about which nothing general can be said, save "nothing general can be said".

step_write_normal

$$
\begin{array}{l}
\qquad\qquad \Longrightarrow \mathcal{P} = i \\
path(i,p,tid),\ lastwrittenby(v.a^{\sharp}) \neq tid \Longrightarrow \\
\qquad\qquad \exists e.\, mem(v.a^{\sharp}, lastwrittenby(v.a^{\sharp}), tid, e) = visible \\
path(i,p,tid) \Longrightarrow \langle[\underline{v.a^{*(tid)}{=}se^{*(tid)}}]\rangle \\
\qquad\qquad \{lastwrittenby(v.a^{\sharp}) := tid\} \\
\qquad\qquad \{\texttt{for}\ t, e;\ t \neq tid;\ mem(v.a^{\sharp}, tid, t, e) := dirty\} \\
\qquad\qquad \langle[\alpha\ ^{\{n-1\}} v.a{=}se^{\{k+1\}}\ \omega]\rangle \phi \\
\neg path(i,p,tid) \Longrightarrow \langle[\alpha\ ^{\{n-1\}} v.a{=}se^{\{k+1\}}\ \omega]\rangle \phi \\
\hline
\qquad\qquad \Longrightarrow \langle[\alpha\ ^{\{n\}}\ \underbrace{v.a = se}_{\substack{\text{position } i \text{ in } p, \\ a \text{ is non-volatile}}}\ ^{\{k\}}\ \omega]\rangle \phi
\end{array}
$$

step_read_normal

$$
\begin{array}{l}
\qquad\qquad \Longrightarrow \mathcal{P} = i \\
path(i,p,tid),\ lastwrittenby(v.a^{\sharp}) \neq tid \Longrightarrow \\
\qquad\qquad \exists e.\, mem(v.a^{\sharp}, lastwrittenby(v.a^{\sharp}), tid, e) = visible \\
path(i,p,tid) \Longrightarrow \langle[\underline{v_0^{*(tid)}{=}v.a^{*(tid)}}]\rangle \langle[\alpha\ ^{\{n-1\}} v_0{=}v.a^{\{k+1\}}\ \omega]\rangle \phi \\
\neg path(i,p,tid) \Longrightarrow \qquad\qquad \langle[\alpha\ ^{\{n-1\}} v_0{=}v.a^{\{k+1\}}\ \omega]\rangle \phi \\
\hline
\qquad\qquad \Longrightarrow \langle[\alpha\ ^{\{n\}}\ \underbrace{v_0 = v.a}_{\substack{\text{position } i \text{ in } p, \\ a \text{ is non-volatile}}}\ ^{\{k\}}\ \omega]\rangle \phi
\end{array}
$$

step_write_volatile

$$
\begin{array}{l}
\qquad\qquad \Longrightarrow \mathcal{P} = i \\
path(i,p,tid) \Longrightarrow \langle[\underline{v.a^{*(tid)}{=}se^{*(tid)}}]\rangle \\
\qquad\qquad \{\texttt{for}\ l, t;\ mem(l, tid, t, v.a^{\sharp}) = dirty;\ mem(l, tid, t, v.a^{\sharp}) := flushed\} \\
\qquad\qquad \langle[\alpha\ ^{\{n-1\}} v.a{=}se^{\{k+1\}}\ \omega]\rangle \phi \\
\neg path(i,p,tid) \Longrightarrow \langle[\alpha\ ^{\{n-1\}} v.a{=}se^{\{k+1\}}\ \omega]\rangle \phi \\
\hline
\qquad\qquad \Longrightarrow \langle[\alpha\ ^{\{n\}}\ \underbrace{v.a = se}_{\substack{\text{position } i \text{ in } p, \\ a \text{ is volatile}}}\ ^{\{k\}}\ \omega]\rangle \phi
\end{array}
$$

step_read_volatile

$$
\begin{array}{l}
\qquad\qquad \Longrightarrow \mathcal{P} = i \\
path(i,p,tid) \Longrightarrow \langle[\underline{v_0^{*(tid)}{=}se^{*(tid)}}]\rangle \\
\qquad\qquad \{\texttt{for}\ l, t;\ mem(l, t, tid, v.a^{\sharp}) = flushed;\ mem(l, t, tid, v.a^{\sharp}) := visible\} \\
\qquad\qquad \langle[\alpha\ ^{\{n-1\}} v_0{=}v.a^{\{k+1\}}\ \omega]\rangle \phi \\
\neg path(i,p,tid) \Longrightarrow \langle[\alpha\ ^{\{n-1\}} v_0{=}v.a^{\{k+1\}}\ \omega]\rangle \phi \\
\hline
\qquad\qquad \Longrightarrow \langle[\alpha\ ^{\{n\}}\ \underbrace{v_0 = v.a}_{\substack{\text{position } i \text{ in } p, \\ a \text{ is volatile}}}\ ^{\{k\}}\ \omega]\rangle \phi
\end{array}
$$

Figure 7.8. JMM-faithful rules for read and write access

Extending to non-atomic loops

The restriction of only dealing with atomic loops is admittedly quite unsatisfying. An obvious line of research would be towards overcoming this limitation.

The problem with non-atomic loops is that they cause threads to "jump back" in the program (in an observable way). On the other hand, the configuration concretization (⇒ Def. 5.6), which is at the core of our model, depends on the fact that the number of threads past a given position never decreases.

One possible way to attack the loop problem is by giving the thread choice functions a parameter: an iteration counter. Another possibility could be to model the loop in such a way that each iteration is run by a dedicated thread. The latter involves turning the threads on and off at the right moment as well as "handing over" the thread-local data.

On the other hand, even programs with non-atomic loops can already be meaningfully verified. A typical such program is a server:

```
while (true) {
    Socket socket = serverSocket.accept();
    new Thread(new Handler(socket)).start();
}
```
Java

Instead of including the listening loop in verification, it is often sufficient to cut off the loop and verify the correctness of n handler threads running in parallel. The proof obligation then looks like this:

$$\forall n.\big(n \geq 0 \longrightarrow \langle{}^{\{n\}} handler^{\{0\}}\rangle \forall i.\,(1 \geq i \geq n \longrightarrow output(i) = f(input(i)))\big) \enspace .$$

Here *handler* is the body of the `run()` method of the `Handler` class. The postcondition asserts that the output of each threads is correctly related to its input (which may be the `Socket` object or the data read from it).

Abrupt termination

Currently, we treat abrupt termination in concurrent programs half-heartedly. Exceptions can only be thrown but not caught. Abrupt method completion upon return is, in contrast, possible.

We have mostly elided treating method invocations so far. Here's how it works. When inlining method implementations KeY marks the method boundaries with so-called method frame blocks:

`method-frame(result->`*retvar*`, source=`*T*`, this=`*target*`) : {` *body* `}` ,

which also record the variable to store the return value upon return, the source class of the method body, and the value of `this` reference.

Upon encountering a return statement in the *body* of a method frame, the rule for handling returns assigns the returned value to *retvar* and then moves the involved thread id to the position immediately outside the method frame.

return

$$\frac{\begin{array}{r l} & \Longrightarrow \mathcal{P}=i \\ \neg path(i,p,tid) & \Longrightarrow \langle\!\langle[\alpha\ \texttt{method-frame(...)}\ : \\ & \qquad \{\beta\ ^{\{n-1\}}\texttt{return}\ se;^{\{k+1\}}\gamma\}\ ^{\{l\}}\omega]\!\rangle\phi \\ path(i,p,tid) & \Longrightarrow \langle\!\langle[\underline{retvar{=}se;\,^{*(tid)}}]\!\rangle\langle\!\langle[\alpha\ \texttt{method-frame(...)}\ : \\ & \qquad \{\beta\ ^{\{n-1\}}\texttt{return}\ se;^{\{k\}}\gamma\}\ ^{\{l+1\}}\omega]\!\rangle\phi \end{array}}{\Longrightarrow \langle\!\langle[\alpha\ \texttt{method-frame(...)}\ :\ \{\beta\ ^{\{n\}}\underset{\substack{\uparrow \\ \text{position } i \text{ in } p}}{\texttt{return}}\ se;^{\{k\}}\gamma\}\ ^{\{l\}}\omega]\!\rangle\phi}$$

Now, it would be desirable to extend the treatment of abrupt termination to full handling of exceptions. The sequential KeY calculus does it by syntactically rearranging program parts. In the concurrent case this approach is probably not feasible, as it would result in different program texts for different threads. A more promising approach is to equip each thread with a ghost variable "exception status", which would contain either the last thrown exception or $\perp$ (for normal execution). We also extend the definition of the path condition to include exception status. According to this flag threads will execute or skip certain parts of the program. If the exception status is $\perp$, the program executes normal code and skips any catch clauses. If an exception has been thrown, the program skips normal code and executes the (appropriate) catch clause. Executing finally clauses would require jumps similar to those after a return, but these are harmless, since it is impossible to create a loop.

Modularization

It is known that the efficiency of a verification system is bounded to a great degree by the compositionality of reasoning it offers. Suggestions for modularizing reasoning about concurrent JAVA programs have been made in [Greenhouse and Scherlis, 2002b; Rodríguez et al., 2005] and others. Research indicates consistently that programmers use a small number of "serializability techniques", such as locking protocols and reference confinement, to ensure correctness of programs.

With sequential programs, methods and their contracts are common units of composition. Of course, regular contracts are meaningless in concurrent setting due to potential interference from other threads. On the other hand, the researchers cited above have developed additional annotations and analyzes to mitigate this problem. For instance, if the objects mentioned in the pre- and postconditions are referenced only by the caller thread, then replacing the method call by its contract is sound again. The same holds if the state relevant for a method call is consistently protected by a lock and the calling thread holds this lock.

These techniques, originally developed for model checking and static analysis, can be put to efficient use in a deductive framework.

We have also experimented with incorporating the rely-guarantee approach into our verification framework [Schaaf, 2008]. In this setup, the given program is interleaved with an "opaque" environment, which produces updates to shared state that are described by the rely predicate. The experiments have shown that the rely-guarantee method is compatible with Dynamic Logic and our symbolic execution calculus. Writing transitive specifications and managing shared state turned out to be challenging though.

Verification of lock-free algorithms

An interesting class of algorithms are so-called lock-free data structures [Herlihy, 1993]. Their goal is to increase the level of concurrency—and thus throughput—in an application by not relying on critical sections and mutual exclusion.

Threads do not need to lock the data structure before reading or updating a lock-free data structure. A thread wishing to perform an update makes a copy of the data, modifies the copy, and tries to install the modified version using a special atomic compare-and-swap (or similar) instruction. This succeeds if no other thread has performed an update in the meantime. Otherwise, installation fails, the overtaken thread discards its modified copy and repeats the process from the beginning.

Lock-free concurrency is rapidly entering the mainstream, e.g., as part of the standard JAVA library. The problem with lock-free algorithms is that they are notoriously difficult to design and implement correctly. Verifying them would bring a significant benefit to the field. At this time, only first attempts are being made to produce mechanized proofs of real implementations.

8

Implementation and Case Studies

8.1 Implementation

We have implemented the basic calculus described in Chapter 6 and the JMM-safety extension (Section 7.3) in the KeY system. The changes w.r.t. a stock system amount to about 3200 lines of code in 56 files. The greatest technical difficulty by far was a generalization of the rule application engine. From the very beginning the KeY system was designed to apply program-manipulating rules only at the beginning of a program. This limitation had to be lifted in order to support multi-threaded execution.

Specification

Verification problems are specified in Dynamic Logic and input to the prover as so-called dot-key files [Beckert et al., 2007]. We have extended the syntax of dot-key files with a keyword `\local`. The keyword distinguishes thread-local from static variables in declarations.

```
\programVariables {
    \local int loc; // thread-local
    int glob;       // static
}
```

KeY

Thread configurations are specified with updates to the non-rigid function `pos`. A typical formula thus looks like:

```
\problem {
    {  \for int i; pos(i) := 0 || pos(1):=2  }
    \<{
        glob = loc;
        ;
    }\> glob = loc_l(p(1,2))
}
```

——— KeY ———

The update (first line) states that pos is always zero except at position one, where two threads are ready to be scheduled. The extra semicolon in the diamond is concrete syntax for the `stop` statement. `p(1,2)` is concrete syntax for $\pi_1(2)$, and `loc_l` is the prover's way to refer to the local variable `loc` outside the modalities. As promised, `loc_l` has one argument more than `loc` (i.e., the thread id).

The calculus rules

The step rule is implemented slightly differently from the formulation shown in Section 6.3. There is no premiss $\Longrightarrow \mathcal{P} = k$. Instead the implementation follows the pattern of the invariant rule (⇒ Sect. 6.5) and automatically performs a case distinction over all positions. The rule is shown in Figure 8.1. Per position at least two subgoals are generated: one for the positive and one for the negative path condition. In the positive case, a rule from the sequential calculus is matched to the position. The rule describes the effect on the state resulting from executing this position. This effect may include generating an update or producing further case distinctions, e.g., to check for a null reference. An additional subgoal is added for the case that no position is enabled.

Automation

Proof search is automated by the usual strategies of the KeY prover. We have extended the main strategy with a further parameter controlling when the step rule is to be applied automatically:

- never
- until some thread becomes disabled
- without limitation.

The second setting is especially useful when performing induction proofs. In all cases, step is executed with very low priority, i.e., only after no other rules are applicable and the state description has been simplified as far as possible.

We have also implemented a separate *unfolding* strategy that pools all rules for program unfolding (⇒ Sect. 6.2). This strategy is only used for preparing proof obligations and is not active during proof search.

8.2 Full functional correctness of java.lang.StringBuffer

We have applied our system to verify the full functional correctness of a method of the `StringBuffer` class in presence of unbounded concurrency. The class `java.lang.StringBuffer` is a key class of the standard Java library that represents a mutable character sequence. Its central method is `append(char c)`, which appends the character `c` to the end of the sequence.

step (impl.)

$$\frac{\begin{array}{l}\forall i.\,(1 \le i \le q \longrightarrow \neg enabled(i)) \Longrightarrow \phi \\[1em] path(1,p,tid(1)),\ enabled(1) \Longrightarrow \\ \quad \langle\![\underline{p_1^{*(tid(1))}}]\!\rangle\{pos(1) := pos(1) - 1\}\{pos(2) := pos(2) + 1\}\langle\![p]\!\rangle\phi \\[1em] \neg path(1,p,tid(1)),\ enabled(1) \Longrightarrow \\ \quad \{pos(1) := pos(1) - 1\}\{pos(2) := pos(2) + 1\}\langle\![p]\!\rangle\phi \\ \qquad \vdots \\ path(q,p,tid(q)),\ enabled(q) \Longrightarrow \\ \quad \langle\![\underline{p_q^{*(tid(q))}}]\!\rangle\{pos(q) := pos(q) - 1\}\{pos(q+1) := pos(q+1) + 1\}\langle\![p]\!\rangle\phi \\[1em] \neg path(q,p,tid(q)),\ enabled(q) \Longrightarrow \\ \quad \{pos(q) := pos(q) - 1\}\{pos(q+1) := pos(q+1) + 1\}\langle\![p]\!\rangle\phi \end{array}}{\Longrightarrow \langle\![p]\!\rangle\phi}$$

Figure 8.1. Implementation of the step rule

```
private char value[];
private int count;

public synchronized StringBuffer append(char c) {
    int newcount = count + 1;
    if (newcount > value.length)
        expandCapacity(newcount);
    value[count++] = c;
    return this;
}

private void expandCapacity(int minimumCapacity) {
    int newCapacity = (value.length + 1) * 2;
    if (newCapacity < 0) {
        newCapacity = Integer.MAX_VALUE;
    } else if (minimumCapacity > newCapacity) {
        newCapacity = minimumCapacity;
    }

    char newValue[] = new char[newCapacity];
    System.arraycopy(value, 0, newValue, 0, count);
    value = newValue;
    shared = false;
}
```

Figure 8.2. `StringBuffer` source code (excerpt)

We have used the original source code shipped by SUN with the JDK 1.4.2 (shown in Figure 8.2). The `StringBuffer` implementation is backed by a `char` array, which is initially 16 elements long. Should the array become full, a new, longer array is allocated and the contents copied. This happens transparently for the user.

We now describe the verification process.

