Bernhard Beckert
Reiner Hähnle
Peter H. Schmitt (Ec

Verificatio
of Object-(
Software

The KeY Appro

Foreword by K. Rusta

Bernhard Beckert   Reiner Hähnle
Peter H. Schmitt (Eds.)

Verification of Object-Oriented Software

The KeY Approach

Foreword by K. Rustan M. Leino

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Volume Editors

Bernhard Beckert
University of Koblenz
Department of Computer Science, AI Research Group
Universitätsstr. 1, 56070 Koblenz, Germany
E-mail: beckert@uni-koblenz.de

Reiner Hähnle
Chalmers University of Technology
Department Computer Science and Engineering
41296 Göteborg, Sweden
E-mail: reiner@cs.chalmers.se

Peter H. Schmitt
Universität Karlsruhe
Institut für Theoretische Informatik
Am Fasanengarten 5, 76131 Karlsruhe, Germany
E-mail: pschmitt@ira.uka.de

Books must follow sciences, and not sciences books
— Francis Bacon, *Proposition touching Amendment of Laws*

Foreword

Long gone are the days when program verification was a task carried out merely by hand with paper and pen. For one, we are increasingly interested in proving actual program artifacts, not just abstractions thereof or core algorithms. The programs we want to verify today are thus longer, including whole classes and modules. As we consider larger programs, the number of cases to be considered in a proof increases. The creative and insightful parts of a proof can easily be lost in scores of mundane cases.

Another problem with paper-and-pen proofs is that the features of the programming languages we employ in these programs are plentiful, including object-oriented organizations of data, facilities for specifying different control flow for rare situations, constructs for iterating over the elements of a collection, and the grouping together of operations into atomic transactions. These language features were designed to facilitate simpler and more natural encodings of programs, and ideally they are accompanied by simpler proof rules. But the variety and increased number of these features make it harder to remember all that needs to be proved about their uses.

As a third problem, we have come to expect a higher degree of rigor from our proofs. A proof carried out or replayed by a machine somehow gets more credibility than one that requires human intellect to understand.

What it then comes down to is mechanical tools: tools that manage the details for us, tools that support the kinds of specifications we want to write, tools that understand the semantics of the programming language, tools that only allow logically valid proof steps, tools that automate as many of the proof steps as possible, tools that fit into the development environments that programmers use.

The KeY tool addresses all of these demanding desiderata. It targets the modern programming language JAVA, supports multiple common specification notations, and integrates into two popular programmers' development environments. It offers rules for reasoning about common programming idioms, like the use of frame conditions. And its proof engine empowers users with the mathematical ingredients we most commonly need when establish-

ing the correctness of programs, like induction. The advent of the KeY tool constitutes a major scientific advance.

Yet, program verification tools have not reached the same kind of maturity as, say, compilers. It took many years of developing and refining the theory underlying modern compilers, including context-free grammars and data-flow analyses, but these are now taught in undergraduate computer science curricula. We can only hope that program verifiers will eventually become as well understood.

In addition to the engineering effort required to build a program verification tool, building and using such a tool require many skills from computer science, logic, and mathematics. To advance the field of program verification, we must facilitate the acquisition of these skills for our colleagues, students, and other tool builders. This kind of material is typically made available in research papers, but there's nothing like collecting it under one roof. Moreover, there are deviations from some published practices that our experience reveals to be useful. Combine these with some case studies that show the application of the techniques through a verification tool and you provide a taste of the whole picture.

That is what this KeY book does.

The ultimate goal of program verification is not the theory behind the tools or the tools themselves, but the application of the theory and tools in the software engineering process. Our society relies on the correctness of a vast and growing amount of software. Improving the software engineering process is an important, long-term goal with many steps. Two of those steps are the KeY tool and this KeY book.

Redmond, Washington   K. Rustan M. Leino
September 2006

Preface

The need for verified software has always been around and, since the late 1960s, also the vision of tools to satisfy it [King, 1969]. But it took a long time for the field of software verification to move from foundations of computing in the direction of an engineering-oriented approach. This path was amply accompanied by derisive comments from practitioners. However, in the last decade the prospects of formal software verification technology dramatically improved and many now feel that verification is one of the most exciting and promising areas of computer science to currently work in. The two developments that are mainly responsible for this can be roughly identified with the key words (a) *scope* and *scalability*, and (b) *integration*.

Scope and Scalability

Contemporary verification methods are way beyond academic languages and problems: target programming languages are mainly JAVA [Burdy et al., 2003, Ahrendt et al., 2005a, Stenzel, 2004, Marché and Rousset, 2006], C# [Barnett et al., 2005], as well as C [Ball et al., 2004, Cook et al., 2006], and not merely small fragments are covered, but most or all of these languages. Formal specification and verification of object-oriented industrial software of considerable complexity have become routine in the research domain [Jacobs et al., 2004, Bubel and Hähnle, 2005, Mostowski, 2005, Schellhorn et al., 2006]. Verification-based tools for bug-finding in drivers and system software written in C became available recently [Ball et al., 2004, Cook et al., 2006].

One of the pioneers of the field, C.A.R. Hoare, suggested that formal verification is mature enough to embark on building a "routinely usable Program Verifier" as an international Grand Challenge for the Computer Science community [Hoare, 2003, 2006].

Integration

At the same time, the formal verification community has realized that verification cannot be done in isolation from other software validation methods. It will not replace traditional software engineering techniques and quality assurance methods, but complement them. The current vision is that at some

point formal verification becomes a standard part of the tool portfolio of the software engineer. This requires the integration of verification into processes, development tools, as well as into safety and security policies.

A closely related development is that technologies pioneered in verification and theorem proving are more and more used within scenarios that go beyond verification, including debugging [Flanagan et al., 2002, Barnett et al., 2005], test generation [Tillmann and Schulte, 2005], fault analysis [Ortmeier et al., 2005], or fault injection [Larsson and Hähnle, 2006]. A further encouraging trend is the convergence of tools and methods developed in the formal methods and programming languages communities [Barthe et al., 2004, Darvas et al., 2005, Gedell and Hähnle, 2006, Müller et al., 2006].

The Concept Behind This Book

One area that has yet to catch up with the new spirit of software verification is that of books. Most books on foundations of formal specification and verification orient their presentation along traditional lines in logic. This results in a gap between the foundations of verification and its application that is too wide for most readers. There are, of course, a number of good books on particular verification methods [Boyer and Moore, 1988, Abrial, 1996, Nipkow et al., 2002, Holzmann, 2003], but it is difficult to extract general material from them. A book that aims to go in the right direction has been written by Huth and Ryan [2004]: it presents the logical foundations of specification and verification in as much as they are required by those who want to verify systems, in particular, for verification based on model checking. The present book goes in a similar direction, but with different emphases:

- The material is presented on an advanced level suitable for graduate courses (and, of course, active researchers with an interest in verification).
- The underlying verification paradigm is deductive verification in an expressive program logic.
- As a rule, the proofs of theoretical results are not contained here, but we give pointers where to find them.
- The logic used for reasoning about programs is not a minimalist version suitable for theoretical investigations, but an industrial-strength version. The first-order part is equipped with a type system for modelling of object hierarchies, with underspecification, and with various built-in theories. The program logic covers full JAVA CARD (plus a bit more such as multi-dimensional arrays, characters, and long integers).
- Much emphasis is placed on specification, including two widely used object-oriented specifications languages (OCL and JML) and even an interface to natural language generation. The generation of proof obligations from specified code is discussed at length.
- Two substantial case studies are included and presented in detail.

Nevertheless, we cannot and do not claim to have fully covered formal reasoning about (object-oriented) software in this book. One reason is that the choice of topics is dependent on our research agenda. As a consequence, several important themes, such as specification refinement, model checking, or predicate abstraction, are not covered. In addition, there are topics that we are working on, but we felt that we have not yet reached a sufficient stage of maturity for their inclusion. These include the integration of static analyses with deductive verification, verification of parallel programs, generation of counter examples, proof visualization, verification of recursive programs, modular verification, and test generation, to name just a few.

Background: The KeY Project

The context for this book is the KeY project (`www.key-project.org`), which aims to create a formal methods tool that integrates design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. The project was started in November 1998 at the University of Karlsruhe. It is now a joint project of the University of Karlsruhe, Chalmers University of Technology in Göteborg, and the University of Koblenz-Landau.

Besides theoretical advances that are documented in numerous research papers, the most visible result of the KeY project is the KeY tool, a verification tool that is unique in several ways:

- It is the only publicly available theorem prover that supports the full JAVA CARD language standard including the memory model (with persistent and transient memory) and atomic transactions.
- Specifications can be written in OCL, JML, and in program logic. OCL specifications can even be translated into natural language.
- Plugins to the popular Eclipse IDE and to Borland's Together CASE tool suite are provided as well as stand-alone versions.

The KeY tool is freely available to anyone who is interested. It can be downloaded from the Web site mentioned above.

As we use the tool in several courses we wanted to collect the theory that is necessary for a thorough understanding of the workings of the KeY tool. This was the starting point for the present book. In the beginning we hoped to build on existing overview articles or books, for example, on first-order and program logics, but we soon realized that the treatment of logical foundations there was too idealistic for specification and verification in a realistic setting.