Specification

A functional specification of the append method can be given as:

$$\langle \underline{\texttt{strb = new StringBuffer();}} \rangle \forall n.$$
$$\Big(n > 0 \longrightarrow \langle^{\{n\}} \texttt{strb.append(c);}^{\{0\}} \rangle \texttt{strb.count} = n \land \forall k. \big(0 \le k < n \longrightarrow \texttt{strb.value}[k] = \texttt{c}(\pi_1(k+1))\big)\Big) \quad , \qquad (8.1)$$

where `strb` is a static variable of type `StringBuffer`[1] and `c` is a thread-local `char` variable.

Plainly speaking: if *n* threads are concurrently performing an append on a freshly created shared `StringBuffer` object, then all threads will eventually run to completion, and the StringBuffer will contain exactly the characters deposited by the threads. Furthermore, the characters will fill the backing array in the "natural" order, i.e., the order induced by the thread scheduling.

After symbolic execution of the `StringBuffer` creation (in the sequential diamond) and Skolemization, the original conjecture becomes:

$$\mathit{Init} \land n_0 > 0 \longrightarrow \langle^{\{n_0\}} \texttt{strb.append(c);}^{\{0\}} \rangle \texttt{strb.count} = n_0 \land \forall k. \big(0 \le k < n_0 \longrightarrow \texttt{strb.value}[k] = \texttt{c}(\pi_1(k+1))\big) \quad , \qquad (8.2)$$

where n_0 is a fresh integer constant and *Init* is a formula capturing the state after `StringBuffer` creation. *Init* is shorthand for:

$$\texttt{strb} \neq \texttt{null} \land \texttt{strb.<lockcount>} = 0 \land \texttt{strb.count} = 0 \land$$
$$\texttt{strb.value} \neq \texttt{null} \land \texttt{strb.value.length} = 16 \land$$
$$\texttt{strb.value} \neq \texttt{jchar[]::<get>(jchar[].<nextToCreate>)} \ .$$

The cryptic last subformula states that the current `value` array is not aliased to the next `char` array to be created. While this precondition is completely obvious, it is owed to the way KeY deals with instance creation.

[1] The semantics definition calls for a local variable here, but the calculus is more liberal in this regard. We use this liberty to write a simpler proof obligation while still achieving the same effect.

Unfolding

To proceed with verification, we first "unfold" (⇒ Sect. 6.2) the implementation of `append()`. The `expandCapacity()` method is inlined, and fresh local variables are introduced to eliminate side effects and make explicit the atomicity granularity of the code. The result is shown in Figure 8.3, though exceptions and array creation are still in their folded state for brevity.

The code also shows a call to `System.arraycopy()`, which cannot be unfolded. This native method call can be seen as one big parallel assignment, which is sound under the atomicity proviso proven below. During symbolic execution, the KeY system translates a call like `arraycopy(`*src*`,`*srcPos*`,`*dest*`,`*destPos*`,`*len*`)` into a quantified update (⇒ Sect. 2.4.2)

$$\{\texttt{for}\ l;\ 0 \leq l < len;\ dest[srcDest + l] := src[srcPos + l]\}\ ,$$

which is a concise way to express a number of updates at once.

```
strb.<lock>();
newcount=strb.count+1;
j_1=strb.value.length;
b=newcount>j_1;
if (b) {
    j_2=strb.value.length;
    j_3=j_2+1;
    newCapacity=j_3*2;
    b_1=newCapacity_<0;
    if (b_1) {
        newCapacity=Integer.MAX_VALUE;
    } else {
        b_2=newcount>newCapacity;
        if (b_2) {
            newCapacity=newcount;
        }
    }
    b_3=newCapacity<0;
    if (b_3) throw new NegativeArraySizeException();
    newObject=new char[newCapacity];
    src_1=strb.value;
    len_2=strb.count;
    System.arraycopy(src_1,0,newObject,0,len_2);
    strb.value=newObject;
}
val_1=strb.value;
j_4=strb.count;
strb.count=j_4+1;
val_1[j_4]=c;
strb.<unlock>();
```

Figure 8.3. `StringBuffer` source code after unfolding

Establishing Atomicity

To separate concerns, we now use the invariant rule to establish atomicity of the method. This greatly simplifies further proof. We follow the pattern from Section 7.1

and show that the method can only be executed by one thread at a time (on the same object). This property can be stated as

$$N \leq 1, \text{ with } N = \sum_{i=2}^{q} pos(i) \enspace ,$$

so the configuration never has more than one thread between its second and the last but one position. Before the proof can proceed, the above has to be strengthened to

$$INV = N \leq 1 \wedge \big(N > 0 \longleftrightarrow \texttt{strb.<lockcount>>0}\big) \enspace .$$

This invariant clearly holds in the initial state, since both N and `<lockcount>` are zero. Statements at positions $2 \ldots q$ preserve the invariant, since they cannot increase the value of N, as only the statement at position 1 can. Finally, the locking statement at position 1 also preserves the invariant. If the lock is available, then $N = 0$ before the locking per the second conjunct of the invariant. After the execution, both N and `<lockcount>` are equal to 1. If the lock is not available, then the locking statement is disabled altogether.

Per this invariant, once a thread has entered the method it will run to completion without interference. Thus, the method is atomic, and we can elide locking, replacing it by an atomic block. Our conjecture becomes:

$$\begin{aligned} Init \wedge n_0 > 0 \longrightarrow \langle^{\{n_0\}} \ll \texttt{strb.append1(c);} \gg^{\{0\}} \rangle \texttt{strb.count} = n_0 \wedge \\ \forall k. \big(0 \leq k < n_0 \longrightarrow \texttt{strb.value[}k\texttt{]} = \texttt{c}(\pi_1(k+1))\big) \enspace , \end{aligned} \tag{8.3}$$

where the method `append1(c)` (shown here folded) is identical to `append(c)` save for the removed locking operations.

Establishing Functional Correctness

So far, we know that the method is correctly synchronized, but is it also functionally correct? Using the Java-faithful bounded integer semantics of KeY, we have, of course, discovered that the specification shown above is not quite right, as it holds true only for $n_0 < 2^{31}$. Trying to insert more characters into a `StringBuffer` results in an `ArrayIndexOutOfBoundsException`. This bound may seem of little practical importance, but it is an instance of a general problem. Concurrent access to bounded data structures is likely to result in subtle bugs, even in presence of proper synchronization.

Since there is no way to fix the method, we have to amend the conjecture with a pre-condition limiting the value of n_0. Please note that this is not due to a limitation of our proof method. We now prove full functional correctness with the following, quite natural invariant:[2]

[2] It is also possible to use induction in a manner similar to Example 6.1.

$$\begin{aligned}
INV = {} & pos(1) + pos(2) = n_0 \land n_0 < 2^{31} \land pos(2) \geq 0 \land \\
& \texttt{strb.count} = pos(2) \land \\
& \forall k.\, (0 \leq k \land k < pos(2) \longrightarrow \texttt{strb.value}[k] = \texttt{c}(\pi_1(k+1))) \land \\
& \texttt{strb} \neq \texttt{null} \land \texttt{strb.value} \neq \texttt{null} \land \\
& \texttt{strb.value.length} \geq \texttt{strb.count} \land \\
& \texttt{strb.value} \neq \texttt{jchar[]::<get>(jchar[].<nextToCreate>} \ .
\end{aligned}$$

Applying the invariant rule to (8.3) produces three premisses.

Premiss 1: invariant initially valid

In this premiss we need to prove the sequent $\Gamma \Longrightarrow \mathcal{U} INV, \Delta$. Here, Γ contains just $Init \land (n_0 > 0) \land (n_0 < 2^{31})$, and Δ is empty. The update $\mathcal{U}$ is given by the thread configuration of the original program. The formula

$$\langle^{\{n_0\}} \ll \texttt{strb.append1(c);} \gg^{\{0\}} \rangle \phi$$

is shorthand for

$$\{pos(1) := n_0 \,\|\, pos(2) := 0\} \, \langle \ll \texttt{strb.append1(c);} \gg \rangle \phi \ .$$

The proof obligation is thus:

$$\begin{aligned}
& Init \land n_0 > 0 \land n_0 < 2^{31} \Longrightarrow \\
& \quad \{pos(1) := n_0 \,\|\, pos(2) := 0\} \Big(pos(1) + pos(2) = n_0 \land n_0 < 2^{31} \land pos(2) \geq 0 \land \\
& \quad \texttt{strb.count} = pos(2) \land \\
& \quad \forall k.\, (0 \leq k \land k < pos(2) \longrightarrow \texttt{strb.value}[k] = \texttt{c}(\pi_1(k+1))) \land \\
& \quad \texttt{strb} \neq \texttt{null} \land \texttt{strb.value} \neq \texttt{null} \land \\
& \quad \texttt{strb.value.length} \geq \texttt{strb.count} \land \\
& \quad \texttt{strb.value} \neq \texttt{jchar[]::<get>(jchar[].<nextToCreate>} \Big) \ .
\end{aligned}$$

The quantifier in the succedent has an empty range (due to the update $pos(2) := 0$), and further basic rewriting renders the sequent proved. The calculus implementation finds the proof automatically in 67 steps.

Premiss 2: invariant implies postcondition upon termination

In this premiss we need to prove the sequent $INV, \mathcal{P} = 0 \Longrightarrow \phi$, where ϕ is the postcondition. Since the atomic block is the only position, $\mathcal{P} = 0$ is equivalent to $pos(1) = 0$ (per Axiom (5.4)). The proof obligation is thus:

$$\begin{aligned}
& pos(1) + pos(2) = n_0 \land n_0 < 2^{31} \land pos(2) \geq 0 \land \\
& \texttt{strb.count} = pos(2) \land \\
& \forall k.\, (0 \leq k \land k < pos(2) \longrightarrow \texttt{strb.value}[k] = \texttt{c}(\pi_1(k+1))) \land \ldots, \\
& pos(1) = 0 \Longrightarrow \\
& \quad \texttt{strb.count} = n_0 \land \\
& \quad \forall k.\, \big(0 \leq k < n_0 \longrightarrow \texttt{strb.value}[k] = \texttt{c}(\pi_1(k+1))\big) \ .
\end{aligned}$$

This sequent is easily discharged, since $pos(1) + pos(2) = n_0$ together with $pos(1) = 0$ implies $pos(2) = n_0$. The calculus implementation finds the proof automatically in 108 steps.

Premiss 3: invariant preservation

In this premiss we need to prove

$$INV,\ path(1, p, tid(1)),\ enabled(1) \Longrightarrow \langle\![\underline{p_1^{*(tid(1))}}]\!\rangle \{pos(1) := pos(1) - 1\}\{pos(2) := pos(2) + 1\} INV\ ,$$

which is a purely sequential proof obligation. After expanding the definitions, the path condition simplifies to *true* and the predicate *enabled*(1) to $pos(1) > 0$. We also expand the definition of sequential program instantiation, obtaining the goal

$$INV,\ pos(1) > 0 \Longrightarrow \langle\![\underline{\ll \texttt{strb.append1(c(}\pi_1(pos(2)+1)\texttt{));} \gg}]\!\rangle \{pos(1) := pos(1) - 1\}\{pos(2) := pos(2) + 1\} INV\ .$$

This goal is the most difficult to prove, since it requires symbolic execution of the method, reasoning about Java-faithful arithmetics, and quantifier instantiation. The calculus implementation finds the 2898-step long proof automatically in about 30 seconds.

Further Issues with java.lang.StringBuffer

```
public synchronized StringBuffer append(StringBuffer sb) {
    if (sb == null) {
        sb = NULL;
    }

    int len = sb.length();              // 1
    int newcount = count + len;
    if (newcount > value.length)
        expandCapacity(newcount);
    sb.getChars(0, len, value, count); // 2
    count = newcount;
    return this;
}
```

Figure 8.4. Atomicity failure in `StringBuffer`

An interesting issue is present in a related method of `StringBuffer` class: the method `append(StringBuffer sb)` shown in Figure 8.4.

The method has two critical points: when the length of `sb` is queried (at 1) and when the characters are actually copied (at 2). The problem with this code is that nothing prevents some other thread to be scheduled between the execution of (1) and (2). The intruding thread may end up removing characters from `sb`; the length read at (1) becomes stale and an attempt to copy no-longer-existing characters at (2) produces an exception.

Note that this scenario does not constitute a data race. All methods involved (i.e., `length()`, `getChars(...)`, and `delete(...)`) are synchronized, thus, all access to shared data of `sb` is protected by locks. It's rather that the lock is released and then re-acquired, violating the application-specific atomicity policy. One can speculate that this was done for performance reasons.

The question of course arises how this issue can be detected by verification. It is clear that the approach taken in verification of the `append(char c)` method is not sufficient. Our proof system operates under the closed world assumption, and it would be necessary to verify the execution of `append(StringBuffer sb)` in parallel with `delete(...)` and possibly other methods. If the full program is not available, it is possible to interleave the execution of the method at hand with an abstract environment program and find a set of assumptions about this environment that is still sufficient for the correctness of the method. This kind of rely-guarantee reasoning is addressed briefly in Section 7.4.

Conclusion

During the development of the presented proof system we have learned several lessons.

It is possible to execute multi-threaded programs symbolically while taking full data into account. By employing an explicit scheduler function, our calculus can track full information about state quite efficiently, but permits abstraction for further improvement.

Underspecification is better than non-determinism. The huge range of scheduler choices can be adequately modeled by a deterministic function that has a fixed but unknown value. This formalization enables efficient deduction. Relating different runs of the scheduler can be achieved by incorporating different "don't-knows".

Describing a domain algebraically is better than giving an enumeration. Descriptions allow us to postpone reasoning until a maximum of information is available. At this point, some distinctions may have become irrelevant. It might also be possible to apply powerful simplification laws. If everything else fails, it is still not too late to produce an explicit enumeration.

Completeness is a desirable property, but a proof system need not be always efficient. It is enough if the system is efficient on benign cases. Modern programming languages may offer obscure features and means to write very complicated programs, but programmers' ability to use them correctly still remains limited. Failing to find a correctness proof with a sufficiently developed proof system is often a sign that something is wrong with the program to be verified.

Atomicity failure is a better notion than data race. The absence of data races is, in general, neither sufficient nor necessary for correct multi-threaded programs.

Furthermore, the correct level of atomicity for an application is always domain-dependent. Data race definitions do not take this into account.

Currently, deductive verification can offer advantages for verification problems that are data-centric or that involve an unbounded number of threads. At the same time, a convergence of deductive verification, static analysis and model checking can be noted. Latest incarnations of static verifiers and model checking frameworks successfully incorporate symbolic execution techniques and make use of theorem provers and theory solvers. On the other hand, deductive verifiers are adapting static analysis techniques and optimizations originally developed for model checking.

Part III

Must Program Verification Systems and Calculi Be Verified?

Typographic Conventions

The following symbols have a different meaning in the following Part:

$\mathcal{T}$	denotes in the following a so-called domain theory. A *theory* is a set of formulas of the underlying logic. These formulas are called *axioms*.

Ensuring Reliability of Reasoning in Program Verification

Mechanized deductive reasoning involves many ingredients. Among these are a deduction calculus, a reasoner implementation, domain-specific theories, and user input. All of these ingredients can contribute to a reasoning failure. The problem is especially acute when reasoning over large domain theories, such as formal programming language semantics. We analyze how different methods combat different modes of failure. In particular, we raise the question of self-application of reasoning-based formal methods as a means to ensure reliability. We summarize the situation in the KeY project and give recommendations based on our experience in design of verification systems.

9.1 Introduction

Deduction is reasoning about models or abstractions of—sometimes purely mathematical but mostly real and practical—aspects of the world. To use mechanized deduction, we have to do two things. First, we have to formalize a given real-world problem, the domain knowledge, and the domain laws in a suitable logic. On the formal level, this gives us a query ϕ, which is a formula of the logic, and a domain theory $\mathcal{T}$, which is a set of formulas (axioms). Second, we ask the question whether the theory logically entails the query ("$\models_{\mathcal{T}} \phi$?" or "$\mathcal{T} \models \phi$?"). The answer to this question is usually computed by means of a calculus implemented in a computer system ("$\mathcal{T} \vdash \phi$?").

The two-step process outlined above implies that the issue of reliability of reasoning is actually twofold. On the one hand, the reasoning must be formally *correct*, i.e., $\vdash$ must correctly implement $\models$. On the other hand, the reasoning must be *adequate*, i.e., $\mathcal{T}$ and ϕ as well as $\models$ must represent reality in the intended way.

The latter part of the issue—adequacy—is the controversial one. It is possible, and indeed has been long customary, to consider the theory $\mathcal{T}$ as being part of the input to the reasoning system in the same way as the query ϕ, and as opposed to being part of the system in the same way as $\models$ resp. $\vdash$. In other words, one asks the question "$\models (\mathcal{T} \rightarrow \phi)$?" instead of "$\models_{\mathcal{T}} \phi$?", thus making the adequacy of $\mathcal{T}$ a "somebody else's problem". We argue that drawing the system boundaries in such a way is not a good solution as it's simply offloading the problem into the area of *usability*.

The reasoning community must deal with both correctness and adequacy in order to be successful in practice. As mechanized deduction gains power to tackle more complex problems, the inadequacy of large domain theories outgrows incorrectness as the primary source of undesired results. This finding is corroborated by our experience in building software verification systems, but the problem also persists in other domains.

There is already a huge body of work on the design of both correct calculi and adequate theories. Mostly, however, a particular calculus or a particular domain is investigated. We want to take a more general point of view. We investigate how different validation methods—both formal and conventional, from verification to testing—are best used to ensure seamless reliability of the reasoning process, how different methods relate to each other, and which methods are best suited to uncover which kind of faults.

Discussions and some reactions we got to talks on this subject show that the question of how best to ensure reliability of deduction systems is often fraught with ideological arguments, such as "If you build a program verification system, you have a *moral* obligation to formally verify it." With this work, we neither try to reject such arguments as wrong nor as irrelevant, but intend to put the discussion on more solid ground. If reasoning technology is to be used in practice, the developers have to be able to explain to users (and certification agencies) why their reasoning process is reliable in its entirety.[1] And they have to know how and where resources are best spent to improve the systems.

The structure of this chapter is as follows. In Section 9.2, we clarify some notions related to the reliability of reasoning. Such a clarification is important as a clear understanding of the differences between notions such as reliability and correctness or fault and failure is important for the following discussions (and is often ignored in the reasoning community). Then, in Section 9.3, we describe the particular problems of ensuring reliability of large domain theories. In Section 9.4, we define and discuss different methods and their efficacy for validating reasoning systems, calculi, and theories. In Section 9.5, we report how we ensure reliability of our own verification system and discuss the role of competitions. Finally, in Section 9.6, we summarize our recommendations for ensuring reliability of the reasoning process.

9.2 Clarification of Concepts and Notions

9.2.1 Dependability and Reliability

Dependability of a computing system is the ability to deliver service that can justifiably be trusted [Avižienis et al., 2000]. While dependability of deductive reasoning systems is important, the exact requirements are substantially different from other

[1] Very interesting research in this direction is carried out by Denney and Fischer [2005].

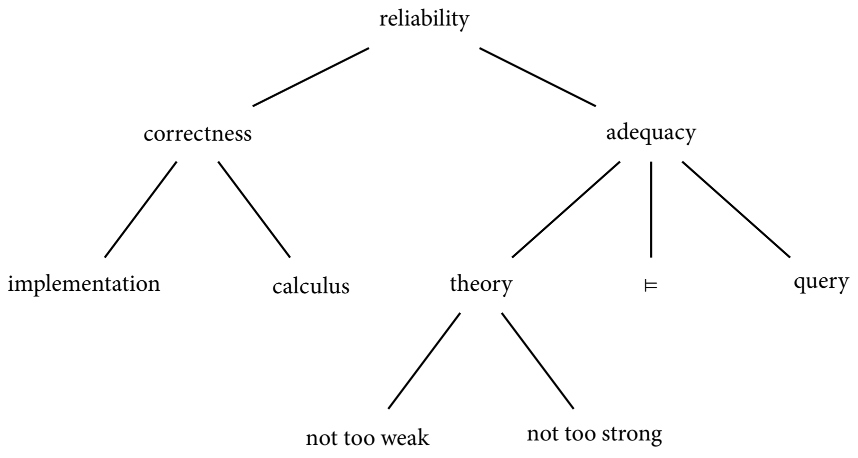


Figure 9.1. A reasoning reliability ontology

critical software. What users demand of dependable reasoning systems is in the first place *reliability*, i.e., a high probability that the system will perform requests as desired. Other aspects of dependability, such as confidentiality or safety, are not considered essential for reasoning systems.

9.2.2 Faults and Failures

Dependability research distinguishes between *faults*, *errors*, and *failures*. In the classification of [Avižienis et al., 2000], an error is a part of the system state that can cause a failure. A failure occurs when an error reaches the service interface of a system and alters the service. A fault is an adjudged or hypothesized cause of an error. A fault is active when it produces an error and dormant otherwise. Fault activation is the application of an input (the activation pattern) to the system that causes a dormant fault to become active.

As stated in the introduction, reasoning systems can suffer from two kinds of failures. A correctness failure is when the system reports the wrong answer to the question "$\mathcal{T} \models \phi$?" Correctness failures are caused by faults in the calculus or its implementation. An adequacy failure is when the system reports the correct answer to "$\mathcal{T} \models \phi$?", but a problem with $\mathcal{T}$, ϕ, or $\models$ makes this answer unfaithful to the real-world domain.

In the same way as only one aspect of dependability, namely reliability, is really essential for reasoning systems, only one kind of failure is really critical, namely *unsignaled* failure. While a crash of avionics software is unacceptable, a crash of a theorem prover may be annoying but is in general not harmful. In this vein, any failure that is *evidently* out of the norm is tolerable in a reasoner.

$$\begin{tikzcd}\text{fault} \arrow[r, "\text{activation}"] & \text{error} \arrow[r, "\text{propagation}"] & \text{failure}\end{tikzcd}$$

Figure 9.2. Faults, errors and failures

9.2.3 Ingredients of the Reasoning Process and their Faults

As mentioned above, the following main ingredients play a role in what answer is produced by a reasoner:

Calculus. The deduction calculus can have faults, in particular if it employs complex techniques for reducing the search space (e.g., lemma generation) or if the logic is non-standard (e.g., several different kinds of modalities).

Implementation. The implemented system that executes the calculus can have faults not only in the way it applies rules but also in functionality on which rule application relies, such as Skolemization, term indexing, or unification. There can also be faults in the proof search procedure that lead to incompleteness.

Domain-specific theories. Large domain-specific theories (examples of which range from a formal specification of programming language semantics to a domain ontology[2]) are the biggest source of failures in mechanized deductive reasoning. They may be inadequate (i.e., not reflecting reality) in two ways: being too strong or too weak. A theory is *too strong* when it excludes desired models and *too weak* when it allows undesired models. Note that one part of a theory can be too strong while another part is too weak at the same time. We explore these notions in more depth in Section 9.3.

Sometimes a theory used by a reasoner is "unnecessarily" large. For instance, it may be possible to axiomatize some mathematical domain by a small set of axioms, but a reasoner uses a larger though equivalent set of axioms for efficiency reasons. In this case, the adequacy problem turns mostly into a correctness problem. Even though the correctness problem still needs to be handled, this is a better situation than with many theories that are not derivable from small axiom sets (e.g., ontologies).

User input. The input to the system is the query posed by the user (e.g., a program piece to be verified and its specification, or a query submitted to a semantic web reasoner). A query submitted by the user to the system can be inadequate (i.e., it does not mean what the user thinks it means) or even vacuous (trivially satisfiable

[2] A theory of a programming language may formalize its semantics in a number of different forms. Examples are: a structural operational semantics, a program logic, a verification calculus, etc. We have stated three such theories for a concurrent Java-like language in this thesis: in Chapters 4, 5, and 6 respectively. Obviously, our main interest is directed towards theories used in verification, but our remarks are not limited to this area. We will also brush the topic of ontologies, which are among the largest domain theories used for mechanized reasoning today.

in an unintended way[3]). Faults in the input have been neglected in deduction research for a long time. The study [Beer et al., 2001], however, indicates that up to 20% of properties submitted to a model checker in practice are vacuous and that such "trivial validity always points to a real problem in either the design or its specification or environment". Furthermore, the study reports adequacy problems with up to 10% of non-vacuous properties.

Note that the distinction of ingredients is conceptual. Concrete systems may incorporate these ingredients in different guises. The same domain theory can be encoded in form of logical formulas in one system ($\mathcal{T} \vdash \phi$), be part of the basic calculus in another ($\vdash_{\mathcal{T}} \phi$), or be hard-coded in the implementation of a decision procedure in a third. This polymorphism does not invalidate our remarks.

9.3 The Problem of Theory Adequacy

9.3.1 The Lack of Formal Semantics

If the domain that a theory formalizes is itself formally defined, it is possible to formally prove its adequacy—even if that may be a difficult and tedious task.

For mathematical domains, a formal reference definition is usually available, while that is hardly possible for theories that formalize an aspect of the real world (such as ontologies).

Domains from computer science applications usually lie in between. Most of these domains involve formal languages, but in practice a formal definition is often not readily available. For example, hardly any programming language in wide use today has an official formal semantics. For instance, there is no official formal semantics of the Java programming language. Sun Microsystems, the holder of the Java trademark, decides what constitutes a valid Java implementation within the framework of the Java Community Process. It is required that every such implementation adheres to the Java Language Specification, which is a precise but informal document. Conformance is, in particular, checked by a compatibility test suite.

Many research groups have come up with their own formal semantics of (fragments of) the Java language.[4] Ultimately, there is no formal way to judge whether any of these semantics is adequate, i.e., reflects the official informal specification correctly. Verifying one theory of Java against another is helpful, but some doubt will always remain about whether both theories are correct w.r.t. the official language specification and its implementations (compilers, virtual machines), which is what counts in practice.

Consequently, other methods such as testing the theory using a large number of programs (e.g., a compiler test suite like [Jacks]) can lead to the same—or even

[3] For instance, the temporal assertion that every request is followed by a response is vacuously true in a model with no requests.

[4] Beckert et al. [2007]; Jacobs and Poll [2001c]; Poetzsch-Heffter and Müller [1999b]; von Oheimb [2001a]; Zee et al. [2008]; Marché et al. [2004], and many others.

a higher—degree of reliability w.r.t. the informal language specification as a formal proof.

9.3.2 The Problem of Too Strong Theory

Theories may be too strong and thus exclude desired models. From the logical perspective, the interesting case of a too strong theory is an inconsistent theory. While it is perfectly sound to derive any formula from an inconsistent theory, this is clearly not what the users of deduction systems want. Deduction must not leave this problem out of consideration. There is a number of reasons for inconsistencies in domain theories:

- One reason is misunderstandings and clerical mistakes. When detected, these can be easily fixed.
- Another reason is a problem in the domain itself. For example, an important part of the Java Language Specification is the Java Memory Model. Lately a semi-formal definition of the model was adopted by Sun [Manson et al., 2005b]. Recent research [Aspinall and Ševčík, 2007; Huisman and Petri, 2007], however, has shown that the proposed model is inconsistent, and there is also no obvious "fix" to the problem. Domain experts agree though that there is a fragment of the model that is safe for programming and reasoning.
- Yet another reason lies in the size of the theory and its authoring process. In the domain of semantic web, very large ontologies are routinely produced by combining several smaller ones. Rigorously ensuring the consistency of the result is—in this domain—often impractical. It is to be expected that large ontologies will contain inconsistencies [Huang et al., 2005].

It remains an open research question, how to build deduction systems so that inconsistencies are detected, and if not—the probability remains small that a wrong (inadequate) answer is derived.

9.3.3 The Problem of Too Weak Theory

Theories may be too weak and thus admit models undesirable in practice. We distinguish two basic sources of theory weakness: missing features and the chosen level of abstraction.

It is likely that a large domain-specific theory does not cover some features of the domain. This incompleteness manifests itself as failure to verify correct programs or inability to answer queries over particular vocabulary. When solving this deficiency by incorporating an additional ontology or extending the language semantics it is often impossible or impractical to guarantee consistency of the result. Still, practitioners might prefer a verification system that is 99% correct but covers all of the programming language to a 100% correct system that covers only 75% of the target language.

Furthermore, theories only capture reality up to a certain level of abstraction. Practitioners know that verified software may still fail. This can happen because the software is part of a larger system that fails (compiler, operating system, hardware);

or because the verification assurance does not cover an important aspect, such as security, quality of service, or, in particular, fault tolerance of the whole system. Thus, formally verified correctness never leads to absolute reliability. Nothing is 100% reliable in the sense that it does never fail.

9.4 Different Ways to Reliability of Reasoning

9.4.1 Conventional Ways to Reliable Software

Practitioners know that formal methods are not the only way to reach a high level of dependability and, in particular, reliability. High dependability of software used in practice can be achieved with testing and experience from long-term use as well.

The aviation industry, which has a high level of reliability in all its systems, is a good example. A very important measure used to achieve this reliability is the careful investigation and analysis of accidents (failures) and immediate feedback to design and operation. The use of well-matured technology also contributes to keeping the reliability level [Sakugawa et al., 2005]. Aviation industry also has universal regulations for the use of software in airborne systems. One part of these regulations is the guideline DO-178B [RTCA, 1992]. It lists objectives (for different levels of criticality) that a piece of software must satisfy in order to be certified for airborne use. With the increasing level of criticality, the total number of objectives increases, as well as the number of objectives that have to be satisfied "with independence", i.e., the validation activity has to be performed by a person other than the original developer. The main activity used to validate avionics software is rigorous testing. Reasoning-based formal methods are permitted but neither required nor sufficient by themselves. In general, DO-178B states that "formal methods are complementary to testing".

Complementarity is good for yet another reason. While dependability is about justifiable trust, trust is still a social process. Thus, introducing a technology (such as reasoning-based methods) cannot be done abruptly but requires a step-wise process. The new technology has to be evaluated in practice, even if it has been formally proven correct. When it is introduced, it has to be compared to and supported by well-known and trusted techniques (such as testing). This is the only way to ensure adequacy.

9.4.2 Measures Against Faults in Reasoning Systems

The reasoning community often favors a self-application of reasoning-based formal methods to ensure reliability of its tools, but conventional methods like testing and using mature technology are also useful. In reality, both kinds of methods are complementary and a balanced mix is necessary to achieve reliable systems.

Below, we survey different means to validate the ingredients of the reasoning process.

Measures against faults in the calculus

Formally verifying the (core) calculus, i.e., proving that $\vdash = \vDash$ or at least $\vdash \subset \vDash$, is an efficient method for removing faults. This assumes that the logic used for deduction has a well-established formal semantics (i.e., a definition of $\vDash$), against which verification can be done. This is often the case. If the core calculus is relatively small, its correctness proof can even be performed with paper and pencil (e.g., [Beckert and Platzer, 2006b]).

Of course, a calculus can also be tested. The answers it gives for test queries are compared with answers known to be correct and/or adequate. Usually, this is done as part of testing the calculus implementation (see below).

Measures against faults in the implementation

Verifying the reasoner implementation (with a program verification tool) is a possibility even though rarely practical due to the large size of the reasoner. The size problem can be alleviated by employing and verifying a proof checker—a small program that only has to check proofs and not find them. It is a common misconception, though, that a verified reasoner or proof checker makes reasoning completely reliable. These methods can avoid resp. detect faults of the implementations, but they do not mitigate adequacy problems in the domain theories.

Testing and service history also can assert reliability of the implementation. For example, people do believe in the correctness of Isabelle [Nipkow et al., 2002; Isabelle], even though the implementation is not verified (neither is the implementation of ML verified, etc.). That shows that at some point, even formal methods people stop verifying things that are well tested. Testing, here, means submitting queries ϕ to the implementation and comparing the answer to the definition of $\vdash \phi$ (correctness of the implementation w.r.t. the calculus), to the definition of $\vDash \phi$ (correctness of the calculus), and/or the expected answer (adequacy).

A good source of test cases for provers for non-program logics are the various benchmark suites, such as:

- the TPTP library [Sutcliffe and Suttner, 1998] (for first-order logic)
- the SATLIB library [Hoos and Stützle, 2000; Hoos and Stützle] and the problems used in the International SAT Competition [Le Berre and Simon] (for propositional logic)
- the ILTP library [Raths et al., 2007; Raths et al.] (for intuitionistic logic)
- the QBFLIB library [Giunchiglia et al.] (for quantified boolean formulas)
- the SMT-LIB library [Ranise and Tinelli, 2006; Barrett et al.] (for satisfiability-modulo-theories problems).