Hence, we decided to make a serious effort to document the knowledge we gained during the course of the project about first-order and program logics for object-oriented programs, about the high-level specification languages OCL and JML, and about how to formulate proof obligations over JAVA CARD programs and then verify them. We make this knowledge available to the research community and to interested students in this book. Even though it is

grounded in a particular project, we think that most parts of the book are interesting in their own right to those working on object-oriented specification and verification. The fact that all examples can be tried out with a concrete system should be seen as a plus, not as a limitation to a particular verifier. More details on the organization and on suggested reading strategies can be found in Chapter 1.

Companion Web Site

This book has its own Web site at `www.key-project.org/thebook`, where additional material is provided: most importantly, the version of the KeY tool that was used to run all the examples in the book including all source files for example programs and specifications (unless excluded for copyright reasons), various teaching materials such as slides and exercises, and the electronic versions of papers on KeY.

Second Readers

Most chapters of the book have been read by experts in the formal verification community. We benefited immensely from their feedback that included many valuable suggestions. The second readers were:

Chapter 2	Temur Kutsia (RISC), Arild Waaler (Univ. Oslo)
Chapter 4	Yves Bertot (INRIA), Lawrence Paulson (Univ. Cambridge)
Chapter 5	Thomas Baar (EPF Lausanne), Gary Leavens (Iowa State Univ.), Erik Poll (RU Nijmegen), Steffen Zschaler (TU Dresden)
Chapter 7	Aarne Ranta (Chalmers Univ.)
Chapter 9	Engelbert Hubbers (RU Nijmegen)
Chapter 12	Richard Banach (Univ. Manchester)
Chapter 13	Bernd Fischer (NASA), Dieter Hutter (DFKI)
Chapter 14	Jean-Louis Lanet (Gemplus), Renaud Marlet (INRIA), Martijn Warnier (VU Amsterdam)

Sidebars and Typographic Conventions

We use a number of typesetting conventions to give the text a clearer structure. Occasionally, we felt that a historical remark, a digression, or a reference to material outside the scope of this book is required. In order not to interrupt the text flow we use *sidebars*, such as on page 30, whenever this is the case.

In this book a considerable number of specification and programming languages are referred to and used for illustration. To avoid confusion we usually typeset multiline expressions from concrete languages in a special environment that is set apart from the main text with horizontal lines and that specifies the source language as, for example, in (1.1) on page 9.

Expressions from concrete languages are written in typewriter font with keywords highlighted in boldface, the exception being UML class and feature names. These are set in sans serif, unless class names correspond to JAVA types. Mathematical meta symbols are set in math font and the rule names of logical calculi in sans serif.

Acknowledgements

Koblenz,	Bernhard Beckert
Gothenburg,	Reiner Hähnle
Karlsruhe,	Peter H. Schmitt
October 2006	

Contents

Part III Using the KeY System

10   Using KeY

11   Proving by Induction

List of Tables

List of Figures

1

A New Look at Formal Methods for Software Construction

by

Reiner Hähnle

This chapter sets the stage. We take stock of formal methods for software construction and sketch a path along which formal methods can be brought into mainstream applications. In addition, we provide an overview of the material covered in this book, so that the reader may make optimal use of it.

1.1 What KeY Is

The KeY project[1] was conceived because, after having worked in logic and theorem proving for many years, we became convinced that a different kind of tool than the existing range of editors and theorem provers is necessary to push formal methods further into industrial applications. We know that not everyone agrees that formal methods have a place in the software industry, but recent success stories, such as the SDV project at Microsoft [Ball et al., 2004], are indicators that formal methods can become mainstream provided that they are appropriately packaged and marketed. We think that formal methods are robust and powerful enough for applications, but they need to become (much!) more accessible.

With this in mind, the KeY system was not designed merely as a theorem prover for verification of object-oriented (OO) software, but as a formal methods tool that integrates design, implementation, formal specification and formal verification as seamlessly as possible. The intention is to provide a platform that allows close collaboration of conventional and formal software development methods.

This sounds as if KeY were a silver bullet. So let us be very clear that we do not think that formal specification and verification of complex systems is a task that can be done automatically or by people who are completely unskilled in formal methods. This is as improbable as automatised programming of complex systems. Everyone accepts that specialists are needed to

---

[1] www.key-project.org

write, say, reliable and efficient systems software. If complex software is to be formally specified and verified, it should be clear that some serious work by specialists is called for. But if formal methods specialists are still required for complex tasks, what is gained by KeY then? In a nutshell, the intention is to lower the cost of formal methods to an acceptable level from where it is clear that formal methods actually will save cost in the end. In the following, we map out the basic principles of such a conception of formal methods in software construction.

Easy Things Made Easy
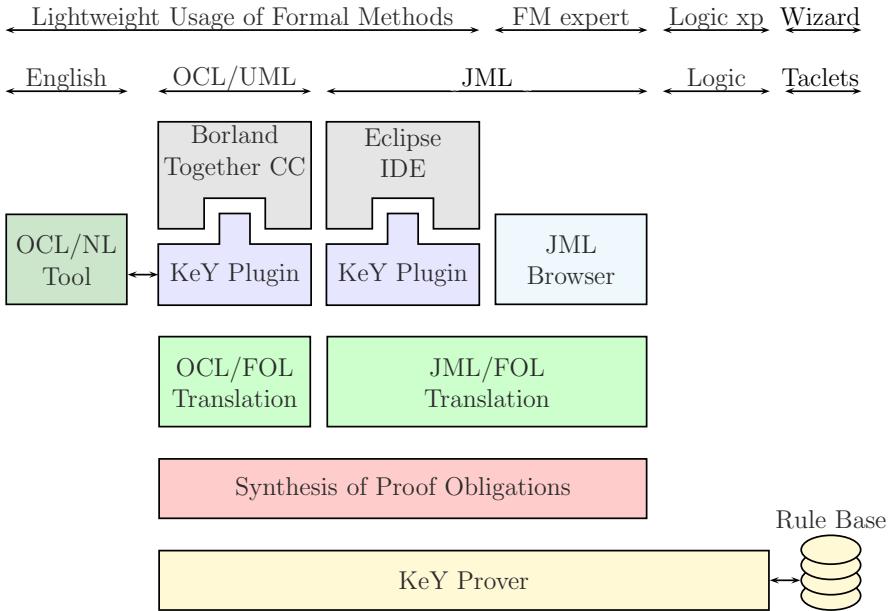
KeY provides interfaces and tools that enable non-specialists in formal methods to use and understand formal artifacts *to a certain extent*. For example, we provide idioms and patterns that can be simply instantiated to create formal specifications. This is comparable to using a visual editor in order to create a JAVA GUI instead of having to master the Swing framework. Developers can also run standard checks, such as the consistency of existing formal specifications by menu selection from their usual case tool. Such provisions push the boundary beyond which a formal methods specialist is required. It also provides a learning path to formal methods for interested developers.

Integration of Informal and Formal Notation

From the view of the non-expert user, KeY appears not as a stand-alone tool, but as a plugin to a familiar case tool (at the moment Borland Together and the Eclipse IDE are supported). Translation of specifications written in UML's Object Constraint Language (OCL) and the Java Modeling Language (JML) into logic, as well as synthesis of various proof obligations is completely automatic, as is, to a large extent, proof search. In addition, KeY features a syntax-directed editor for OCL that can render OCL expressions in several natural languages while they are being edited. It is even possible to translate OCL expressions automatically into English and German (stylistically perhaps not optimal, but certainly readable). This means that KeY provides a common tool and conceptual base for developers and formal methods specialists. The architecture and interface characteristics of KeY are depicted in Fig. 1.1.

Teaching Formal Methods for Software Construction

We think that it is necessary to change the way formal methods are taught. Many of us used to teach traditional courses in logic, theorem proving, formal languages, formal specification, etc. Ten years ago, in a typical Computer Science programme at a European university you could find a wide variety of such courses with at least logic or formal systems courses being compulsory. While such courses are still taught in theoretical specialisations, compulsory logic or formal specification courses have mostly been scrapped. In the

Lightweight Usage of Formal Methods   FM expert   Logic xp   Wizard

English   OCL/UML   JML   Logic   Taclets

Borland Together CC	Eclipse IDE		

OCL/NL Tool	KeY Plugin	KeY Plugin	JML Browser

OCL/FOL Translation	JML/FOL Translation

Synthesis of Proof Obligations

Rule Base

KeY Prover

Fig. 1.1. Architecture and interface of the KeY system

post-Bologna bachelor programmes there will be little room for foundational courses. Even if this were not so, we think that it would be time to look at software construction not as an afterthought in formal methods courses, but as the starting point and main driver for the curriculum. The goal of such a *formal methods for software engineering* course is not only to teach formal specification and verification in the context of OO software development, but also exactly those topics in logic, semantics, formal specification, theorem proving that are necessary for a deepened understanding. Such a presentation of this material would necessarily be less systematic and complete than if it were taught in a traditional manner, but we think this is far outweighed by a number of advantages:

- It is notoriously difficult to motivate students (in particular those interested in software development) to theoretical studies, which are often perceived as useless. The tight integration of formal methods into software development provides a strong and direct motivation.
- Many students find it easier to grasp theoretical concepts when these are explained and motivated with natural examples.
- We often encountered students who, even after taking several foundational courses, perceived, for example, logic and programming language semantics as completely different topics and failed to see their close connections. Compartmentalisation is increased by presentations based on traditional

notations developed in separate fields. It is important to point out similarities and identical concepts. Most of all, it is important to relate to concepts from programming languages, because this is what students of computer science or software engineering are most familiar with. To take a trivial example, students often find it easier to grasp universal quantification when the analogies to for-loops are pointed out, including declaration of index variables, scoping, binding, hiding, etc.