The suites can be used for evaluating both the *performance* and the *reliability* of systems.

Measures against faults in the domain theories

It is possible to verify a domain theory $\mathcal{T}$ formally against *another formalization* $\mathcal{T}'$ of the same domain. In program verification, for example, academia has put a lot of effort into "soundness proofs" of verification calculi.[5],[6] Such "soundness proofs" are in reality adequacy checks for theories of programming languages. They are well-suited for finding most kinds of faults, however complex or obscure. In domains, for which an *official* formal semantics $\mathcal{T}'$ is available, it is even possible to *guarantee* adequacy.

In other domains (and these are more common), a proof that $\mathcal{T}$ and $\mathcal{T}'$ are equivalent or that $\mathcal{T}$ is a refinement of $\mathcal{T}'$ is still useful, but it may fail to uncover adequacy faults. The chances that this happens increase if both formalizations $\mathcal{T}$ and $\mathcal{T}'$ as well as the equivalence proof are made by the same person. This renders the fault detection process less effective due to the increased probability that the author has misunderstood the domain, and that *both* theories are inadequate *in the same way*. Therefore, if theories are verified, they should be *cross-verified* against other people's formalizations of the domain laws. With cross-verification, the probability of uncovering faults is much higher.[7]

An underestimated way to ensure reliability of theories is testing. Testing means evaluating the relation $\mathcal{T} \vdash \phi$ for a number of queries ϕ and comparing the results with known adequate answers. Testing is good for uncovering misunderstandings.[8] It is much easier to detect faults with a test suite written by other people than to cross-verify a theory. For many domains, test suites are readily available as their creation does not require formalizing the domain laws. Note, however, that it is important to use both derivable and non-derivable queries for testing.

Testing may, of course, fail to find faults with complex or rarely occurring activation patterns. Nonetheless, tests build trust among the users of the system. Furthermore, tests validate a system on all levels simultaneously. It is also easy to redo tests automatically when any part of the system is modified. Re-doing a verification proof may be difficult and require interaction and/or a proof-reuse mechanism.

[5] "A Hoare logic that is unsound would be useless since its very purpose is to verify correctness of programs. Thus after giving a Hoare logic the proof of its soundness is obligatory, in particular when—like in our case—the rules are rather involved and thus their correctness is by far not obvious." [von Oheimb, 2001b]

[6] "The proof rules are specified in KIV and their correctness with respect to the [own] semantics has been proved. [...] All 57 rules have been proved correct. The specification and verification effort required several months of work. [...] As can be imagined several errors were found during verification. Most of them are errors only for type incorrect programs." [Stenzel, 2005]

[7] The UK Defence Standard 00-55 "Requirements for Safety Related Software in Defence Equipment" [UK Ministry of Defence, 1997] demands that "[...] there should be at least a peer review of the proof obligations and formal arguments [by a member of the team] other than the author [...]".

[8] "However, both semantics and calculus could be wrong. It is possible to validate the semantics by 'running' test programs in KIV (automatically applying the proof rules) and comparing the output with a run of a Java compiler and JVM (currently 150 examples), and this certainly increases confidence in the semantics [...]" [Stenzel, 2005] (the author goes on to argue that both testing and verification of the calculus are needed).

Measures against faults in the user input

User input is, by its very nature, not part of the reasoning system. The formulation of queries is, however, part of the reasoning process and, thus, the adequacy of queries is relevant for the process's reliability.

A fault in user input is present whenever the query ϕ does not mean what the user thinks it means. A correct answer to the question "$\mathcal{T} \vdash \phi$?" will in this case not have the expected impact in the real world.

User input cannot easily be verified or tested. But, apart from many systematic approaches for elicitation of requirements or construction of ontologies (which we will not cover here), there is a number of ways in which deduction technology can assist the user to formulate meaningful queries.

First, the builders of deduction systems can work on formalisms that do not make it unnecessarily hard for the users to express their exact intentions. Second, the deduction systems can produce a proof or a trace to justify the deduction result. Inspection of the proof is a very effective—if costly—measure to combat misunderstandings in the meaning of the query.

Third, a whole new class of sanity checks based on mutation has been developed lately for automated program verification [Kupferman, 2006]. After a successful verification attempt, the query (the program or the specification) is mutated and the deduction is repeated. If verification succeeds again, then the mutated part of the query probably plays no role in determining the outcome. This indicates a problem with the query.

9.4.3 Scalability of Fault Removal Approaches

When choosing a validation method, one needs to consider its scalability. There are two things that can make it hard to find a particular fault in a system: the size and complexity of the system and the "complexity" of the fault. The latter is a function of how complex and how rare an input is that activates the fault and causes a failure. Validation methods must be sufficiently scalable in both dimensions.

An additional factor to consider is how easy it is to re-validate a system, i.e., to reuse and adapt a correctness proof or to re-run tests, when a part of the reasoning system is changed or used in a different configuration (as both happens frequently in practice).

The scalability of theory validation depends on the domain. For example, when validating a theory of a programming language, no scaling-up in size is required, since the size of the theory is fixed or at least clearly bounded. The validation method has to be able to handle a theory of that particular size. Scalability becomes relevant again when theories of standard libraries are added to the system. Java programs, for instance, rely heavily on libraries shipped with the language. The good news is that libraries are independent from each other and from the core language. Adding a theory for a library does not create faults in other theories. This emphasizes the importance of modularization when dealing with large-scale problems.

A measure for the complexity of faults (i.e., the second dimension of scalability) in a programming language theory is the number of language features that are involved. For example, the rule for handling while-loops in Java has to consider the possibility that the loop body throws an exception and, thus, terminates abruptly. If the loop rule is faulty and does not cover the case of abrupt termination, then this problem can only be found with a test case involving both features: while-loops and exception throwing.

Finding very complex faults is difficult to do by testing. This problem is mitigated, however, by the fact that programming languages are designed for humans by humans. Language designers try to make individual features as independent as possible, since otherwise the language is hard to understand and use for programmers. Altogether, the question of scaling along the fault complexity dimension is an argument in favor of verifying the verification system—but not a very strong one.

The situation is more problematic with ontologies. Modern ontologies dwarf other theories in size, and the domain they model is, in most cases, part of the real world and not human-designed. Faults can thus span a large number of features (concepts). Even if a modularization of the domain is possible, it is often not readily available. Moreover, ontologies tend to change and evolve quickly.

9.4.4 Consequences of Residual Faults

Since attaining a fault-free system is very difficult, we must consider the consequences of residual faults. The most problematic fault class are catastrophic faults, i.e., faults that lead to the system performing arbitrary deductions. The biggest potential for catastrophic fault lies, in our experience, in the core calculus of the system (e.g., a faulty induction or Skolemization rule). Therefore, the core calculus has to be validated to the highest reliability levels.

In theories of programming languages, on the other hand, most of the (numerous) axioms correspond to particular features of the language. Therefore, the effects of a fault in most cases remain localized and do not lead to catastrophe. That is, verification proofs for (parts of) programs not containing the particular feature are not affected, which may be the very reason why a fault remains undetected.

The sheer size and the dynamic nature of ontologies may prevent effective fault removal. Thus, development of techniques for non-trivial reasoning with faulty, and in particular inconsistent, ontologies is still a hot research topic [Huang et al., 2005]. Possible solutions include paraconsistent logics, multi-valued logics, reasoning with consistent subsets of a theory, etc.

9.5 Finding Faults in Practice

9.5.1 The Situation in the KeY Project

The KeY tool is a mature and established verification system for Java with high coverage of the language. The KeY team has stated only one theory of Java (i.e., the KeY calculus). We have refrained from stating two theories and proving their equivalence.

Resources saved on this were instead spent on further improvement of the system. At the same time, the KeY project performs ongoing cross-verification against other people's Java formalizations to ensure adequacy.

One such effort compares the KeY calculus with Bali [von Oheimb, 2001b], which is a Java Hoare Logic formalized in Isabelle/HOL. The published result [Trentelman, 2005] describes in detail cross-verification of the rules for local variable assignment, field assignment and array assignments. These rules are of particular importance to every Java theory. The KeY rules were translated manually into Bali rules, and these were then proven sound with respect to the rules of the standard Bali calculus. This is how the missing check for an `ArrayStoreException` in the array assignment rules was detected (⇒ Sect. 10.8.6).

A different approach has been taken by Ahrendt et al. [2005b]. It takes as a reference another Java semantics [Farzan et al., 2004], which is formalized in Rewriting Logic and mechanized in the input language of the Maude system. This semantics is an executable specification, which together with Maude constitutes a Java interpreter. The nature of this semantics made it particularly suitable for verifying program transformation rules of KeY. These are rules that decompose complex expressions, take care of the evaluation order, etc. (about 45% of the KeY calculus). For the cross-verification, the Maude semantics was "lifted" in order to cope with schematic programs like the ones appearing in KeY. The rewriting theory was further extended with means to generate valid initial states for the involved program fragments, and to check the final states for equivalence. The result is used in frequent completely automated validation runs, which is beneficial, since the KeY calculus is constantly extended with new features.

Furthermore, the KeY calculus is regularly tested against the compiler test suite Jacks [Jacks]. The suite is a collection of intricate programs covering many difficult features of the Java language. These programs are symbolically executed with the KeY calculus and the output is compared to the reference provided by the suite. This approach has also been taken by others [Stenzel, 2005].

All of the above methods have found faults in the KeY system (in the calculus, in the Java theory, and in the implementation), while none of the methods alone would have been sufficient to uncover them all. A balanced mix of validation methods is necessary to attain high reliability at reasonable costs. In our experience, an important role is also played by good software engineering practices, such as extensive unit tests, bug tracking, peer review of code, etc.

9.5.2 Reliability and Deduction System Competitions

Most competitions for reasoning systems today assume that reliability is something that systems must have by definition. Wrong answers are often considered an embarrassment for both the system implementors and the competition. Both the rules of CASC [Sutcliffe, b] and the SAT Competition [Le Berre and Simon], for instance, state that a system exhibiting unsoundness will be disqualified. Competitions, however, should acknowledge that reliability is a criterion for comparing systems. The SMT-COMP [Stump] competition already does this.

The history of CASC shows that many participating systems have soundness faults, which is not surprising as developers always submit the latest versions of their systems. Since 1996, when the first CASC was held, 27 systems were disqualified in 12 installments of CASC [Sutcliffe, a] (an average of two per year) because they failed a soundness test by giving a wrong answer to at least one problem from the TPTP library. More faults probably existed in the participating systems but remained undetected.

Of course, a systems exhibiting unsoundness should not win a competition. But instead of merely disqualifying a faulty system, one should investigate the reasons and publish the detailed findings of the investigation so that others may learn how to avoid such faults.

9.6 Recommendations

As a conclusion, we summarize our recommendations for ensuring reliability of reasoning systems, in particular if they use large domain-specific theories.

There is a trade-off between reliability and other qualities.

Reliability is a critical property for reasoning systems. But there are other important qualities as well: functionality, dependability (of which reliability is one aspect), usability, performance, and cost.[9] Moreover, reliability is a measure of probabilities: no system is 100% reliable. Consequently, reliability is not an absolute, but there is a trade-off between reliability, i.e., probability that the right answer is given, and other properties of the reasoning system. All properties should be considered when the quality of a system is evaluated.

When developers think about how they should improve their system, they should analyze which kinds and what frequency of failures would be acceptable to the users.

Reliability is a *gesamtkunstwerk.*[10]

All ingredients of the reasoning process contribute to its reliability. Validation of all ingredients should be taken into consideration. One cannot claim reliability just by proving the calculus to be sound if the implementation has not been thoroughly tested.

Also, deduction systems should be built such that they support the search for inadequacies in domain theories or queries.

[9] "Of course, a theorem prover should be sound. [...] However, also efficiency is an important consideration in the design. If a tool is sound, but too slow, it is not useful for verifications of larger systems. Also, as explained above, even though PVS contains soundness bugs, it is still a great help in specification and verification, since most of the time it works 'correctly'." [Huisman, 2001a]

[10] The term *gesamtkunstwerk*, which might be translated from German as "synthesis of the arts", is commonly used to describe any integration of multiple art forms.

Both formal and conventional validation methods have their strengths and their weaknesses.

Neither conventional nor formal methods are inherently superior. The reasoning community has to be at the forefront of users of formal methods, but practitioners using conventional methods also have a point. We should use a balanced mixture of both approaches.

Use cross-verification.

If theories are verified, they should be cross-verified against other people's formalizations of the domain laws. With cross-verification, the probability of uncovering errors is much higher.

Do not draw the system boundaries artificially tight.

One should not turn adequacy of the domain theory or the query into a non-issue or somebody else's problem by defining the theory or query to be outside the system's boundaries. That does not solve the problem of making the reasoning process reliable.

Investigate and publish the reasons for failures.

The reasons for failures should be investigated, and immediate feedback to design and operation should be given. The investigation should go beyond finding the fault that caused the failure. "Why was the error made in the first place?", "how could it have been avoided?" and, "why wasn't it detected before?" are also important questions. The findings of the investigation should be published so that others may learn how to avoid errors of the same kind.

Also, being open about failures and the faults causing them (as opposed to hiding them as an embarrassment), builds trust among the users of the system.

Reliability should be a criterion in competitions.

Competitions for reasoning systems should treat reliability as a (high-impact) criterion for system quality. They may bar unsound systems from winning but should report their performance results together with an analysis of the soundness problem and its impact.

We are thankful to the anonymous Reviewer 1 for very helpful comments on an earlier version of this text.

Part IV

Managing Change in Deductive Program Verification

10

Applying Proof Reuse in the Verification Cycle

Experience shows that the prevalent use case of program verification systems is not a single prover run. Most of the time verification engineers iterate verification attempts. This happens for a plethora of reasons, such as a fixed bug in the code, an extension to the program, a revised specification, a new try after a failed proof attempt, or even a change in the proof system itself.

In such a case, if the change is small, it is often better to adapt and reuse the existing partial proof(s) than to verify the program again from first principles. A particular advantage of proof reuse for interactive verification systems is that it can reduce the total number of user interactions.

Here we present such a technique for proof reuse. We have developed this technique earlier to help recycle proofs after fixing bugs (this is indeed the scenario that we will use to explain the technique). New in this thesis is how our method can improve the user experience for a whole range of everyday verification scenarios.

After discussing the features of the method, we will introduce a small running example, cover the theoretical and practical details of proof reuse, examine other solutions to the problem, and finally survey a wide range of proof reuse applications in deductive verification of Java software.

10.1 Introduction

Features of Our Reuse Method

The main features of our reuse method are:

(1) The units of reuse are single rule applications. That is, proofs are reused incrementally, one proof step at a time[1]. This allows us to keep our method flexible, avoiding the need to build knowledge about the target programming language or the particular calculus rules into the reuse mechanism. Another consequence of this feature is the guaranteed soundness of proofs, since the usual rule application mechanism of the prover is used for proof construction.

[1] Alternative approaches are discussed under "related work" (⇒ Sect. 10.7).

(2) Proof steps can be adapted and reused even if the situation in the new proof is merely similar but not identical to the template.

(3) In case reuse has to stop because a changed part in the new program is reached that requires genuinely new proof steps, reuse can be resumed later on when an unaffected part is reached. The system detects when this is the case.

A Review of Basic Notions and Definitions

At this point, we review some important calculus-related notions from Section 2.5. As usual, we assume that rules are represented by *rule schemata*. Rule instances are derived from rule schemata by instantiating schema variables. In the following, we identify rules and their schema representations.

A *proof* for a goal (a sequent) S is a tree with S at the root. A proof is constructed by matching an open goal with the conclusion of a rule and extending the tree at this point with child nodes (sub-goals) corresponding to the premisses of the rule. Rules without premisses (axioms) finalize this process at a given goal. A *rule application*, thus, consists of a rule instance and a node in the proof tree that is a logical consequence of its child nodes via this instance.

Most rules have a *focus*, i.e., a single formula, term, or program part in the conclusion of the rule that is modified or deleted by applying the rule. The focus of the if rule in Section 2.7.2, for example, is the `if`-statement. An example for a rule that does not have a focus is the cut rule; it can be applied anywhere.

A Running Example

We now motivate our approach using a simple example. While utterly contrived, this example is well-suited to give insight into the setting and the mechanics of proof reuse.