- A course that teaches base knowledge in logic, specification, and semantics under the umbrella of high-quality software construction is much easier to integrate into an educational programme than dedicated foundational courses. The latter tend to be optional and are taken only by a small minority of interested students. We see a great danger, in particular with respect to bachelor programmes, that students are completely deprived of foundations. We believe that an attractively packaged course with foundational material tailored to the requirements of software engineering could be a solution.

A course along the lines just sketched is taught by the author of this chapter at Chalmers University since 2004.[2] Many chapters in this book are suitable as background material for (advanced) courses related to logic, specification, and verification (see also the following section).

Towards Formal Verification as a Debugging Tool

Formal verification is unlikely to be a fully automatic procedure in the foreseeable future. This is true even for less demanding tasks than full functional verification of concrete source code: the availability of so-called push button tools notwithstanding, verification remains a highly interactive process. The main problem, of course, is that most of the time the specification or the implementation (or both) are buggy. Hence, proof attempts are doomed to fail. In software verification, it is also often necessary to strengthen induction hypotheses or invariants before they can be proven. In either case, the source of a failed proof must be located and patched. Then the proof must be retried, etc. This means that it must be possible to inspect a partial or stuck proof and make sense of it. This process has strong similarities to debugging. Therefore, it is important to equip the user interface of a prover with similar capabilities than that of a debugger.

While the debugger view has not quite been realised yet for the KeY prover, which also can be used stand-alone without any CASE tool (see Fig. 1.1), the system offers a wide variety of visual aids and controls. These range from highlighting of active parts in proofs and proof nodes, drag-and-drop application of rules, tool tips with explanations of logical rules to execution control with local computations, breakpoints, etc. Automatic reuse of

---

[2] The course web site is http://www.cs.chalmers.se/Cs/Grundutb/Kurser/form. The LaTeX sources of slides, labs, exercises and exams are available on request.

failed proofs and correctness management of open goals and lemmas round off the picture.

In order to increase automation, a number of predefined search strategies are available. There is a back end to SMT-LIB syntax[3] for proving near propositional proof goals with external decision procedures. A back end to TPTP syntax[4] is under construction.

Nevertheless, it is in this area, where the book gives only a snapshot of the current capabilities. Ongoing research that is hoped to boost interactive proof construction dramatically, includes proof visualisation [Baum, 2006] and automatic search for finite counter examples [Rümmer, 2005].

Industrially Relevant Languages

In our opinion it is essential to support an industrially relevant programming language as the verification target. We have chosen JAVA CARD source code [Chen, 2000] because of its importance for security-critical applications. We refrained from using a home-spun sublanguage of JAVA, because it is unrealistic to assume that applications are written in it. It would have been simpler to create support for JVM bytecode, but while it is easier to build a verification system for byte code than for source code, it becomes more difficult to verify byte code, because it contains much less information. Besides, neither JAVA CARD nor native code compilers produce JVM bytecode.

The KeY prover and calculus support the full JAVA CARD 2.2.1 language. This includes all object-oriented features, atomic transactions, JAVA integer types, abrupt termination (local jumps and exceptions) and even a formal specification (in OCL) of the essential parts of the JAVA CARD API. In addition, some JAVA features that are not part of JAVA CARD are supported as well: multi-dimensional arrays, JAVA class initialisation semantics, `char` and `String` types. In short, if you have a sequential JAVA program without dynamic class loading and floating point types, then it is (in principle) possible to verify it with KeY.

On the front end, we support the OMG standard Object Constraint Language (OCL) [Warmer and Kleppe, 2003] for specification as well as the Java Modeling Language (JML) [Leavens et al., 2006], which is increasingly used in industrial contexts [Burdy et al., 2005].

The KeY system is written in JAVA and runs on all usual architectures. The same is true for the Borland Together and Eclipse CASE tools. Everything, with the exception of Borland Together, is freely available, open software.

---

[3] http://combination.cs.uiowa.edu/smtlib/
[4] http://www.cs.miami.edu/~tptp/

1.2 About This Book

This book is mainly written for two kinds of readers: first, as explained in the previous section, it can serve as a textbook in an advanced formal-methods course. All, but the most basic, required mathematical notions are contained and explained. Chapters 2 and 3 are self-contained discussions of first-order and program logics and calculi *tailored to the needs of formal analysis of OO software.* This means, for example, that meta-logical results such as incompleteness are only fleetingly discussed, as far as necessary to explain the limitations of first-order program logics. Among the many calculi available for automated reasoning we concentrate on sequent calculi, because they are most widely used in deductive software verification. On the other hand, we introduce a richly typed first-order logic including essential notions for program analysis such as rigid/flexible terms, none of which is treated in logic textbooks.

We assume that readers of this book are familiar with object-oriented design and software development, including UML class diagrams and the programming language Java. As mentioned above, no special mathematical knowledge is required, with the exception of basic set theory and propositional logic. Naturally, we do not deny that a certain mathematical maturity is helpful to obtain a deepened understanding of the material in Part I.

Although the book is about a specific tool, the KeY tool, much of the material can be read independently and is transferable to other contexts. The book becomes more KeY-specific towards the end, more precisely, Parts I and II are fairly independent of KeY while Parts III and IV contain specific solutions and case studies. As a consequence, in a course on software development with formal methods, one might stick to the first two parts plus Chapter 11. In the book we proceed in a bottom-up style to avoid dangling definitions, but in the context of a course the material might well be presented top-down (as it is, in fact, done in the course mentioned above). Chapter 10 is an informal introduction to the main features of the KeY tool and can be recommended as an entry point.

The second kind of reader is any kind of computer professional (developer, researcher, etc.) who is interested in formal methods for software development or in KeY in particular. There is no need to read the book sequentially or in any particular order. The chapters can be read fairly independently (but following some dependencies is unavoidable, if a full grasp on technical details is desired). Those who are familiar with formal techniques would only skim Chapter 2, but will find Chapter 3 still interesting. If you are mainly interested in usage and capabilities, Chapter 10 plus some of the case studies are a good start.

Finally, two things that this book is *not*: it is not a reference manual for the KeY system. Even though many features are explained and discussed, for the sake of readability we did not strive for completeness. Some parts of the manual are available online at the KeY website, for example, a browsable

list of all calculus rules. This book is also not simply a collection of papers. All chapters were specifically written for this book and not just culled from a technical report. We aimed at self-containedness, many examples and not too terse explanations. For this we sacrifice some of the technical details. Wherever technical explanations have been abridged or simplified, we give pointers to the full treatments in papers and theses.

In the remainder of this chapter, we explain some of the problems that must be solved during formal specification and verification of object-oriented software. The idea is to give the reader a better idea of what is covered in various parts of the book. It will also help not to loose sight of the big picture.

1.3 The Case for Formalisation

For the following considerations we use a small example. We stress that the size of this example is *not* indicative for the problems that can be modelled with the KeY system. Realistic case studies are discussed in Chapters 14 and 15.

Assume that we are given the following informal specification for developing a simple electronic paycard application:

> "The function of a paycard is to let its owner pay bills or withdraw money from terminals authorised by the card provider.
>
> A paycard contains information about its current balance. The balance must not be negative and must not exceed a given limit. The limit of a paycard cannot be changed, although different cards can have different limits.
>
> Each paycard provides a charge operation that updates the balance according to the amount involved in a transaction."

This specification seems quite precise, but even in a small example like this, unclear issues appear immediately. For example, it is not specified what happens when the charge operation is called with an amount that would exceed the card limit. One of the main advantages of formalisation is to exhibit such imprecision.

Imprecise specifications can lead to serious problems that are detected (too) late, when it is expensive to fix them. In the example, without a clear guideline, an implementor might write a charge method that simply does nothing when the amount to be charged exceeds the card limit. Such an implementation gives no feedback when something went wrong during the operation. Probably, the application needs to be redesigned. (In JAVA CARD redesigning an application can be problematic, because additional features, such as a counter for unsuccessful charge operations, might not fit into memory.)