Consider the following program:

```
int x;
int res;
res=x/x;
```

Java

Its intended behavior and specification is that it should always terminate with `res` set to 1. The program, however, contains a bug and cannot be proven correct, since an arithmetic exception can be thrown on division by zero.[2] Figure 10.1 (a) shows the beginning of the failed correctness proof. It has one open branch (the "division by zero" branch) where an exception is thrown. The other branch (the "normal execution" branch) can be closed. We will use this proof as a template for reuse and refer to it as "old proof".

We now amend the program and obtain the following "new" version:

[2] In fact, Java requires initializing the program variable `x`. However, here we treat `x` as if it were an input parameter with unknown value. The variable declarations play the role of the leading program part that is not affected by the bug fix.

```
int x;
int res;
if(x==0) {
   res=1;
} else {
   res=x/x;
}
```
Java

This new program is correct w.r.t. the specification. It always terminates with `res` set to 1. Figure 10.1 (b) shows the beginning of the proof for this, which consists of a completely new branch for the case that `x` is zero (shaded) and a "non-zero" subproof that handles the division statement.

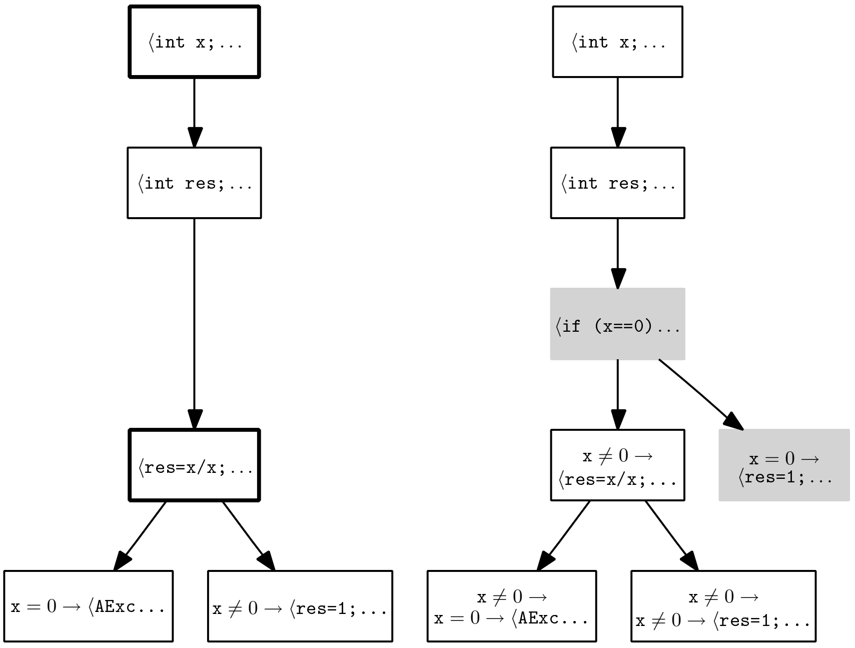


Figure 10.1. Schematic proofs (a) before and (b) after program correction. The leftmost branch of the old proof cannot be closed, since the program contains a bug. `AExc` is shorthand for `throw new ArithmeticException();`

Comparing the old and the new proof we can see that there are parts that are in some way common to both. We can also see that in the new proof these recyclable parts are interspersed with proof steps that are genuinely new. Furthermore, the formulas in the new proof are not always identical to their counterparts: some have additional premisses, but the similarity is discernible. This is a common situation where proof reuse is called for. We will return to this example and show how reuse works for it in Section 10.6.

10.2 The Main Reuse Algorithm

Basic ideas

As said in the introduction, we start with two versions of a program: an old one, and a corrected new one. We also have two proofs in the system: the old, template proof dealing with the old program—it may or may not be a complete proof—and an incomplete new proof dealing with the new program. At the beginning, the new proof is a tree of a single node. This initial proof goal is constructed from the new program and the specification, which we assume to have remained unchanged.

For each application of the reuse facility—as for any interactive proof step—there are choices to be made:

(a) the rule (schema) to be applied
(b) the focus of application, i.e., a suitable goal/position
(c) instantiations for schema variables.

On the one hand, our goal is to make in the new proof—if possible—the same choices as in the template proof. On the other hand, we expect the two to have parts, which evolve in a similar but not identical manner. This requires us to generalize and extract the essence of the above choices in the old proof.

For finding the rules that are candidates for choice (a), such a generalization is readily available. The rule schemata are natural generalizations of particular rule applications. We then try to adhere to the overall succession of rule applications in the template proof. But, since proofs are not linear, at each point in time there can still be several candidate rules that compete for being used first.

Choice (b), i.e., the point where a given candidate rule is to be applied, is more difficult as it is hard to capture the essence of a formula or sequent. To solve this problem, we define a similarity measure on formulas (⇒ Sect. 10.3). Fortunately, there is usually only a moderate number of possibilities, because program verification calculi are to a large degree "locally deterministic". That is, given a proof to be extended, most rule schemata only have a small number of potential application foci.

These combinations of candidate rules and their potential focus points—which we call *reuse pairs* in the following—are ordered according to the similarity between the potential focus in the new proof and the actual focus in the template proof. Thus, the similarity measure both implements the generalization for choice (b) and is used to prioritize the rule candidates left from choice (a).

Finally, to make choice (c), schema variable instantiations are computed by matching the rule schema against the chosen focus of application. Schema variables that do not get instantiated that way, e.g., quantifier instantiations, are simply copied verbatim from the old proof.

The main algorithm

The main reuse algorithm is shown in Figure 10.2. It maintains an unsorted list C of distinguished rule applications in the template proof, which are the *reuse candidates*. While reuse progresses and the new proof grows, those old rule applications that are considered currently available for reuse are listed in C. In the beginning, C is initialized with the list of *initial candidates* C_0, which is computed by the function *initialCandidateList* from the differences in programs.

```
input $oldProof, oldProgram, newProgram, specification$;
$newProof$ := initialProofGoal($newProgram, specification$);
$C_0$ := initialCandidateList($oldProof, \Delta(oldProgram, newProgram)$);
$C := C_0$;
while $newProof$ has open goals do
    $\langle candidate, newFocus \rangle$ := chooseReuse($C, oldProof, newProof$);
    if $\langle candidate, newFocus \rangle \neq \bot$ then
        $newProof$ := result of applying rule($candidate$) at $newFocus$ in $newProof$;
        if $candidate \notin C_0$ then $C := C \setminus \{candidate\}$; fi;
        $C := C \cup \{c \mid c$ is a child of $candidate$ in $oldProof\}$;
    else
        $newProof$ := applyRuleWithoutReuse($newProof$);
    fi;
od;
output $newProof$;
```

Pseudocode

Figure 10.2. Main reuse and proof construction algorithm

At each iteration step, the function *chooseReuse* is invoked to compute all potential *reuse pairs* and choose the most appropriate one. A reuse pair consists of (1) a candidate rule application and (2) a potential new focus, i.e., a position in a goal sequent of the new proof, where the same rule is applicable. The implementation of *chooseReuse* is shown in Figure 10.3.[3] For the reuse pair selection process *chooseReuse* employs the similarity function *score*, which will be discussed later on. The function *score* is mainly based on focus similarity.

[3] We show a nested loop implementation for its clarity. The actual implementation uses an optimized incremental computation algorithm.

```
function chooseReuse(list $C$ of candidates, $oldProof$, $newProof$)
possibleReuses := {};
Goals := open goals of newProof;
foreach $c \in C$ do
    foreach $g \in Goals$ do
        foreach position $p$ in the sequent of $g$ do
            if the rule schema of $c$ is applicable at $p$ then
                $possibleReuses := possibleReuses \cup \langle c, p \rangle$;
            fi;
        od;
    od;
od;
if $possibleReuses = \{\}$ then return $\perp$ fi;
select $\langle c, p \rangle$ from $possibleReuses$ with score($\langle c, p \rangle$) maximal;
if score($\langle c, p \rangle$) $> \varepsilon$ then
    return $\langle c, p \rangle$;
else
    return $\perp$;
fi;
```

Pseudocode

Figure 10.3. Function for the best possible reuse pair

The rule of the selected reuse pair is then applied at the target focus, extending the new proof. The candidate rule application is removed from the list *C*.[4] Finally, the children of the used candidate rule in the old proof tree become new candidates and are added to *C*.

In other words: the proof steps appearing in the list *C* at a given time can be considered as marked in the template proof. These markers form a "wavefront" extending through the old proof tree during reuse. The markers are indeed visible in the KeY prover as »-signs attached to nodes of the template proof tree.

So far, two very important questions remain open: how is the quality of possible reuse pairs computed (i.e., how does the function *score* that is used by *chooseReuse* work)? And where do the initial candidate proof steps come from (i.e., how does the function *initialCandidateList* work)? These questions are answered in Sections 10.3 and 10.4, respectively. Note that our algorithm is "modular" in the sense that the answers can be given independently.

[4] Unless it is an initial candidate (i.e., an element of C_0), in which case it is persistent in *C*. The reason for making the initial candidates persistent is explained in Section 10.4.

Avoiding confusion: a quality threshold

While performing reuse, the danger is not only to do too little, but also to do "too much". Sometimes, even though there are possible reuse pairs available, it is better to use none of them. This is not so odd as it seems, since a reuse pair's existence alone means little more than a *possibility* of applying a single rule. Whether the rule is appropriate in a particular context is another question.

The most prominent opportunity for exercising restraint is when a genuinely new situation in the new proof is reached. In this case we want reuse to stop, since reuse pairs used up here would not in general be available when an unaffected proof part is reached again. This does not undermine the correctness of the proof under construction (since the prover only allows correct rule applications), but it can confuse the user and impede performance.

To safeguard against confusion, we compare the quality scores of reuse pairs to a threshold value ε. In case the score of all possible reuse pairs is below ε—which is an indication that we have reached a situation that is either different or not present in the old proof—a completely new proof step has to be chosen by the user or the automated proof search procedure (this choice is symbolized by calling *applyRuleWithoutReuse* in the algorithm). In the meantime, the system constantly checks whether reuse can be restarted using one of the available candidates.

What to do with instantiations?

For some rules it is not sufficient to know *where* they will be applied (i.e., what their focus is), but additional information is required. For example, (a) the cut formula has to be known for an application of the cut rule, (b) for induction rules, the induction hypothesis has to be known, and (c) for quantifier rules, the appropriate instantiation has to be provided. Since it would be a very hard task to adapt this kind of information from the old rule application to the new one, we currently attempt to use the same information as in the old proof.

10.3 Computing Rule Application Similarity

Recall that a possible reuse pair consists of a rule application in the old proof and a focus (formula, term, or program) in the new proof where the same rule is applicable.

The similarity score for quality assessment of possible reuse pairs is a key part of our reuse facility, since it is one of the most crucial and difficult parts in our effort. We have to distinguish between proof parts that are appropriate for reuse in a given situation and parts that only seem to be so on first sight. In other words, similarity scoring must prevent mis-application of proof steps from the old proof that are not appropriate for reuse.

When all possible reuse pairs have been computed for an iteration step of the reuse algorithm, we are (usually) left with a choice. Several features may influence the quality of a reuse pair. The first and most important one is the similarity between the application foci in the old and the new proof. How it is computed is described in detail in the following, where we distinguish three kinds of rules:

Rules for symbolic execution, which focus on a program. The similarity score is determined by comparing the focus programs in the old and the new proof. The non-program parts of the formulas in question are not considered, since in our calculus they rarely provide additional discriminating evidence.

Analytic first-order logic and rewrite rules, which manipulate a (sub-)formula or term without modifying program parts. A similarity analysis of the foci tailored to the first-order fragment is performed.

Focus-less rules, which are the few rules of our calculus, that do not have a focus. The score of such a reuse candidate is solely based on other features, in particular proof connectivity.

To get a single numerical quality value for a reuse pair, we sum up the scores computed for different features.

Similarity Score for Program Parts

We evaluate the appropriateness of symbolic execution proof steps by comparing the programs that these steps focus on. In general, symbolic execution rules only touch the first statement of a program. Our comparison is not limited to the first statement though, the entire focus programs are considered as well.

A straightforward way to compare two programs is to compute the edit distance between them, which is the length of the minimal edit script for turning one program into the other. Since, for example, the particular names of variables, methods, etc. have no effect on the structure of proofs, we use an abstraction of actual programs for comparison.

Below, the following steps of the comparison are explained in more detail: (1) the algorithm for computing the minimal edit script, (2) the program abstraction that we use, and (3) the computation of a numerical similarity score from an edit script.

Computing the minimal edit script

Currently, our similarity assessment function treats programs as linear sequences of symbols. Experiments with this implementation show that it is an efficient and successful way to compare programs for our purposes. Theoretically, a program similarity measure based on a tree editing distance algorithm (e.g., [Zhang and Shasha, 1989]) would yield even better discrimination.

In the following we use Myers's classical Longest Common Subsequence (LCS) algorithm [Myers, 1986] to efficiently compute the minimal edit script of two sequences of symbols. It takes two sequences

$$A = a_1\, a_2 \cdots a_N \quad \text{and} \quad B = b_1\, b_2 \cdots b_M$$

as input, where the a_i and b_j are elements of an arbitrary alphabet, and produces the minimal edit script for turning A into B.

An edit script is a list of insertion and deletion commands. The delete command "$x\, D$" deletes the symbol at position x from A. The insert command "$x\, I\, b_1\, b_2 \cdots b_t$"

inserts the sequence of symbols $b_1 b_2 \cdots b_t$ immediately after position x. The script commands refer to symbol positions in A after the preceding commands have been executed. The length of the script is the number of symbols inserted or deleted.

Program abstraction

The computation of a minimal edit script requires as input two sequences of symbols. To construct such sequences from the two programs that are to be compared, we first linearize the programs into a sequence of statements. Then, the statements are abstracted into *statement signatures*.

Statement signatures are defined to abstract from names, expressions, most literal values, etc. That is, they are designed to remove all features that tend not to influence the shape of the control flow and, thus, proof structure. Abstraction reduces noise and increases reuse performance. As a byproduct, it allows our algorithm to deal with such program changes as renamings and changes of literal values. This "coarsening" approach has parallels to the technique of boolean program abstraction [Ball and Rajamani, 2000], which produces an equivalent—in some sense—program with a reduced state space. In contrast, we are only interested in a means to syntactically discern related and unrelated programs and not in behavioral refinement.

The first element of the abstraction of a statement S is the name of S (e.g., *If*, *LocalVarDecl*, *Assignment*). In the following cases, more details are added to the abstraction:

- If the statement S is also an expression, the static type of the expression is added. If, moreover, S is an assignment whose right operand is a `boolean` literal, then the value of that literal is appended as well.
- If the statement S is a method invocation, the signature of the method and the name of the class containing the referenced implementation are added.

The `boolean` literal assignment has indeed to be treated in this special way. First, the symbolic execution rules of our calculus often introduce two symmetrical assignments of this kind when branching upon JAVA's relational and equality expressions. Without the special treatment, the two branches would be indistinguishable. Also, the small domain of the `boolean` data type and the direct impact of the particular value assigned on the control flow do not permit removal of this information.

Example 10.1. Consider the following two programs α and β:

$$\alpha = \begin{cases} \texttt{int x; int res;} \\ \texttt{res = x/x;} \end{cases}$$

$$\beta = \begin{cases} \texttt{int x; int res;} \\ \texttt{if (x==0) res=1; else res=x/x;} \end{cases}$$

The result of abstracting them into sequences A resp. B of signatures is:

$$A = \begin{cases} \texttt{LocalVarDecl}, \texttt{LocalVarDecl}, \\ \texttt{Assignment(int)} \end{cases}$$

$$B = \begin{cases} \texttt{LocalVarDecl}, \texttt{LocalVarDecl}, \\ \underline{\texttt{If}}, \texttt{Assignment(int)}, \underline{\texttt{Assignment(int)}} \end{cases}$$

The underlined parts correspond to the insertions in the minimal edit script. It consists of the two commands $2\,I\,$`If` and $4\,I\,$`Assignment(int)`. ◁

One could devise more elaborate abstraction schemes. Our experience, though, shows that this only leads to a marginal improvement.

From edit script to similarity score

To compute a similarity score for two programs α and β, we have computed a minimal edit script between their abstract representations A and B. Now we must condense this edit script into a single numerical value.