In Fig. 1.2, a very simple UML class diagram with a first design based on the specification above is given. Partly it reflects the above requirements by

stipulating instance attributes balance and limit, of which the latter is designated as immutable (indicated by the property {frozen}). But what about the other requirements, for example the legal values of balance being between zero and the value of limit? Of course, one could design a wrapper class that provides a type for the legal values, but this has a number of disadvantages: first, a premature decision on how to implement the balance attribute is taken: for example, if the implementation language is C++, one would probably use a scalar type instead. Second, wrapper classes for primitive datatypes lead to clumsy and inefficient code. Third, one needs different wrappers for different limit values, which leads to various rather complex implementation options. Fourth, it does not ensure that illegal values of the balance do not occur but, at best, that a runtime error or exception occurs once this happens.
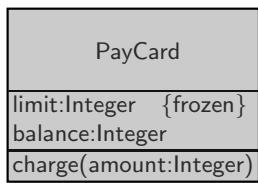
Fig. 1.2. Simple PayCard class diagram

The brief discussion above reflects the fact that purely programming-language-based mechanisms such as type systems cannot ensure all kinds of runtime requirements. The same holds, of course, for design languages (like UML) that are even less expressive than programming languages. In order to specify and guarantee runtime requirements the following ingredients are needed (see also Fig. 1.3):

1. A formal specification language that is expressive enough to capture the requirements a design stipulates on an implementation, for example, that it is an invariant of the PayCard class that the values of the balance attribute are between zero and the value of limit.
2. A framework that allows to formally prove that a given implementation satisfies its requirements. This involves a formalisation of an execution model of the programming language in which the verification targets are written, in our case JAVA CARD.

The designers of UML became aware quite early of the need for a formal specification language. It is a part of the UML since version 1.1 and is called *Object Constraint Language* (OCL) [Warmer and Kleppe, 1999b, 2003]. OCL allows to attach invariants to classes of UML diagrams and to specify operation contracts in the form of pre- and postcondition pairs. Being part of the UML, OCL is standardised by the OMG[5]. The consequent visibility of

---

[5] Object Management Group, www.omg.org

OCL ($\Rightarrow$ Sect. 5.2) and its integration into the world of OO software development motivated us to support it as one of the formal specification languages in the KeY tool. Back to our example, the requirement on the admissible values of balance can be expressed as an OCL invariant:
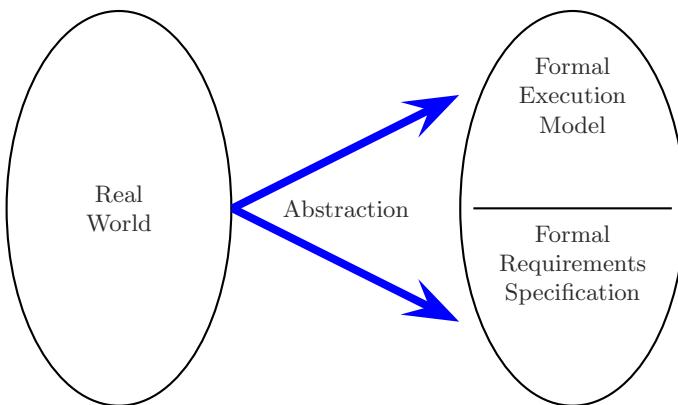
── OCL (1.1) ──────────────────────────────

```
context PayCard
inv withinLimit : balance >= 0 and balance <= limit
```
────────────────────────────────── OCL ──

In order to *prove* that a given implementation of the class PayCard (and possibly other classes) respects this invariant, however, a lot more work is necessary. First of all, if we prove something, there must be some underlying notion of what is a valid statement. In formal logic, as well as in the theory of programming languages, validity is defined in terms of a formal semantics: a mapping between expressions of a formal language and a suitable mathematical model of the underlying domain. In our setting, OCL expressions over a given UML class diagram $\mathcal{D}$ are mapped into an algebra that models objects and object diagrams. As a consequence one can precisely say, for example, that a given object diagram satisfies a given Boolean OCL expression.

Fig. 1.3. The most important ingredients of formalisation

Unfortunately, semantic notions do not lend themselves directly to mechanised proofs, the reason being that semantic domains typically contain infinitely many entities (e.g., all instances of a class). In automated theorem proving, therefore, one reasons over syntactic expressions that represent the semantics adequately. This leads to the notion of a *calculus*, a set of rewrite rules that specify how syntactic expressions are to be manipulated in or-

der to be *derivable*. The idea is to design a calculus that is *sound* with respect to its semantics, that is, all derivable expressions are supposed to be valid. For example, one could conceive of a proof rule that reduces derivability of the invariant `withinLimit` above to derivability of the two invariants `balance >= 0` and `balance <= limit`:

$$\frac{\texttt{context} \quad \mathcal{C} \quad \texttt{inv:}\ \mathcal{I}_1 \qquad\qquad \texttt{context} \quad \mathcal{C} \quad \texttt{inv:}\ \mathcal{I}_2}{\texttt{context} \quad \mathcal{C} \quad \texttt{inv:}\ \mathcal{I}_1 \text{ and } \mathcal{I}_2}$$

Such a syntactic proof rule captures a property of an infinite number of semantic entities: it is valid when $\mathcal{C}$ is replaced by any concrete class name and $\mathcal{I}_1$ and $\mathcal{I}_2$ by any concrete Boolean OCL constraints in any UML diagram.

Although possible, there are good arguments against building such a calculus directly for OCL:

- It is difficult and expensive to develop a theorem prover for a given formal language. OCL is a big language compared to logic languages (such as first-order logic) and, in contrast to them, proof search in OCL is not well understood. Moreover, OCL is frequently revised.
- OCL was not designed with proof support in mind, and like UML it is independent of the implementation language. It does not know about concrete implementations of datatypes such as the integers. Before version 2.0, there was no way to specify initial states of classes. OCL is also not intended to express complex proof obligations that involve several invariants (see below).

As a consequence, we take a "compilation" approach: OCL expressions are translated ($\Rightarrow$ Sect. 5.2) into formulae of first-order logic (FOL). OCL compilation circumvents the difficulties outlined above. It also makes KeY independent from OCL as the sole specification language: recently, JML emerged as a popular specification language used in many formal methods projects dealing with JAVA and JAVA CARD [Burdy et al., 2005]. Replacing the OCL to FOL compiler with a JML front end enables the use of KeY with JML ($\Rightarrow$ Sect. 5.3).

A further major advantage of translating OCL and JML into FOL is that we do not need to define a dedicated formal semantics for these specification languages. Their semantics is implicitly defined by the translation into FOL, the latter having a standard semantics that is widely agreed upon. The translation approach works only if it is natural to represent a specification language by FOL. Admittedly, this is not the case for "vanilla" FOL as encountered in logic textbooks. Object types, undefined expressions, and predefined operators need to be added to the syntax, semantics, and calculus of FOL in order to allow a natural and adequate translation. None of these extensions to FOL is new, but surprisingly no tutorial treatment of this material accessible to non-specialists is available. This justifies Chapter 2 in this book, where we give a self-contained treatment of a FOL tailored to the analysis of object-oriented designs.

1.4 Creating Formal Requirements

Our aim is to develop a formal specification alongside the design. Those requirements that cannot be captured diagrammatically must be formalised either in OCL or in JML and are connected with the design in the form of class invariants or operation contracts. Technically, formal specifications are written as structured comments in JAVADOC style and precede the declaration of the context element they relate to. For example, constraint (1.1) appears in the file `PayCard.java` as follows:

```Java
/**
 * @invariants
 * balance >= 0 and balance < limit
 */
public class PayCard { ... }
```

But before proving that programs fulfill requirements, it is necessary to formalise requirements in the first place. So far, we only know that we can use OCL or JML, but formal requirements specification turns out to be not at all an easy task.
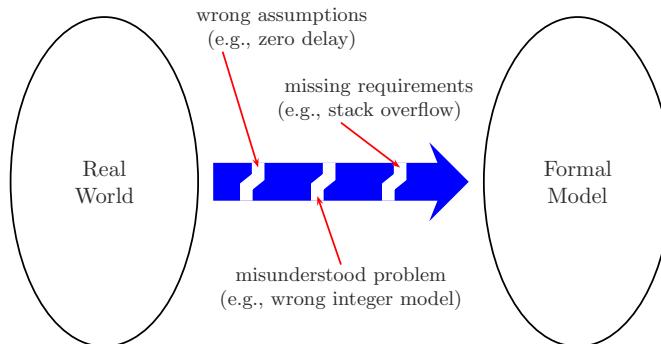
Fig. 1.4. Sources of specification errors

When we formalise any real phenomena, we necessarily have to abstract from the real world (see Fig. 1.3). Many times, this abstraction is the source of errors when specifying requirements. This may involve undue simplifications, missing or misunderstood (and, hence, erroneously modelled) requirements (see Fig. 1.4). Formalisation, as we have seen and will see more below, can detect such errors, but it introduces also additional problems: one needs to master a specification language, but this is not enough—just like software, specifications should be well-crafted. Badly written formal specifications are

at least as difficult to understand and debug as badly written programs. In addition, formal specifications must be well structured and it must be possible to render them in natural language, otherwise informal and formal specifications are in constant need of synchronisation. As a consequence, besides the two capabilities of expressing formal requirements and proving them, it is important to add a third one:

3. Support authoring of formal requirements by providing libraries with idioms and patterns, editors, integration with CASE tools, automatic translation to natural language.

We do not claim to have solved all this, but we address the problems and give partial solutions.

To take a very simple example, imposing upper and lower bounds on a variable with scalar datatype is a very common and typical class invariant, see (1.1). In the KeY system, we call this a *specification idiom*. The KeY extension of Borland Together offers a facility to use a number of specification idioms without the need to know OCL: by filling in forms. More generally, there is an extensible library of design patterns, each of which comes with a number of generic OCL constraints that capture some of the requirements associated with each pattern [Andersson, 2005]. In the hand of an OCL expert this becomes even a flexible extension mechanism for OCL. In Chapter 6 this is discussed further.

In an extensive case study it was shown that at least 25% of the formal specifications could be obtained from standard idioms and very few application-specific patterns [Bubel and Hähnle, 2005]. Although helpful, this leaves a considerable part of the formalisation to be crafted by hand (a similar situation arises in coding, where code generators and templates do not go all the way).

There is some further support in obtaining specifications that can be given. When writing specifications it is useful to keep in mind that there are two important factors driving them:

- Structural properties of the design—for example, the class hierarchy.
- Functional requirements—typically, the state update and returned result that effects from calling a method.

The latter is well-known from the *design-by-contract* methodology [Meyer, 1992], an approximation to full verification. Design-by-contract can be seen as an efficient and elegant alternative (to, for example, dynamic typing) to check requirements at runtime, but it does not prove that the requirements actually hold. Still, the contract metaphor is very useful when specifying the functionality of a method. In our context we often call the pre- and postconditions of a method its contract.

One limitation of contracts is that they emphasize the (method-)local view and do not give a clue as to whether a program achieves its purpose as

a whole. Such "global" properties are difficult to define and prove and one needs to know the implementation of all methods. But already the structure of an OO design (that is, its associations and inheritance relation) gives important information on whether contracts and invariants are sufficient.

The two most important OO techniques for implementation by reuse are inheritance and delegation. They have strong consequences for the properties of an implementation and, therefore, need to be reflected in the formal specification. For inheritance, often Liskov's principle [Liskov and Guttag, 2000] is stipulated: it must be possible to replace an object of a class with an object declared in any of its subclasses without breaking the program. From a specification point of view, this means, for example, that invariants of subclasses must be implied by invariants of super classes.

Delegation is perhaps even more important and mostly preferable to inheritance [Gamma et al., 1995], but the consequences for formal specification are rarely made explicit. Assume, for example, that we want a method checkPIN(PIN:int):boolean in our PayCard class. Typically, this method would be implemented elsewhere, say, in class PIN, but in order to minimize the dependencies between PayCard and PIN (and to allow decoration) one might want to implement a delegator method checkPIN(PIN:int):boolean in PayCard. At this point, it is advisable to copy the contract from the implementing to the delegating method.

In summary, Liskov's principle applied to subclassing and delegation yields an important completion of a formal specification that is driven by structural design properties. In KeY it is even possible to prove that a given specification fulfills various aspects of Liskov's principle, see also the following section.

In the end, it is, of course, unavoidable to author a certain amount of formal specifications in OCL or JML by hand. This means that one has to master one of these languages. For a deeper understanding, it is even advisable to know the principles of the translation of OCL and JML into first-order logic. This, together with a concise introduction to the main syntactic elements of these languages, is the content of Chapter 5.

A major problem with formal specification is that formal and informal specifications tend to drift apart over time, because it is very tedious to keep them in sync. On the other hand, it is not sufficient to maintain merely a formal specification, because it cannot easily be communicated to managers or customers. The KeY tool addresses this problem with a feature for translation of OCL into natural language based on the Grammatical Framework [Ranta, 2004]. Example (1.1) is rendered in (stylistically suboptimal, but readily understandable) English as follows:

> "For the class PayCard the following invariant holds:
>   the balance is at least 0 and the balance is less than the limit."

The translation tool, which also features a multi-lingual, syntax-directed editor for OCL, is explained in Chapter 7.

1.5 Proof Obligations

The invariants and contracts present in a design give no immediate clue of what one actually wants to verify. There is a wide range of possible *proof obligations* that can be constructed from invariants, contracts, method implementations, class initialisers, and the class hierarchy as building blocks.

In the previous section, we mentioned already Liskov's principle as one possible property that one might wish to ensure for a given design and formal specification. In the KeY system, for such and other *lightweight design validation properties*, corresponding proof obligations expressed in the KeY program logic can be synthesised automatically from context sensitive menus available in the KeY plugins. Other lightweight properties include *invariant consistency*, *precondition disjointness*, *contract consistency*, and *strong operation contracts*. They are all formally defined and explained in detail in Chapter 8. Let us give an example for the last one. Given the following contract for a `charge` method of class PayCard:

─── OCL ─────────────────────────────────

```
context PayCard::charge(amount: Integer)
post :  balance = balance@pre + amount
```
────────────────────────────────── OCL ───

It is desirable to ensure that after any returned method call, the invariant of its class is restored provided that it held before the call, here (1.1) (see p. 9). This is an essential part of any strategy to ensure that all class invariants hold at all times. We call this property *strong operation contract*.

The KeY prover fails to show this and it is in fact easy to see that the property does not hold, because the sum of `balance` and `amount` may well be greater than `limit`. This gives early feedback on the insufficiency of this particular contract (assuming we want to keep the invariant). Note that we do not need to prove at this time whether the contract is actually respected by `charge`. This can be postponed. In fact, the implementation of `charge` may well be unknown yet.

The contract can be patched either by adding a precondition or by weakening the postcondition. If we go for the former, the result looks like this:

─── OCL (1.2) ───────────────────────────

```
context PayCard::charge(amount: Integer)
pre :   balance + amount < limit and amount >=0
post :  balance = balance@pre + amount
```
────────────────────────────────── OCL ───

Unfortunately, this is not sufficient to establish the strong operation contract property either. The problem is that the attribute `limit` might have been changed by `charge` (there is no problem for the argument `amount` which is immutable according to OCL semantics). One way to proceed is to strengthen

the postcondition with an expression such as `limit=limit@pre`. This becomes tedious in the presence of many attributes. Even worse, `charge` could have destroyed the invariants of *other* classes involving any public attribute in the whole system. The problem to succinctly express what *has not been modified* by an action is known as the *frame problem* in Artificial Intelligence [Shoham, 1987]. To get a handle on it, it is much more efficient to say what *has been modified* and to assume everything else is not. In JML a list of locations that are assignable by a method can be specified. OCL lacks this feature, but can easily be extended. The complete example including an *assignable clause* is:

```
─── OCL ─────────────────────────────────────────────
  context      PayCard::charge(amount: Integer)
  assignable : balance
  pre :        balance + amount < limit and amount >=0
  post :       balance = balance@pre + amount
                                      ─── OCL ───
```

With this information a proof obligation can be synthesised that establishes the strong operation contract property. The key point is that the assignable clause ensures that `limit` is unchanged and this is enough to establish invariant (1.1). Part of the information contained in assignable clauses can sometimes be derived automatically, for example, the {frozen} property in the class diagram Fig. 1.2 suggests to leave out the location `limit` from the locations modifiable by `charge`. Of course, the validity of the assignable clause of a method must be proven for a given implementation.

Even though assignable clauses make proofs much simpler, it is still a problem that *all* class invariants in a system can be potentially affected by *any* method of *any* class. There is much ongoing research to alleviate this problem, for example, program slicing, containment, type-based approaches, etc. Common to all approaches is the idea that suitable statically checkable information about which methods affect which classes can be used to soundly omit most invariants from a proof. Such kind of analyses are indispensable for *modular verification*, because practically all proof obligations make it necessary to include all existing invariants in order to be sound. Modular verification is discussed in Chapter 8.

1.6 Proving Correctness of Programs

Design and coding should be different activities during software construction. In formal verification, this is reflected by the fact that we generate quite different proof obligations for source code verification as compared to design validation in the previous section. The most standard proof obligation is *total correctness* of a method implementation with respect to its contract. Consider the following implementation of the `charge` method:

```java
━━ JAVA ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
public void charge(int amount) {
  if (this.balance + amount >= this.limit) {
    this.unsuccessfulOperations++;
  } else {
    this.balance = this.balance + amount;
  }
}
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ JAVA ━━
```

Total correctness with respect to contract (1.2) means: if `charge` is called in any state satisfying the precondition, then `charge` terminates normally (that is, without throwing an exception) and in its final state the postcondition holds. The assignable clause is not part of this proof obligation, but creates a proof obligation of its own.

Note that the first branch of the conditional does not compromise correctness, because the precondition ensures that this branch is never taken. This shows that both pre- and postconditions of contracts are essential to specify method correctness.

Correctness cannot be expressed in specification languages such as OCL or JML, because it is necessary to logically relate specification expressions and source code in non-trivial ways in order to produce proof obligations. For example, most notions of a proof obligation for total correctness would encompass not merely the preconditions of the contract, but also the class invariant (in fact, *all* class invariants). Clearly, first-order logic is not sufficient to express correctness either—a dedicated program logic is necessary.

The best-known program logic is Hoare logic [Hoare, 1969]. In KeY we use an extension of Hoare logic called *dynamic logic* [Harel et al., 2000]. The main difference is that dynamic logic is syntactically closed under all propositional and first-order operators. The advantage is increased expressiveness: one can express not merely program correctness, but also security properties [Mostowski, 2005], correctness of program transformations, or the validity of assignable clauses. Other verification approaches [Paulson, 1994, Boyer, 2003] encode program syntax and semantics in higher-order logic, but this creates considerable overhead, in particular during interactive proving. Dynamic logic, like Hoare logic, works directly on the source code.