Definition 10.2 (Program similarity score). Let $E(A,B) = e_1\, e_2 \cdots e_n$ be the minimal edit script for the abstractions A, B of programs α, β. Then, the *similarity score* of A, B resp. α, β is defined by

$$\delta(\alpha,\beta) = \delta(A,B) \quad = \quad -\sum_{i=1}^{n} P(e_i)$$

where the penalty $P(e)$ for an edit command e is[5]

$$P(e) \quad = \quad \begin{cases} \displaystyle\sum_{k=1}^{t} \frac{0.75}{x+k} & \text{if } e = x\, I\, b_1\, b_2 \cdots b_t \\ \displaystyle\frac{1}{x+1} & \text{if } e = x\, D \end{cases}$$

We remind that x is the numeric position of the insertion/deletion as counted from the beginning of the linearized program. ◁

Note that higher values of $\delta(\alpha,\beta)$ mean higher similarity, and that $\delta(\alpha,\beta)$ is always less than or equal to zero. The maximal value 0 is reached for programs with identical signatures. The quality threshold is chosen at -0.72 for the given values of penalty constants.

The function δ is not symmetric, meaning that in general $\delta(A,B) \neq \delta(B,A)$. Statement insertions are penalized less than deletions. The reason for defining δ in that way is that additional statements in the new program are easier to handle for reuse than missing statements. Deleting statements does usually not simply correspond to deleting proof parts but requires more complex changes of the proof.

Program differences are penalised less the farther they are from the active (first) statement, which is the target of symbolic execution.

[5] Please note that all numbers provided here are for orientation purposes only. The numbers in your version of the KeY system may vary.

Example 10.3 (Example 10.1 continued). We now consider the minimal edit script for the programs α and β presented above. It consists of the two commands $2I$`If` and $4I$`Assignment(int)`.

The similarity score is thus:

$$\delta(\alpha,\beta)=\delta(A,B) \quad = \quad -\frac{0.75}{2+1}+-\frac{0.75}{4+1}=-0.4 \ ,$$

which signifies a medium to high similarity. The score is above the threshold and warrants reusing the application of the local-variable-declaration rule from the old proof in the new one. $\triangleleft$

Similarity Score for First-order Logic Parts

Assessing the quality of possible reuse pairs that do not deal with symbolic program execution is a more difficult challenge. This is due to the lower degree of local determinism of the first-order fragment of the calculus and the high "volatility" of first-order formulas in a proof.

We use two different similarity criteria for first-order-related proof steps. First, a high bonus (+1.0) is added to the quality score if the foci in the old and the new proof are identical up to variable renaming. Otherwise, a small penalty (−0.2) is added. Second, the two formulas that contain the actual rule application foci are compared in a similar manner as programs: formulas are linearized, then the names of variables, functions, etc. are abstracted to their sorts, and finally a minimal edit script is computed. The script is scored uniformly, with every deletion worth a penalty of 0.1 and every insertion a penalty of 0.05. Additionally, the programs in the formulas contribute their similarity scores with a weight of 0.25.

The results of using these criteria are sufficient for a high ratio of correctly reused rule applications but are not as good as for rule applications with a program part in focus.

Similarity Score for Focus-less Rules and a Refinement Based on Proof Connectivity

An additional feature that can be used to score possible reuse pairs (besides similarity of rule foci), is the *connectivity* of the new proof (as compared to the old proof). This criterion gives a bias against tearing apart proof steps that are connected in the old proof. Reuse pairs disrupting connectivity are assigned a small penalty (of −0.1). This is enough to tip the scales in case other features do not provide discrimination between several possible reuse pairs.

10.4 Finding Reusable Subproofs

Our main reuse algorithm requires an initial list of reuse candidates. These initial candidates, which are rule applications in the old proof, can be seen as the points where the old proof is cut into subproofs that are separately reusable. They are the points where reuse is re-started after program changes required the user or the automated proof search mechanism to perform new rule applications not present in the old proof. The choice of the right initial candidates is important for reuse performance.

Since program changes may lead to additional case distinctions in the new proof, it may be necessary to reuse old subproofs repeatedly in the new setting. In order to deal with this necessity, we make the initial candidate proof steps persistent. As shown in Figure 10.2, the *initial* candidates (they are the elements of C_0) are not consumed when they are reused. Thus an initial candidate proof step is always available to seed the corresponding template subproof when needed.

The way initial candidates are computed depends on the way the program and thus the initial proof goal has changed. For changes affecting single statements (local changes) we extract the differences right from the source files, using an implementation of the GNU diff utility (`www.bmsi.com/java/#diff`) in JAVA. The diff utility is based on the same algorithm by Myers [Myers, 1986] that we use for program similarity scoring. GNU diff is well-known to produce meaningful change sets for modifications of source files. A number of heuristics help identify common sections of code in the old and the new program based on diff output. The proof fragments dealing with these common parts are good candidates for reuse; thus, their root nodes are marked as initial reuse candidates.

In the KeY system, the differences between program revisions are provided by the integrated source tracking system based on CVS, which in turn uses GNU diff. Based on that information, markers for initial reuse candidates are automatically inserted by our reuse facility into the proof to be reused.

```
int x;
int res;
res=x/x;
```

(a)

```
int x;
int res;
if(x==0) {
   res=1;
} else {
   res=x/x;
}
```

(b)

```
- old
+++ new
@@ -1,3 +1,7 @@
 int x;
 int res;
+if(x==0) {
+  res=1;
+}else {
 res=x/x;
+}
```

(c)

Figure 10.4. Change detection with GNU diff: (a) old program, (b) new program, and (c) output of "`diff -uw`"

Example 10.4. The output of GNU diff for our running example is shown in Figure 10.4. The first three lines show bookkeeping information (names of the compared files, position of the difference found). The lines after this starting with "+" have been added to the old program. Lines starting with a "`-`" (not occurring here) have been removed from the old program. Lines starting with a space are common to both programs.

In this example, the common program parts start with the statements `int x;` and `res=x/x;`. Thus we scan the old proof top-down and look for proof steps with these statements in focus. This procedure yields two initial reuse candidates for our example. These are the proof steps with the bold border in Figure 10.1 (a). ◁

Caveats and limitations

We have to note that the heuristics used to detect initial reuse candidates are quite accurate but not infallible. Their biggest adversary is again the fact that program structure is more adequately represented as a tree than as a linear sequence of symbols, which is the view we take.

The detection performance can further be impaired, for example, if the programmer puts several statements on one line. Given that this is (a) not too common and (b) explicitly discouraged by the official JAVA Coding Conventions [Sun Microsystems, Inc., 2003], we did not provide a solution (such as an additional intra-line diff).

Also, non-local changes, such as renaming of classes or changes in the class hierarchy, cannot be detected in a meaningful way by the standard diff algorithm; the user has to announce these changes separately. In the meantime, techniques have been developed for computing a precise and semantics-aware diff of two JAVA programs [Apiwattanapong et al., 2004]. Unfortunately, this work is limited to JAVA bytecode, which complicates the workflow in a source-based verification system.

10.5 Implementation and a Short Practical Guide

To profit from reuse we simply have to load another instance of a problem already present in the prover. A dialog will appear asking whether we want to reuse a previous proof. If we say yes, the system will analyze the differences in the source code, compute initial reuse candidates, and, if reuse is indeed possible, enable the »-marked reuse button.

Hitting the button activates the reuse process. Should reuse stop, the system will indicate its idea of how the proof continues via a message in the status line: **template proof continues with ⟨rulename⟩**. We can hit Alt-space to switch the view to this particular proof step. Hitting Alt-space again takes us back to the open goal in the current proof. This can give us some idea of where to steer the proof. Now we have to perform proof steps interactively or run a strategy. Once a state is reached where reuse is possible again, the reuse button will be enabled.

The candidate proof steps ("reuse candidates") are always distinguished in the template proof by a »-sign at the corresponding node of the template proof tree. It is possible to add or remove candidate markers at any time via the context menu of a proof node. For this, the context menu offers the item **mark for reuse**, which toggles the marked state.

In order to provide feedback, the reuse facility can color the nodes in the proof tree it constructs with different colors. The ecru (yellowish) nodes are the ones created by the reuse procedure. Red nodes are the ones where the connectivity of the old proof has been broken for some reason.

10.6 The Example Revisited

We trace the first few interesting steps in detail, while slightly simplifying the presentation for clarity (e.g., the connectivity feature is not considered).

First, we need to compute a set of initial reuse candidates based on the differences between the old and the new version of the program (both given in the introduction). How this is done is explained in Example 10.4, which shows that we obtain two candidates in our case. These are the nodes with a bold border in Figure 10.1 (a).

For now, we only consider the first one, namely the rule for variable declarations applied to "`int x;`" in the old proof (the rule of the second initial candidate concerning "`res=x/x;`" is not applicable anyway). It has one possible focus in the following (new) initial proof goal (it cannot be applied to the second variable declaration, since our calculus always treats the left-most statement first):

$$\Longrightarrow \langle \texttt{int x; int res;} \\ \quad \texttt{if (x==0) res=1; else res=x/x;} \rangle (\texttt{res} = 1) \tag{G0}$$

The similarity score for the single possible reuse pair (see Example 10.1 for the computation) is -0.4, and reuse is performed. We get the new goal

$$\Longrightarrow \langle \texttt{int res;} \\ \quad \texttt{if (x==0) res=1; else res=x/x;} \rangle (\texttt{res} = 1) \tag{G1}$$

and a new reuse candidate (the child of the initial candidate in the old proof), which is again an application of the rule for variable declarations, this time applied to "`int res;`". It also has one possible focus in the new proof in goal (G1). The similarity score for the resulting possible reuse pair is -0.62. This is less than before as there are now fewer identical parts in the programs of the old and the new focus, and the first difference is closer to the active statement. Nevertheless, reuse is still indicated. The resulting new goal sequent is

$$\Longrightarrow \langle \texttt{if (x==0) res=1; else res=x/x;} \rangle (\texttt{res} = 1) \tag{G2}$$

and the new candidate is the rule handling the assignment "`res=x/x;`" in the old proof (which happens to be identical to the second initial candidate). This candidate, however, is not applicable in (G2). We have reached a genuinely new part of the amended program and, thus, of the proof.

To deal with the new program parts, where no reuse is possible, we manually apply the rules for handling the `if` statement and evaluating its condition (in practice this can be done automatically). The proof tree splits, and we get two subgoals:

$$\Longrightarrow \mathtt{x}=0 \longrightarrow \langle\mathtt{res=1;}\rangle\,(\mathtt{res}=1) \tag{G2.1}$$

$$\Longrightarrow \neg(\mathtt{x}=0) \longrightarrow \langle\mathtt{res=x/x;}\rangle\,(\mathtt{res}=1) \tag{G2.2}$$

There are still two identical candidate proof steps with the rule tackling "`res=x/x;`". This rule cannot be applied to (G2.1), as handling an assignment with a literal instead of a division on the right requires a different rule. But the candidate can, of course, be applied to (G2.2). The similarity score for this possible reuse pair is 0.0. The candidate is reused, and (G2.2) is replaced by two new subgoals:

$$\begin{aligned}&\Longrightarrow \neg(\mathtt{x}=0) \longrightarrow\\ &\qquad \neg(\mathtt{x}=0) \longrightarrow (\mathtt{res}=div(x,x) \longrightarrow \langle\rangle\,(\mathtt{res}=1))\end{aligned} \tag{G2.2.1}$$

$$\begin{aligned}&\Longrightarrow \neg(\mathtt{x}=0) \longrightarrow\\ &\qquad \mathtt{x}=0 \longrightarrow \langle\mathtt{throw\ new\ ArithmeticException();}\rangle\,(\mathtt{res}=1)\end{aligned} \tag{G2.2.2}$$

We now have three open goals: (G2.1) is on the "new" branch, (G2.2.1) is on the "normal execution" branch, and (G2.2.2) is on the "division by zero" branch. Things get a bit complicated now as we also obtain two new reuse candidates. Both are applications of the same rule, namely the first-order logic rule for handling implications; their foci are:

$$\neg(\mathtt{x}=0) \longrightarrow (\{\mathtt{res}=div(x,x)\}\,\langle\rangle\,(\mathtt{res}=1)) \tag{C-N}$$

$$\mathtt{x}=0 \longrightarrow \langle\mathtt{throw\ new\ ArithmeticException();}\rangle\,(\mathtt{res}=1) \tag{C-Z}$$

Each of these two candidates has a possible focus in all three open goals. Thus we obtain six possible reuse pairs, of which in fact only two are appropriate—(C-N) must be reused at (G2.2.1) and (C-Z) at (G2.2.2), not the other way round. We also do not want to waste any of these two candidates on the branch (G2.1), which was not present in the template. The reuse facility computes the following quality scores for the six pairs:

	(C-N)	(C-Z)
(G2.1)	−0.53	−0.81
(G2.2.1)	−**0.35**	−0.77
(G2.2.2)	−0.58	−**0.35**

As desired, the two right possibilities (shown in bold) have the highest similarity scores and are selected for application. Subsequently the candidate markers move on, and the other 4 possible reuse pairs become obsolete.

From here on, reuse can be continued to the successful completion of the proof. If we immediately close the branch under (G2.2.2), which is obviously futile in the new situation, the new proof consists of 45 proof steps, of which 27 have been reused.[6] This is the optimal reuse performance for the given correction. More important than the numbers, though, is the fact that all unaffected parts of the old proofs could be reused completely. For a complicated program, these parts would normally contain non-trivial user interactions (quantifier instantiations, use of lemmas, etc.). Saving these is the main benefit of reuse.

[6] The numbers can vary with the version of the KeY system.

10.7 Other Systems and Related Methods

In this section we give a short survey and comparison of proof reuse-related methods as employed by a number of different verification systems.

Global abstraction methods

An alternative to incremental reuse presented here is global proof abstraction. This broad group of methods attempts to capture the overall gist of whole proofs—at once—and instantiate it for a new problem. Examples are Kolbe and Walther's technique for proving conjectures by induction [Kolbe and Walther, 1994] and the efforts of the Omega Project [Melis and Whittle, 1999]. To our knowledge, this approach has not been successfully applied to verification of object-oriented software. This might be attributed to the fact that the relevant changes in this domain are of local nature.

Constructive methods

Another non-incremental technique for reusing proofs is *constructive reuse*. The constructive approach is to analyze the changes made to the proof goal (i.e., the program to be verified) and their effects, and to use this information to identify and reassemble parts of the template proof into a new one. This approach, however, needs to have exact knowledge of all calculus rules and effects of program changes ("when an if-statement is inserted, an application of the if-rule must be added to the proof and, below that, the proof branches..."). Thus, constructive methods are infeasible for calculi with complex target programming languages (e.g., JAVA) and a large number of rules.

The software verification system KIV [Balser et al., 2000a], for example, contains a constructive proof reuse facility [Reif and Stenzel, 1993]. It works well as the programs that are verified with KIV are written in a simple Pascal-like language, and the KIV calculus has only a comparatively small number of program logic rules.

Replay methods

The simplest incremental reuse method is to just replay the (old) proof script. This works well as long as the information in which the new proof must differ from the old proof is not contained in the (linear) script but can be inferred during rule application. An example for such types of information are the instantiations of schema variables, which are computed by a matching algorithm. Significant changes in proof structure, however, cannot be handled by a simple replay mechanism.

A typical example for this kind of reuse is the replay mechanism of the Isabelle theorem prover [Nipkow et al., 2002]. It is quite powerful as its proof scripts (usually) contain neither variable instantiations nor the foci of rule applications (which are inferred during rule/tactic application according to simple rules). On the other hand, it cannot automatically cope with changes in proof goal ordering or automatically resume reuse after an intermittent failure.

Similarity guided methods

Melis and Schairer pursue another variation of replay [Melis and Schairer, 1998]; this time specifically for reuse of subproofs in the verification of invariants of reactive systems, which are specified using first-order logic. Due to symmetries and redundancies in the state space, such proofs give rise to many similar subproofs.

Melis and Schairer's approach identifies a suitable previously solved subproblem via a similarity measure on first-order formulas and replays the stored subproof straight on.

This method is related to our work as it operates under the assumption that similar situations (proof goals) warrant similar actions (rule applications or subproofs). The similarity assessment though is performed only once, which is justifiable by a simpler setting.

10.8 Reuse in the Verification Cycle

In this section we discuss how proof reuse fulfills a need that goes beyond the basic scenario that we have presented so far.