The program logic of KeY is called JAVA CARD DL. It has been axiomatised in a sequent calculus and it is relatively complete[6] for any given JAVA CARD program. The actual verification process in KeY can be envisaged as

---

[6] It is well-known that Turing-complete programming languages cannot be completely axiomatised by first-order program logics. As usual, we supply an induction schema to approximate completeness. The axiomatisation is relatively complete to Peano arithmetic. The incompleteness phenomenon is irrelevant for programs that occur in practice.

symbolic execution of source code. Loops and recursion are handled by induction over data structures occurring in the verification target. Alternatively, partial correctness of loops can also be shown by a rule that uses invariants. Symbolic execution plus induction as a verification paradigm was originally suggested for informal usage by Burstall [1974]. The idea to use dynamic logic as a basis for mechanisation of symbolic execution was first realised in the Karlsruhe Interactive Verifier (KIV) tool [Heisel et al., 1987]. Symbolic execution is extremely suitable for interactive verification, because proof progress corresponds to program execution, which makes it easy to interpret intermediate stages in a proof and failed proof attempts.

JAVA CARD is a complex language and this is reflected in the logic JAVA CARD DL. Therefore, in Chapter 3 we break down JAVA CARD DL into several modular components. The core component (⇒ Sect. 3.6) defines symbolic execution rules for JAVA programs with bounded loops and without method calls. Such programs always terminate and it is possible to execute them symbolically in a fully automatic way. The result is the symbolic state update reached after the program terminates (more precisely, a set of updates, each corresponding to one or more execution branches). Updates are applied to first-order postconditions essentially via syntactic substitution, resulting in pure first-order verification conditions dealt with by the first-order rules (⇒ Chap. 2). Further components of the JAVA CARD DL calculus add independent mechanisms for handling loops (⇒ Sect. 3.7) and method calls (⇒ Sect. 3.8). It is essential for efficiency to simplify resulting state updates eagerly after each symbolic execution step. Update simplification is contained in a separate component (⇒ Sect. 3.9).

Full treatment of some JAVA features is so complex that particular chapters have been devoted to them. For example, the `charge` method above is not correct with respect to JAVA integer types, which are finite. How to handle JAVA integer semantics correctly and efficiently is shown in Chapter 12. There is also a dedicated chapter on data structure induction (⇒ Chap. 11).

JAVA CARD has two features that JAVA does not have:

- *Persistent memory* that resides in EEPROM, in addition to standard *transient* memory residing in RAM.
- *Atomic transactions* brace a sequence of statements that are either executed until completion or not executed at all. Transactions are crucial to avoid inconsistent data in the case of interrupted computations caused by card tear-out, power loss, communication failure, etc.

The combination of both features is surprisingly complex to model, because the semantics of transactions treats the two kinds of memory differently. We devote Chapter 9 to the logical modelling of JAVA CARD transactions.

As mentioned above, verification based on the JAVA CARD DL calculus corresponds to symbolic program execution (plus induction). Its rules can, therefore, be seen as an operational semantics for JAVA CARD. As long as neither unbounded loops nor recursion occurs, it is possible to execute programs

symbolically almost without search, even though the full calculus contains several hundred rules. In the case study in Chapter 14 proofs with several ten thousand nodes are automatically constructed in a matter of minutes. At the same time, the KeY prover is very flexible: for example, there is a rule set that axiomatises Gurevich Abstract State Machines [Nanchen et al., 2003], one for a fragment of C [Gladisch, 2006], one for a object-oriented core language called ODL [Platzer, 2004b], and one for simplification of OCL expressions [Giese and Larsson, 2005]. Work on support for MISRA-C 2004[7] source code is in progress.

It is interesting to look at the reasons how it is possible to create and maintain support for such a variety of imperative languages with relatively modest effort. Many interactive theorem provers are implemented in a meta programming language for rule and proof construction. The advantage is generality: not only software verification but any kind of mathematics can be modelled; in addition, not only rules can be described, but also the way how to prove them. In contrast to this, the KeY rules must adhere to a very specific schema language we call *taclet* [Beckert et al., 2004], which is tailored to the needs of interactive verification. Taclets specify not only the logical content of a rule, but also the context and pragmatics of its application. Since taclets have limited reflection capabilities, the set of primitive rules in JAVA CARD DL that have to be considered as axiomatic is relatively large with over a hundred. Soundness of these rules must be shown with external tools [Ahrendt et al., 2005b, Trentelman, 2005]. But the advantages of taclets are enormous: they can be efficiently compiled not only into the rule engine, but also into the automation heuristics and into the GUI. In addition, it is a matter of hours to master the taclet language. In many cases, new taclets can be verified within KeY by reflection. In Chapter 4, the taclet concept as well as correctness of taclets is discussed. An introduction into proof search and the GUI of the KeY prover is found in Chapter 10.

---

[7] http://www.misra-c2.com/

Part I

Foundations

2

First-Order Logic

by
Martin Giese

In this chapter, we introduce a first-order logic. This logic differs in some respects from what is introduced in most logic text books as classical first-order logic. The reason for the differences is that our logic has been designed in such a way that it is convenient for talking about Java programs. In particular our logic includes a type system with subtyping, a feature not found in most presentations of first-order logic.

Not only the logic itself, but also our presentation of it is different from what is found in a logic textbook: We concentrate on the definition of the language and its meaning, motivating most aspects from the intended application, namely the description of Java programs. We give many examples of the concepts introduced, to clarify what the definitions mean. In contrast to an ordinary logic text, we hardly state any theorems about the logic (with the notable exception of Section 2.6), and we prove none of them. The intention of this book is not to teach the reader how to *reason about* logics, but rather how to *use* one particular logic for a particular purpose.

The reader interested in the theoretical background of first-order logic in the context of automated deduction might want to read the book of Fitting [1996], or that of Goubault-Larrecq and Mackie [1997]. There are a number of textbooks covering first-order logic in general: by Ben-Ari [2003], Enderton [2000], Huth and Ryan [2004], Nerode and Shore [1979], or for the mathematically oriented reader Ebbinghaus et al. [1984]. To the best of our knowledge the only textbooks covering many-sorted logic, but not allowing subsorts, are those by Manzano [1996] and Gallier [1986]. For the technical details of the particular logic described in this chapter, see [Giese, 2005].

2.1 Types

We want to define the type system of our logic in a way that makes the logic particularly convenient to reason about objects of the Java programming

language. The type system of the logic therefore matches JAVA's type system in many ways.[1]

Before we define our type system, let us point out an important fact about the concept of types in JAVA.

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*. This static type is computed by the compiler from the literals, variables, methods, attributes, etc. that constitute the expression, using the type information in the declarations of these constituents. The static type is used for instance to determine which declaration an identifier refers to. A variable declaration `C x;` determines the static type `C` of the variable `x` when it occurs in an expression. Via a set of assignment compatibility rules, it also determines which static types are allowed for an expression `e` in an assignment `x = e`. In contrast to dynamic types, static types can also be abstract class types or interface types.

Every possible dynamic type can also occur as a static type. The static types are ordered in a type hierarchy. It therefore makes sense to talk about the dynamic type of an object being a subtype of some static type.

The connection between dynamic types and static types is this: The dynamic type of an object that results from evaluating an expression is always a subtype of the static type of that expression. For variables or attributes declared to be of type C, this means that the dynamic type of their value at runtime is always a subtype of C.

So, does a JAVA object have several types? No, an object has only a dynamic type, and it has exactly one dynamic type. However, an object can be used wherever a static type is required that is a supertype of its dynamic type.

We reflect this distinction in our logic by assigning static types to expressions ("terms") and dynamic types to their values ("domain elements").

We keep the discussion of the logic independent of any particular class library, by introducing the notion of a *type hierarchy*, which groups all the relevant information about the 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 $\mathcal{T}$,

---

[1] It turns out that the resulting logic is reminiscent of Order-Sorted Algebras [Goguen and Meseguer, 1992].

- *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

- $\mathcal{T} = \mathcal{T}_d \,\dot\cup\, \mathcal{T}_a$
- *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* $\mathcal{T}$, *i.e., for all types* $A, B, C \in \mathcal{T}$,

$$A \sqsubseteq A$$
$$\text{if } A \sqsubseteq B \text{ and } B \sqsubseteq A \text{ then } A = B$$
$$\text{if } A \sqsubseteq B \text{ and } B \sqsubseteq C \text{ then } A \sqsubseteq C$$

- $\bot \sqsubseteq A \sqsubseteq \top$ *for all* $A \in \mathcal{T}$.
- $\mathcal{T}$ *is closed under greatest lower bounds w.r.t.* $\sqsubseteq$, *i.e., for any* $A, B \in \mathcal{T}$, *there is an*[2] $I \in \mathcal{T}$ *such that* $I \sqsubseteq A$ *and* $I \sqsubseteq B$ *and for any* $C \in \mathcal{T}$ *such that* $C \sqsubseteq A$ *and* $C \sqsubseteq B$, *it holds that* $C \sqsubseteq I$. *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* $\mathcal{T}_q := \mathcal{T} \setminus \{\bot\}$.