10.8.1 The Case of a Changed Class Hierarchy

Fixing a bug is the most obvious but not the only reason for re-doing proofs. Unfortunately, every addition or removal of a class in a Java program potentially invalidates all proofs about this program. The problem is that, for two program-related rule schemata of our calculus the particular rule instance depends on the set of classes constituting the program. Using an old instance in the new context may be unsound. The rules in question are:

- the method call rule, which creates an if-cascade simulating dynamic binding and ranging over all possible implementations of a method
- the typeAbstract rule, which implies that a domain element belonging to some abstract type, already belongs to some more specific non-abstract type:

$$\mathsf{typeAbstract}\ \frac{t \sqsubseteq A, t \sqsubseteq B_1 \vee \cdots \vee t \sqsubseteq B_k \Longrightarrow}{t \sqsubseteq A \Longrightarrow} \quad .$$

with $A \in \mathcal{T}_a$ and $B_1, \ldots, B_k$ the direct subtypes of A

The problem lies here with the Java language, and while this situation can be alleviated, it cannot be completely eliminated in a verification tool. In some cases, efficient criteria can establish that the validity of a particular proof is/is not affected by a particular change of the class hierarchy.[7] For example, an instance of the method call rule remains valid if the added class does not override the method in question.

[7] See, for instance, [Roth, 2006] for a detailed discussion.

Nonetheless, the lack of a sufficiently strong module system in Java [Corwin et al., 2003] impedes modular verification and makes every change of the class hierarchy more costly than one would desire.

In general, such changes demand a re-doing of proofs, most of which will stay to a great extent the same. Here reuse can help.

10.8.2 The Case of a Changed Specification

A problem that is a symmetrical variation of the main reuse scenario presented so far is a case of a revised specification. Given a (partial) proof for $\langle p \rangle \phi$ we are trying to construct a proof for $\langle p \rangle \phi'$, where ϕ' is a (slightly) revised version of ϕ. While this case occurs probably just as often as a change of the program, the outlook for reuse is not as optimistic.

Usually, the specification is provided in a high-level language like OCL or JML, which is then translated into Dynamic Logic. A small change of the specification is more likely to produce a significantly different proof obligation. Furthermore, the choice of reuse candidates in the template proof is far from obvious (apart from the root node).

Altogether, it is hardly possible to give a performance prediction, but the procedure might still be helpful in a given case.

10.8.3 The Case of Interactive Proof Search

Complicated proofs almost always require user interaction. Even worse, the quality of the choice required from the user often becomes apparent only much later in the proof. For instance, many proof steps after choosing an induction hypothesis one regularly finds out that it has to be amended for the proof to be successful. In many cases the required change is actually quite simple, like adding a premiss.

In theory, this is not a problem, since the KeY calculus is confluent. Confluence means that there are no dead ends or blind alleys: it is always possible to extend any partial proof to completion if a proof exists at all. In practice this is a small consolation, since the remnants of the old proof attempt clutter the sequents making it impossible to concentrate on the new one.

This way, we are usually stuck with the only choice of performing undo all the way back to the regrettable decision and re-constructing the rest of the proof. Now, it would be tempting to have the ability to edit the proof tree "in place", but this would require some very elaborate presentation. With proof reuse we obtain an alternative solution to the problem.

Here's how it works in practice. If we think that a proof step needs revision, we select this step (node) in the proof tree. From the context menu we select **change this node**. A clone of the current problem instance will be created, with reuse active. Activating reuse will re-enact the existing proof up to the step we wish to change. Then reuse will stop, and we have the possibility to revise our choice at this point. After that, it is possible (if the new situation allows) to reuse the rest of the old attempt in the new setting.

10.8.4 The Case of Redundant Subproblems

Sometimes a verification problem gives rise to several similar subproblems. These may be symmetrical in some sense, or maybe even identical. Having solved one of them it is possible to employ the reuse mechanism to solve the others.

In practice, we identify the root node of the desired template subproof and mark it as a reuse candidate using the contextual menu of the proof tree. The reuse facility then automatically identifies an open goal where this solution may be applicable and attempts to adapt it to the new target in the usual fashion.

10.8.5 The Case of Using Customizable Calculus Modules

Another opportunity for proof reuse arises when using customizable calculus modules. There are several areas of the KeY calculus where the calculus designers provide alternative sets of rules for the user to choose from. These rule sets have different properties and are tailored towards different verification tasks and scenarios. The areas covered by such customizable modules include: null dereferencing checks (on or off), treatment of static initialization (on or off), integer semantics (three different ones) and others. The user of the KeY system can mix and match the rule sets for each verification problem.

Usually, in order to reduce complexity, it is recommended to verify a program with a "simple" calculus version first and then incrementally add assurance by repeating the proof with a more involved calculus setting. In this proof reuse is a real help. We illustrate this using verification of integer manipulation in programs. The approach of choice here is to verify a program using the mathematical integer semantics, and afterwards repeat the proof with the so-called R_{KeY}-semantics.

The rules of R_{KeY}-semantics differ from the mathematical rules by an additional premiss, which is boxed in the following example of an addition rule:

$$\begin{array}{c}\mathsf{assignmentAdditionToUpdateCheckingOF} \\ \boxed{Range_T(se_1),\, Range_T(se_2) \Longrightarrow Range_T(se_1 + se_2)} \\ \Longrightarrow \{var := se_1 + se_2\}\langle \pi\ \omega\rangle\phi \\ \hline \Longrightarrow \langle \pi\ var{=}se_1{+}se_2\,;\ \omega\rangle\phi\end{array}$$

This means that the R_{KeY}-proof has an additional branch for every arithmetical operation considered during the proof.

Once we are satisfied with a proof that uses mathematical integers, we change the integer semantics to the R_{KeY}-based one and reload the problem. The reuse facility creates a single reuse candidate at the root of the template proof. Activating reuse produces a copy of the template with the additional open branches mentioned above. Discharging these branches yields a proof that the program is functionally correct w.r.t. the finite range of JAVA integers. Note that we did not have to engineer any knowledge about the particular structure of the rules or the ordering of the premisses.

The above scenario can also be seen as a benign instance of a more general—and still open—problem, which we discuss in the following section.

10.8.6 The Case of a Changed Proof System

A fact seldomly acknowledged by verification solution providers is that a significant part of the verification cost is due to changes in the verification system itself. If proofs are used as certificates for program correctness, they often have to be maintained over a longer period of time, possibly over many years. For most purposes, it is essential that proofs can be loaded, checked, and manipulated within the verification system during their lifetime. On the other hand, modifications to the proof system itself are to be expected in the meantime.

These modifications are quite frequent and can force users to redo proofs, mostly for two reasons. The first reason is that a critical bug has been fixed in the system and the correctness assertions—while mostly still valid—have to be re-proved with the fixed version. The second reason is that the improved performance and usability of the new version warrants an upgrade. But, of course, every upgrade also has a downside. Old proofs stored on persistent media may have become obsolete and require significant effort to salvage their content. This is a problem for all verification systems that store proofs.

During the years of the development of the KeY system we have encountered numerous changes in the following areas:

1. logic syntax
2. parser/disambiguation
3. formalization of the Java language semantics
4. logical structure of the rules
5. rule execution engine

We briefly discuss the important change classes (3) and (4). Together with Bormer [2007] we have extended the reuse facility to automate translation of proofs between versions of the proof system affected by these changes. The translation mechanism can load a "legacy" proof with the old rule base and simultaneously an identical proof obligation with the new rule base. The system calculates reuse markers from the diff between rule bases. The reuse process then supports efficient porting of the old proof to the new rule base.

Case (3) arises when minor errors in the symbolic execution rules of the KeY calculus have to be fixed. This happens infrequently, but cannot be ruled out, since one can never arrive from an informal specification at a formal one by formal means.[8] The KeY project on regular bases performs the only measure suitable to mitigate this: cross-checking our rules with other formalizations of Java. A recent check of this kind [Trentelman, 2005] has discovered a missing case in our array assignment rule. The erroneous rule and its correction are presented in Figure 10.5. As one can see, the changes are minor and of local nature, lending themselves nicely to similarity-guided proof reuse.

The case (4) is usually not concerned with soundness, but with efficiency. At one point some rules containing a potential case distinction have been reformulated from

[8] For an in-depth discussion of the calculus soundness issue please see Chapter 9.

$$\frac{\begin{array}{c} a = \texttt{null} \Longrightarrow \langle \pi\ \texttt{throw new NPE();}\ \omega\rangle\phi \\ a \neq \texttt{null} \land (i<0 \lor i \geq a\texttt{.length}) \Longrightarrow \langle \pi\ \texttt{throw new AOBE();}\ \omega\rangle\phi \\ a \neq \texttt{null} \land i \geq 0 \land i < a\texttt{.length} \Longrightarrow \{a[i] := val\}\langle \pi\ \omega\rangle\phi \end{array}}{\Longrightarrow \langle \pi\ a[i]\texttt{=}val\ \omega\rangle\phi}$$

$$\frac{\begin{array}{c} a = \texttt{null} \Longrightarrow \langle \pi\ \texttt{throw new NPE();}\ \omega\rangle\phi \\ a \neq \texttt{null} \land (i<0 \lor i \geq a\texttt{.length}) \Longrightarrow \langle \pi\ \texttt{throw new AOBE();}\ \omega\rangle\phi \\ \boxed{a \neq \texttt{null} \land i \geq 0 \land i < a\texttt{.length} \land \neg storable(val, a) \Longrightarrow \langle \pi\ \texttt{throw new ASE();}\ \omega\rangle\phi} \\ a \neq \texttt{null} \land i \geq 0 \land i < a\texttt{.length} \boxed{\land storable(val, a)} \Longrightarrow \{a[i] := val\}\langle \pi\ \omega\rangle\phi \end{array}}{\Longrightarrow \langle \pi\ a[i]\texttt{=}val\ \omega\rangle\phi}$$

Abbreviations: `NPE=NullPointerException`
`AOBE=ArrayIndexOutOfBoundsException`
`ASE=ArrayStoreException`

Figure 10.5. A rule for array assignment: initial and revised version (differences are boxed)

the form splitting the proof (e.g. ifElseSplit) to a form employing a conditional formula (rule ifElse, both rules are given in Section 2.7.2), which has the advantage that one has to reason about the condition only once. Also in this case, proof reuse can enable a smoother transition to the upgraded calculus.

10.9 Conclusion

Practitioners often report that the cost of re-verification is a serious bottleneck in real world formal methods applications [Denney and Fischer, 2005]. We have presented a proof reuse method that works surprisingly well for a broad range of deductive program verification tasks. The method is very flexible and requires no modification even as the calculus is constantly evolving. Also, no knowledge has to be built into the method concerning the effects that a certain program change has on the structure of the correctness proof.

The main reason why the method works is that programs are exceedingly information-rich artifacts, and the KeY calculus preserves this richness with a highly locally deterministic design. First, symbolic execution rules only apply at the foremost, or active, statement of the program, and, second, there is no rule for sequential composition, so active statements do not “multiply”. This way, there are usually only few possible foci for a particular rule to extend a given partial proof.

We have shown that proof reuse has many applications in the verification process beyond the simple scenario presented at first. We have also discussed the biggest remaining challenge: the case when the specification of a system is modified. We have given instructions on using the reuse implementation within the KeY prover.

We are thankful to Bernd Fischer and Dieter Hutter for very helpful comments on an earlier version of this text.

Own Publications

B. Beckert and V. Klebanov. Proof reuse for deductive program verification. In J. Cuellar and Z. Liu, editors, *Proceedings, Software Engineering and Formal Methods (SEFM), Beijing, China*. IEEE Press, 2004. (Cited on page 4.)

B. Beckert and V. Klebanov. A dynamic logic for deductive verification of concurrent programs. In M. Hinchey and T. Margaria, editors, *Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), London, UK*. IEEE Press, 2007a. (Cited on pages 3 and 4.)

B. Beckert and V. Klebanov. Must program verification systems and calculi be verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, pages 34–41, 2006. (Cited on page 4.)

B. Beckert and V. Klebanov. A dynamic logic for deductive verification of concurrent Java programs with condition variables. In *Proceedings, 1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP), Satellite Workshop CONCUR 2007, Lisbon, Portugal*, 2007b. (Cited on page 4.)

B. Beckert, T. Bormer, and V. Klebanov. Reusing proofs when program verification systems are modified. In *Proc. Software Certificate Management Workshop (SoftCeMent), Long Beach, USA*, pages 41–46, 2005. (Cited on page 4.)

B. Beckert, M. Giese, R. Hähnle, V. Klebanov, P. Rümmer, S. Schlager, and P. H. Schmitt. The KeY System 1.0 (deduction component). In F. Pfenning, editor, *Proceedings, International Conference on Automated Deduction, Bremen, Germany*, volume 4603 of *LNCS*. Springer, 2007a. (Cited on page 3.)

B. Beckert, V. Klebanov, and S. Schlager. Dynamic Logic. In B. Beckert, R. Hähnle, and P. H. Schmitt, editors, *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334. Springer-Verlag, 2007b. (Cited on page 3.)

V. Klebanov. Proof reuse. In B. Beckert, R. Hähnle, and P. H. Schmitt, editors, *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334. Springer-Verlag, 2007. (Cited on page 4.)

V. Klebanov. A JMM-faithful non-interference calculus for Java. In *Scientific Engineering of Distributed Java Applications, 4th International Workshop, Proceedings,* *Luxembourg-Kirchberg*, volume 3409 of *LNCS*, pages 101–111. Springer, 2004. (Cited on pages 4 and 51.)

V. Klebanov, P. Rümmer, S. Schlager, and P. H. Schmitt. Verification of JCSP programs. In J. F. Broenink, H. W. Roebbers, J. P. E. Sunter, P. H. Welch, and D. C. Wood, editors, *Communicating Process Architectures (CPA), Proceedings*, pages 203–218. IOS Press, 2005. (Cited on page 54.)

References

M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006. (Cited on pages 53 and 82.)

E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theor. Comp. Sci.*, 331(2–3):251–290, 2005. (Cited on page 51.)

W. Ahrendt, A. Roth, and R. Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In G. Sutcliffe and A. Voronkov, editors, *Proc. 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Montego Bay, Jamaica*, volume 3835 of *LNCS*, pages 412–426. Springer, Dec. 2005b. (Cited on page 130.)

T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 2–13, Linz, Austria, September 2004. IEEE Computer Society. (Cited on page 147.)

V. Arslan, P. Eugster, P. Nienaltowski, and S. Vaucouleur. SCOOP – concurrency made easy. In J. Kohlas, B. Meyer, and A. Schiper, editors, *Dependable Systems: Software, Computing, Networks, Research Results of the DICS Program*, volume 4028 of *Lecture Notes in Computer Science*, pages 82–102. Springer, 2006. (Cited on page 54.)

T. Arts, G. Chugunov, M. Dam, L.-Å. Fredlund, D. Gurov, and T. Noll. A tool for verifying software written in Erlang. *Int. Journal of Software Tools for Technology Transfer*, 4(4):405–420, 2003. (Cited on page 54.)

E. A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10 (1):110–135, 1975. (Cited on page 50.)

D. Aspinall and J. Ševčík. Java Memory Model examples: Good, bad and ugly. In *Proceedings, 1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP), Satellite Workshop CONCUR 2007, Lisbon, Portugal*, 2007. (Cited on pages 50 and 124.)

A. Avižienis, J.-C. Laprie, and B. Randell. Fundamental concepts of dependability. In *Proceedings, 3rd Information Survivability Workshop (ISW'2000)*, pages 7–12, Boston, USA, October 2000. (Cited on pages 120 and 121.)

T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-200-14, Microsoft Research, 2000. (Cited on page 143.)

M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Proc. Fundamental Approaches to Software Engineering, Berlin, Germany*, volume 1783 of *LNCS*, pages 363–366. Springer, 2000a. (Cited on page 150.)

C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The satisfiability modulo theories library (SMT-LIB). At `http://www.smt-lib.org/`. (Cited on page 126.)

B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer, 2001. (Cited on pages 9 and 13.)

B. Beckert and A. Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning, Seattle, USA*, volume 4130 of *LNCS*, pages 266–280. Springer, 2006b. (Cited on pages 24, 43, and 126.)

B. Beckert and S. Schlager. Refinement and retrenchment for programming language data types. *Formal Aspects of Computing*, 17(4):423–442, 2005. (Cited on page 84.)

B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007. (Cited on pages 7, 9, 12, 18, 41, 105, and 123.)