Note 2.2. In JAVA, interface types and abstract class types cannot be instantiated: the dynamic type of an object can never be an interface type or an abstract class type. We reflect this in our logic by dividing the set of types into two partitions:

$$\mathcal{T} = \mathcal{T}_d \,\dot\cup\, \mathcal{T}_a$$

$\mathcal{T}_d$ is the set of possible dynamic types, while $\mathcal{T}_a$ contains the abstract types, that can only occur as static types. The distinction between abstract class types and interface types is not important in this chapter, so we simply call all types that cannot be instantiated abstract.

The empty type $\bot$ is obviously abstract. Moreover, any abstract type that has no subtypes but $\bot$ would necessarily also be empty, so we require some non-abstract type to lie between any non-empty abstract type and the empty type.

Note 2.3. We consider only finite type hierarchies. In practice, any given JAVA program is finite, and can thus mention only finitely many types. The language specification actually defines infinitely many built-in types, namely the nested array types, e.g., `int[]`, `int[][]`, `int[][][]`, etc. Still, even though

---

[2] It is well-known that the greatest lower bound is unique if it exists.

there are conceptually infinitely many types, any reasoning in our system is always in the context of a given fixed program, and only finitely many types are needed in that program.

The reason for restricting the logic to finite type hierarchies is that the construction of a calculus ($\Rightarrow$ Sect. 2.5) becomes problematic in the presence of infinite hierarchies and abstract types. We do not go into the intricate details in this text.

Note 2.4. We do not consider the universal type $\top$ to be abstract, which means that there might be objects that belong to $\top$, but to none of the more specific types. In JAVA this cannot happen: Any value is either of a primitive type or of a reference type, in which case its type is a subtype of `Object`. We can easily forbid objects of dynamic type $\top$ when we apply our logic to JAVA verification. On the other hand, simple explanatory examples that do not require a "real" type hierarchy are more easily formulated if $\top$ and $\bot$ are the only types.

Note 2.5. In JAVA, the primitive types `int`, `boolean`, etc. are conceptually quite different from the class and interface types. We do not need to make this difference explicit in our definition of the logic, at least not until a much later point. For the time being, the important property of an `int` value is that there are indeed values that have the type `int` and no other type at runtime. Hence, `int`, like all other primitive types, belongs to the dynamic, i.e., the non-abstract types.

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.

Example 2.6. Consider the type hierarchy in Fig. 2.1, which is mostly taken from the JAVA Collections Framework. Arrows go from subtypes to supertypes, and abstract types are written in italic letters ($\bot$ is of course also abstract).

In this hierarchy, the following hold:

$$\mathcal{T} = \{\top, \text{Object}, \text{AbstractCollection}, \text{List},$$
$$\text{AbstractList}, \text{ArrayList}, \text{Null}, \text{int}, \bot\}$$

$$\mathcal{T}_q = \{\top, \text{Object}, \text{AbstractCollection}, \text{List}, \text{AbstractList}, \text{ArrayList}, \text{Null}, \text{int}\}$$

$$\mathcal{T}_d = \{\top, \text{Object}, \text{ArrayList}, \text{Null}, \text{int}\}$$

$$\mathcal{T}_a = \{\text{AbstractCollection}, \text{List}, \text{AbstractList}, \bot\}$$

$$\text{int} \sqcap \text{Object} = \bot$$

$$\text{int} \sqcup \text{Object} = \top$$
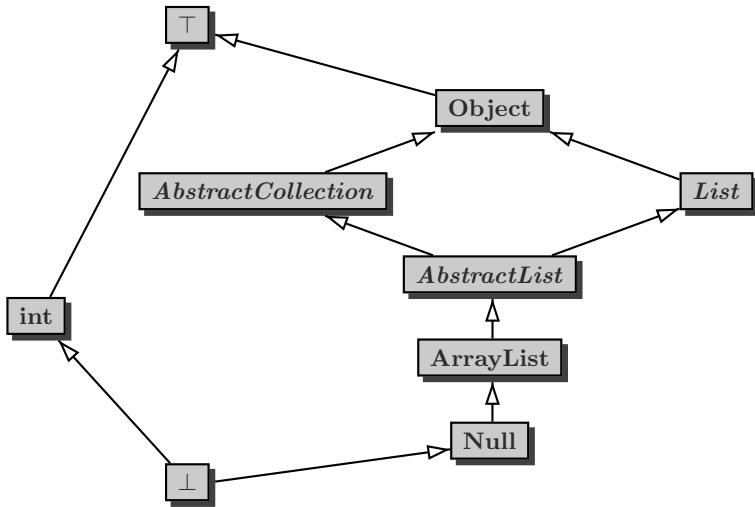
Fig. 2.1. An example type hierarchy

$$\text{AbstractCollection} \sqcap \text{List} = \text{AbstractList}$$

$$\text{AbstractCollection} \sqcup \text{List} = \text{Object}$$

$$\text{Object} \sqcap \text{Null} = \text{Null}$$

$$\text{Object} \sqcup \text{Null} = \text{Object}$$

Example 2.7. Consider the type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ with:

$$\mathcal{T} := \{\top, \bot\}, \quad \mathcal{T}_d := \{\top\}, \quad \mathcal{T}_a := \{\bot\}, \quad \bot \sqsubseteq \top \ .$$

We call this the *minimal type hierarchy*. With this hierarchy, our notions are exactly like those for untyped first-order logic as introduced in other textbooks.

2.2 Signatures

A method in the JAVA programming language can be called, usually with a number of arguments, and it will in general compute a result which it returns. The same idea is present in the form of function or procedure definitions in many other programming languages.

The equivalent concepts in a logic are functions and predicates. A function gives a value depending on a number of arguments. A predicate is either true of false, depending on its arguments. In other words, a predicate is essentially

a Boolean-valued function. But it is customary to consider functions and predicates separately.

In JAVA, every method has a declaration which states its name, the (static) types of the arguments it expects, the (static) type of its return value, and also other information like thrown exceptions, static or final flags, etc. The compiler uses the information in the declaration to determine whether it is legal to call the method with a given list of arguments.[3] All types named in a declaration are static types. At run-time, the dynamic type of any argument may be a subtype of the declared argument type, and the dynamic type of the value returned may also be a subtype of the declared return type.

In our logic, we also fix the static types for the arguments of predicates and functions, as well as the return type of functions. The static types of all variables are also fixed. We call a set of such declarations a *signature*.

The main aspect of JAVA we want to reflect in our logic is its type system. Two constituents of JAVA expressions are particularly tightly linked to the meaning of dynamic and static types: type casts and `instanceof` expressions. A type cast `(A)o` changes the static type of an expression `o`, leaving the value (and therefore the dynamic type) unchanged. The expression `o instanceof A` checks whether the dynamic type of `o` is a subtype of `A`. There are corresponding operations in our logic. But instead of considering them to be special syntactic entities, we treat them like usual function resp. predicate symbols which we require to be present in any signature.

Definition 2.8. *A* signature *(for a given type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$) is a quadruple* $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$ *of*

- *a set of* set of variable symbols VSym,
- *a set of* function symbols FSym,
- *a set of* predicate symbols PSym, *and*
- *a typing function* α,

such that[4]

- $\alpha(v) \in \mathcal{T}_q$ *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:

---

[3] The information is also used to disambiguate calls to overloaded methods, but this is not important here.

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

- $v{:}A$ *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* $A.$

Note 2.9. We require the static types in signatures to be from $\mathcal{T}_q$, which excludes the empty type $\bot$. Declaring, for instance, a variable of the empty type would not be very sensible, since it would mean that the variable may not have any value. In contrast to JAVA, we allow using the Null type in a declaration, since it has the one element null.

Note 2.10. While the syntax $(A)t$ for type casts is the same as in JAVA, we use the syntax $t \sqsubseteq A$ instead of `instanceof` for type predicates. One reason for this is to save space. But the main reason is to remind ourselves that our type predicates have a slightly different semantics from that of the JAVA construct, as we will see in the following sections.

Note 2.11. In JAVA, there are certain restrictions on type casts: a cast to some type can only be applied to expressions of certain other types, otherwise the compiler signals an error. We are less restrictive in this respect, an object of any type may be cast to an object of any other (non-$\bot$) type. A similar observation holds for the type predicates, which may be applied in any situation, whereas JAVA's `instanceof` is subject to certain restrictions.

Note 2.12. We use the symbol $\doteq$ in our logic, to distinguish it from the equality $=$ of the mathematical meta-level. For instance, $t_1 \doteq t_2$ is a formula, while $\phi = (t_1 \doteq t_2)$ is a statement that two formulae are equal.

Like casts, our equality predicate $\doteq$ can be applied to terms of arbitrary types. It should be noted that the KeY system recognises certain cases where the equality is guaranteed not to hold and treats them as syntax errors. In particular, this happens for equalities between different primitive types and between a primitive type and a reference type. In contrast, the JAVA Language Specification also forbids equality between certain pairs of reference types. Both our logic and the implementation in the KeY system allow equalities between arbitrary reference types.

Note 2.13. In our discussion of the logic, we do not allow *overloading*: α gives a unique type to every symbol. This is not a real restriction: instead of an overloaded function f with $f : A \to B$ and $f : C \to D$, one can instead use two functions $f_1 : A \to B$ and $f_2 : C \to D$. Of course, the KeY system allows using overloaded methods in JAVA programs, but these are not represented as overloaded functions in the logic.