I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Form. Methods Syst. Des.*, 18(2):141–163, 2001. (Cited on page 123.)

H. Boehm. Finalization, threads, and the Java technology-based memory model. In *JavaOne, Technical Session 3281*, 2005. (Cited on page 50.)

T. Bormer. Change management in deductive program verification. Studienarbeit, Fachbereich Informatik, Universität Koblenz-Landau, 2007. (Cited on page 154.)

D. I. A. Cohen. *Basic Techniques of Combinatorial Theory.* John Wiley & Sons, 1978. (Cited on pages 75 and 76.)

B. Cook. Automatically proving concurrent programs correct. In M. Hinchey and T. Margaria, editors, *Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), London, UK*, pages 269–272. IEEE Press, 2007. (Cited on page 2.)

S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978. (Cited on page 24.)

J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. MJ: A rational module system for Java and its applications. In *Procİ8th annual ACM SIGPLAN conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 241–254, Anaheim, California, USA, 2003. ACM Press. (Cited on page 152.)

F. S. de Boer. A sound and complete shared-variable concurrency model for multi-threaded Java programs. In M. M. Bonsangue and E. B. Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume

4468 of *Lecture Notes in Computer Science*, pages 252–268. Springer, 2007. (Cited on page 51.)

G. Delzanno, J.-F. Raskin, and L. V. Begin. Towards the automated verification of multithreaded Java programs. In J.-P. Katoen and P. Stevens, editors, *Proceedings, 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2280 of *LNCS*, pages 173–187. Springer, 2002. (Cited on page 52.)

E. Denney and B. Fischer. Software certification and software certificate management systems. In *Proceedings ASE Workshop on Software Certificate Management (SoftCeMent) 2005, Long Beach, USA*, pages 1–6, 2005. (Cited on pages 2, 120, and 155.)

A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In R. Alur and D. Peled, editors, *Proceedings, 16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004. (Cited on page 130.)

C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267. ACM, 2004. (Cited on page 89.)

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002. (Cited on page 53.)

T. Gedell and R. Hähnle. Automating verification of loops by parallelization. In M. Herrmann, editor, *Proc. Intl. Conf. on Logic for Programming Artificial Intelligence and Reasoning, Pnhom Penh, Cambodia*, LNCS. Springer-Verlag, Oct. 2006. (Cited on page 29.)

G. Gentzen. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. (Cited on page 23.)

E. Giunchiglia, M. Narizzano, and A. Tacchella. The quantified boolean formulas satisfiability library. At `http://www.qbflib.org/`. (Cited on page 126.)

K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. (Cited on page 24.)

B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006. (Cited on page 98.)

J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. (Cited on page 17.)

A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 453–463, 2002b. (Cited on pages 53 and 103.)

D. Harel. *First-Order Dynamic Logic*. Springer, 1979. (Cited on page 24.)

D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984. (Cited on page 9.)

K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2(4):366–381, 2000a. (Cited on page 52.)

M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993. (Cited on page 104.)

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985. (Cited on page 54.)

H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. Gent, H. v. Maaren, and T. Walsh, editors, *Proceedings, SAT 2000*, pages 283–292. IOS Press, 2000. (Cited on page 126.)

H. H. Hoos and T. Stützle. SATLIB—the satisfiability library. At `http://www.satlib.org`. (Cited on page 126.)

Z. Huang, F. van Harmelen, and A. ten Teije. Reasoning with inconsistent ontologies. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 254–259, Edinburgh, Scotland, August 2005. (Cited on pages 124 and 129.)

M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, The Netherlands, 2001a. (Cited on page 131.)

M. Huisman and G. Petri. The Java Memory Model: a formal explanation. In *Proceedings, 1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP), Satellite Workshop CONCUR 2007, Lisbon, Portugal*, 2007. (Cited on pages 50, 97, and 124.)

Isabelle. A generic proof assistant. At `http://isabelle.in.tum.de/`. (Cited on page 126.)

Jacks. Jacks is an automated compiler killing suite. At `http://www.sourceware.org/mauve/jacks.html`. (Cited on pages 87, 123, and 130.)

B. Jacobs and E. Poll. A logic for the Java modeling language JML. In *Proceedings, 4th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 284–299. Springer, 2001c. (Cited on pages 51 and 123.)

B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In Z. Liu and J. He, editors, *8th International Conference on Formal Engineering Methods, ICFEM, Macao, China, Proceedings*, volume 4260 of *LNCS*, pages 420–439. Springer, 2006. (Cited on page 53.)

E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006. (Cited on page 54.)

C. B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, 1981. (Cited on page 51.)

JSR-133. Java memory model and thread specification revision. Website at `http://jcp.org/en/jsr/detail?id=133`. (Cited on page 49.)

R. M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976. (Cited on pages 43 and 50.)

J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19 (7):385–394, July 1976. (Cited on pages 28 and 44.)

T. Kolbe and C. Walther. Reusing proofs. In A. G. Cohn, editor, *Proc. 11th European Conference on Artificial Intelligence, Amsterdam, The Netherlands*, pages 80–84. John Wiley and Sons, 1994. (Cited on page 150.)

O. Kupferman. Sanity checks in formal verification. In *Proc., 17th International Conference on Concurrency Theory*, volume 4137 of *LNCS*, pages 37–51. Springer-Verlag, 2006. (Cited on page 128.)

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979. (Cited on page 49.)

D. Le Berre and L. Simon. The International SAT Competition web page. At `http://www.satcompetition.org/`. (Cited on pages 126 and 130.)

K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6): 281–288, 2005. (Cited on page 34.)

Z. Manna and A. Pnueli. Completing the temporal picture. In *Selected papers of the 16th international colloquium on automata, languages, and programming*, pages 97–130. Elsevier Science Publishers B. V., 1991. (Cited on page 51.)

J. Manson. *The Java Memory Model.* PhD thesis, University of Maryland at College Park, 2004. (Cited on page 49.)

J. Manson, W. Pugh, and S. Adve. The Java Memory Model, 2005a. Journal article manuscript. Available from `http://www.cs.umd.edu/users/jmanson/java/journal.pdf`. (Cited on page 49.)

J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 378–391. ACM, 2005b. (Cited on page 124.)

C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certificationof Java/Java Card programs annotated in JML. *J. Log. Algebr. Program.*, 58(1–2):89–106, 2004. (Cited on pages 51 and 123.)

E. Melis and A. Schairer. Similarities and reuse of proofs in formal software verification. In B. Smyth and P. Cunningham, editors, *Proc. European Workshop on Advances in Case-Based Reasoning (EWCBR), Dublin, Ireland*, volume 1488 of *LNCS*, pages 76–78, 1998. (Cited on page 151.)

E. Melis and J. Whittle. Analogy in inductive theorem proving. *J. Autom. Reason.*, 22 (2):117–147, 1999. (Cited on page 150.)

E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2): 251–266, 1986. (Cited on pages 142 and 146.)

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002. (Cited on pages 126 and 150.)

OEIS A000108. Catalan numbers. In N. J. Sloane, editor, *The On-Line Encyclopedia of Integer Sequences (OEIS)*, 2008. Published electronically at `http://www.research.att.com/~njas/sequences/A000108`. (Cited on page 76.)

OEIS A060854. Multidimensional Catalan numbers. In N. J. Sloane, editor, *The On-Line Encyclopedia of Integer Sequences (OEIS)*, 2008. Published electronically at `http://www.research.att.com/~njas/sequences/A060854`. (Cited on page 76.)

S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976b. (Cited on page 50.)

D. Peleg. Communication in concurrent dynamic logic. *J. Comput. Syst. Sci.*, 35(1): 23–58, 1987a. (Cited on page 52.)

D. Peleg. Concurrent dynamic logic. *J. ACM*, 34(2):450–479, 1987b. (Cited on page 51.)

A. Platzer. An object-oriented dynamic logic with updates. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, September 2004a. (Cited on page 24.)

A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In D. Swierstra, editor, *Proceedings, ESOP '99*, LNCS 1576. Springer, 1999b. (Cited on pages 51 and 123.)

S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. (Cited on page 126.)

T. Raths, J. Otten, and C. Kreitz. The ILTP problem library for intuitionistic logic. At `http://www.cs.uni-potsdam.de/ti/iltp/`. (Cited on page 126.)

T. Raths, J. Otten, and C. Kreitz. The ILTP problem library for intuitionistic logic. *Journal of Automated Reasoning*, 38(1–3):261–271, 2007. (Cited on page 126.)

W. Reif and K. Stenzel. Reuse of proofs in software verification. In R. K. Shyamasundar, editor, *Proc. Foundations of Software Technology and Theoretical Computer Science, Bombay, India*, volume 761 of *LNCS*, pages 284–293. Springer-Verlag, 1993. (Cited on page 150.)

Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276, New York, NY, USA, 2003b. ACM. (Cited on page 52.)

Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. In *Proceedings SoftMC 2003, Workshop on Software Model Checking, ENTCS 89*, 2003c. (Cited on page 68.)

E. Rodríguez, M. B. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP*, LNCS 3586, pages 551–576. Springer, 2005. (Cited on page 103.)

A. Roth. *Specification and Verification of Object-oriented Components*. PhD thesis, Fakultät für Informatik der Universität Karlsruhe, 2006. (Cited on page 151.)

RTCA. Software considerations in airborne systems and equipment certification. Guideline DO-178B, Radio Technical Commission for Aeronautics, 1992. (Cited on page 125.)

B. Sakugawa, E. Cury, and E. T. Yano. Airborne software concerns in civil aviation certification. In C. A. Maziero, J. G. Silva, A. M. S. Andrade, and F. M. de Assis Silva, editors, *Proceedings, Dependable Computing, Second Latin-American Symposium, LADC, Salvador, Brazil*, volume 3747 of *LNCS*, pages 52–60. Springer, 2005. (Cited on page 125.)

D. Schaaf. An extension of the KeY system to support compositional deductive verification of concurrent programs. Studienarbeit, Fachbereich Informatik, Universität Koblenz-Landau, 2008. (Cited on page 104.)

J. Ševčík. *Program Transformations in Weak Memory Models.* PhD thesis, School of Informatics, University of Edinburgh, 2008. (Cited on page 98.)

K. Stenzel. *Verification of Java Card Programs.* PhD thesis, Institut für Informatik, Universität Augsburg, Germany, July 2005. (Cited on pages 127 and 130.)

A. Stump. SMT-COMP: The Satisfiability Modulo Theories Competition. Website at `http://www.smtcomp.org`. (Cited on page 130.)

Sun Microsystems, Inc. *Code Conventions for the Java Programming Language*, 2003. Available at `java.sun.com/docs/codeconv`. (Cited on page 147.)

G. Sutcliffe. Systems Revealed as Unsound by TPTP Testing. List at `http://www.cs.miami.edu/~tptp/TPTP/BustedAsUnsound.html`, a. (Cited on page 131.)

G. Sutcliffe. CASC: The CADE Automated Theorem Proving Competition. Website at `http://www.cs.miami.edu/~tptp/CASC/`, b. (Cited on page 130.)

G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998. (Cited on page 126.)

G. Sutcliffe and C. Suttner. The TPTP problem library for automated theorem proving. At `http://www.cs.miami.edu/~tptp/`. (Cited on page 126.)

K. Trentelman. Proving correctness of Java Card DL taclets using Bali. In B. Aichernig and B. Beckert, editors, *Proc. 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM), Koblenz, Germany*, pages 160–169, 2005. (Cited on pages 130 and 154.)

UK Ministry of Defence. Requirements for safety critical software in defence equipment (part 1: Requirements, part 2: Guidance). Defence Standard 00-55, Issue 1, Directorate of Standardisation, MoD, 1997. (Cited on page 127.)

V. Vafeiadis and M. J. Parkinson. A marriage of Rely/Guarantee and Separation Logic. In L. Caires and V. T. Vasconcelos, editors, *Proceedings 18th International Conference on Concurrency Theory (CONCUR 2007), Lisbon, Portugal*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007. (Cited on page 51.)

D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001a. (Cited on pages 51 and 123.)

D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001b. (Cited on pages 127 and 130.)

P. H. Welch and P. D. Austin. Java Communicating Sequential Processes home page. At `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`. (Cited on page 54.)

P. H. Welch, N. Brown, J. Moores, K. Chalmers, and B. H. C. Sputh. Integrating and extending JCSP. In A. A. McEwan, S. A. Schneider, W. Ifill, and P. H. Welch, editors, *Proceedings, 30th Communicating Process Architectures Conference, CPA 2007, Guildford, Surrey, UK*, volume 65 of *Concurrent Systems Engineering Series*, pages 349–370. IOS Press, 2007. (Cited on page 54.)

E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–40. ACM Press, 2001. (Cited on pages 52 and 68.)

K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN*

2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7–13, 2008, pages 349–361. ACM, 2008. (Cited on pages 51 and 123.)

E. Zermelo. Beweis dass jede Menge wohlgeordnet werden kann. *Mathematische Annalen*, 59:514–516, 1904. (Cited on page 18.)

K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989. (Cited on page 142.)

List of Symbols

a	schema variable for a Java attribute	27
α	typing function, Def. 2.3	12
$args$	schema variable for a Java argument tuple	27
$\longleftrightarrow$	logical equivalence	14
$(*)$	indicates rules from which the sequent context cannot be omitted	26
β_x^d	modification of a variable assignment, Def. 2.12	16
$\perp$	empty type, Def. 2.1	10
(A)	cast to type A, Def. 2.3	12
`[]`	predefined array access function	12
$[\cdot]$	box modality	14
$\langle\cdot\rangle$	diamond modality	14
C	schema variable for class or interface name	27
$f:\_\to\_$	function “declaration”, Def. 2.3	12
$p:\_$	predicate “declaration”, Def. 2.3	12
$v{:}A$	variable “declaration”, Def. 2.3	12
$\mathcal{CU}$	set of consistent semantics updates, Def. 2.14	18
cs	schema variable for a sequence of catch clauses	27
$\mathcal{D}$	domain of a model, Def. 2.11	16
δ	dynamic type function, Def. 2.11	16
$\sqsubseteq$	type predicate, Def. 2.3	12
e	schema variable for a Java expression	27
$=$	equality predicate, Def. 2.3	12
$(f,(..,d_i,..),d)$	semantic update, Def. 2.14	18
$f(..,t_i,..):=t$	function update, Def. 2.6	13
$f:\_\to\_$	function “declaration”, Def. 2.3	12
`for` x; ϕ; u	quantified update, Def. 2.6	14
Formulas	set of Java Card DL formulas, Def. 2.8	14
FSym_{nr}	non-rigid function symbols, Def. 2.3	11
FSym_{r}	rigid function symbols, Def. 2.3	11
$\mathcal{I}(f)$	interpretation of symbol f, Def. 2.11	16

$v{:}A$	variable “declaration”, Def. 2.3	12
$u_1 \,\|\, u_2$	parallel update, Def. 2.6	13
VSym	variable symbols, Def. 2.3	11

Index

Numbers of pages on which notions are defined are typeset in bold face; if a whole section is dedicated to discussing a notion or concept, the page numbers of that section are typeset in italics.

Curriculum Vitae

Contact	Vladimir Klebanov Universität Koblenz-Landau, Universitätsstr. 1, 56070 Koblenz, Germany `vladimir@uni-koblenz.de` • `www.uni-koblenz.de/~vladimir`
Born	27. 09. 1977 in Kharkov, Ukraine
Status	Since 1993 resident in the Federal Republic of Germany. Since 2005 German citizen.

School Education

1984–1993	*Physical-Mathematical Lyceum Nr. 27*, Kharkov, Ukraine.
1994–1996	*Schule Schloß Salem*, Überlingen, Germany. International Baccalaureate.

University Education

10/1996–8/2003	Computing Science, Universität Karlsruhe. Diploma.
05/2001–7/2003	Student research assistant, KeY project, Institute of Logics, Complexity and Deduction Systems, Universität Karlsruhe.
9/2003–8/2009	PhD student and full-time researcher with the KeY project, group of Prof. Bernhard Beckert, Universität Koblenz-Landau, Koblenz.

Scholarships

12/1996–05/2002	Studies supported by grant from *Studienstiftung des deutschen Volkes* (German National Merit Foundation).
6/2006–11/2006	Research term at the Chalmers University of Technology, Göteborg, Sweden, group of Prof. Hähnle, on scholarship from the German Academic Exchange Service *DAAD*.