Example 2.14. For the type hierarchy from Example 2.6, see Fig. 2.1, a signature may contain:

VSym $= \{n, o, l, a\}$ with n:int, o:Object, l:List, a:ArrayList

FSym $= \{zero, plus, empty, length, (\top), (\text{Object}), (\text{int}), \ldots\}$

with

$$
\begin{array}{ll}
zero : & \text{int} \\
plus : & \text{int, int} \rightarrow \text{int} \\
empty : & \text{List} \\
length : & \text{List} \rightarrow \text{int} \\
(\top) : & \top \rightarrow \top \\
(\text{Object}) : & \top \rightarrow \text{Object} \\
(\text{int}) : & \top \rightarrow \text{int} \\
& \vdots
\end{array}
$$

and

$$
\text{PSym} = \{isEmpty, \doteq, \sqsubseteq\top, \sqsubseteq\text{Object}, \sqsubseteq\text{int}, \ldots\}
$$

with

$$
\begin{array}{ll}
isEmpty : & \text{List} \\
\doteq : & \top, \top \\
\sqsubseteq\top : & \top \\
\sqsubseteq\text{Object} : & \top \\
\sqsubseteq\text{int} : & \top \\
& \vdots
\end{array}
$$

In this example, *zero* and *empty* are constant symbols.

2.3 Terms and Formulae

Where the JAVA programming language has expressions, a logic has *terms* and *formulae*. Terms are composed by applying function symbols to variable and constant symbols.

Definition 2.15. *Given a signature* $(\text{VSym}, \text{FSym}, \text{PSym}, \alpha)$, *we inductively define the system of sets* $\{\text{Trm}_A\}_{A \in \mathcal{T}}$ *of terms of static type A to be the least system of sets such that*

- $x \in \text{Trm}_A$ *for any variable x:$A \in$ VSym,*
- $f(t_1, \ldots, t_n) \in \text{Trm}_A$ *for any function symbol $f : A_1, \ldots, A_n \rightarrow A \in$ FSym, and terms $t_i \in \text{Trm}_{A_i'}$ with $A_i' \sqsubseteq A_i$ for $i = 1, \ldots, n$.*

For type cast terms, we write $(A)t$ instead of $(A)(t)$. We write the static type of t as $\sigma(t) := A$ for any term $t \in \text{Trm}_A$.

A ground term *is a term that does not contain variables.*

Defining terms as the "least system of sets" with this property is just the mathematically precise way of saying that all entities built in the described way are terms, and *no others*.

Example 2.16. With the signature from Example 2.14, the following are terms:

n	a variable
empty	a constant
$plus(n, n)$	a function applied to two subterms
$plus(n, plus(n, n))$	nested function applications
$length(a)$	a function applied to a term of some subtype
$length((\text{List})o)$	a term with a type cast
$(\text{int})o$	a type cast we do not expect to "succeed"

On the other hand, the following are not terms:

$plus(n)$	wrong number of arguments
$length(o)$	wrong type of argument
$isEmpty(a)$	*isEmpty* is a predicate symbol, not a function symbol
$(\bot)n$	a cast to the empty type

Formulae are essentially Boolean-valued terms. They may be composed by applying predicate symbols to terms, but there are also some other ways of constructing formulae. Like with predicate and function symbols, the separation between terms and formulae in logic is more of a convention than a necessity. If one wants to draw a parallel to natural language, one can say that the formulae of a logic correspond to statements in natural language, while the terms correspond to the objects that the statements are about.

Definition 2.17. *We inductively define the set of formulae* Fml *to be the least set such that*

- $p(t_1, \ldots, t_n) \in$ Fml *for any predicate symbol* $p : A_1, \ldots, A_n$ *and terms* $t_i \in \text{Trm}_{A'_i}$ *with* $A'_i \sqsubseteq A_i$ *for* $i = 1, \ldots, n$,
- true, false $\in$ Fml.
- $!\phi, (\phi \mid \psi), (\phi \,\&\, \psi), (\phi \rightarrow \psi) \in$ Fml *for any* $\phi, \psi \in$ Fml.
- $\forall x.\phi, \exists x.\phi \in$ Fml *for any* $\phi \in$ Fml *and any variable* x.

For type predicate formulae, we write $t \sqsubseteq A$ *instead of* $\sqsubseteq A(t)$. *For equalities, we write* $t_1 \doteq t_2$ *instead of* $\doteq (t_1, t_2)$. *An* atomic formula *or* atom *is a formula of the shape* $p(t_1, \ldots, t_n)$ *(including* $t_1 \doteq t_2$ *and* $t \sqsubseteq A$). *A* literal *is an atom or a negated atom* $!p(t_1, \ldots, t_n)$.

We use parentheses to disambiguate formulae. For instance, $(\phi \,\&\, \psi) \mid \xi$ and $\phi \,\&\, (\psi \mid \xi)$ are different formulae.

The intended meaning of the formulae is as follows:

$p(\ldots)$	The property p holds for the given arguments.
$t_1 \doteq t_2$	The values of t_1 and t_2 are equal.
true	always holds.
false	never holds.

$!\phi$	The formula ϕ does not hold.
$\phi \,\&\, \psi$	The formulae ϕ and ψ both hold.
$\phi \mid \psi$	At least one of the formulae ϕ and ψ holds.
$\phi \mathrel{->} \psi$	If ϕ holds, then ψ holds.
$\forall x.\phi$	The formulae ϕ holds for all values of x.
$\exists x.\phi$	The formulae ϕ holds for at least one value of x.

In the next section, we give rigorous definitions that formalise these intended meanings.

KeY System Syntax, Textbook Syntax

The syntax used in this chapter is not exactly that used in the KeY system, mainly to save space and to make formulae easier to read. It is also different from the syntax used in other accounts of first-order logic, because that would make our syntax too different from the ASCII-oriented one actually used in the system. Below, we give the correspondence between the syntax of this chapter, that of the KeY system, and that of a typical introduction to first-order logic.

this chapter	KeY system	logic textbooks
$(A)t$	`(A) `t	—
$t \sqsubseteq A$	`A::contains(`t`)`	—
$t_1 \doteq t_2$	t_1` = `t_2	$t_1 \doteq t_2$, $t_1 \approx t_2$, etc.
true	`true`	T, $t\!\!t$, $\top$, etc.
false	`false`	F, $f\!\!f$, $\bot$, etc.
$!\phi$	`! `ϕ	$\neg\phi$
$\phi \,\&\, \psi$	ϕ` & `ψ	$\phi \wedge \psi$
$\phi \mid \psi$	ϕ`	`ψ	$\phi \vee \psi$
$\phi \mathrel{->} \psi$	ϕ` -> `ψ	$\phi \rightarrow \psi$
$\forall x.\phi$	`\forall A x; `ϕ	$\forall x.\phi$, $(\forall x)\phi$, etc.
$\exists x.\phi$	`\exists A x; `ϕ	$\exists x.\phi$, $(\exists x)\phi$, etc.

The KeY system requires the user to give a type for the bound variable in quantifiers. In fact, the system does not know of a global set VSym of variable symbols with a fixed typing function α, as we suggested in Def. 2.8. Instead, each variable is "declared" by the quantifier that binds it, so that is also where the type is given.

Concerning the "conventional" logical syntax, note that most accounts of first-order logic do not discuss subtypes, and accordingly, there is no need for type casts or type predicates. Also note that the syntax can vary considerably, even between conventional logic textbooks.

The operators $\forall$ and $\exists$ are called the *universal* and *existential quantifier*, respectively. We say that they *bind* the variable x in the sub-formula ϕ, or that ϕ is the *scope* of the quantified variable x. This is very similar to the way

in which a JAVA method body is the scope of the formal parameters declared in the method header. All variables occurring in a formula that are *not* bound by a quantifier are called *free*. For the calculus that is introduced later in this chapter, we are particularly interested in *closed* formulae, which have no free variables. These intuitions are captured by the following definition:

Definition 2.18. *We define $fv(t)$, the set of free variables of a term t, by*

- $fv(v) = \{v\}$ *for* $v \in \mathrm{VSym}$, *and*
- $fv(f(t_1, \ldots, t_n)) = \bigcup_{i=1,\ldots,n} fv(t_i)$.

The set of free variables of a formula is defined by

- $fv(p(t_1, \ldots, t_n)) = \bigcup_{i=1,\ldots,n} fv(t_i)$,
- $fv(t_1 \doteq t_2) = fv(t_1) \cup fv(t_2)$,
- $fv(\mathrm{true}) = fv(\mathrm{false}) = \emptyset$,
- $fv(!\,\phi) = fv(\phi)$,
- $fv(\phi \;\&\; \psi) = fv(\phi \mid \psi) = fv(\phi \mathbin{->} \psi) = fv(\phi) \cup fv(\psi)$, *and*
- $fv(\forall x.\phi) = fv(\exists x.\phi) = fv(\phi) \setminus \{x\}$.

A formula ϕ is called closed *iff $fv(\phi) = \emptyset$.*

Example 2.19. Given the signature from Example 2.14, the following are formulae:

$isEmpty(a)$ an atomic formula with free variable a
$a \doteq empty$ an equality atom with free variable a
$o \sqsubseteq \mathrm{List}$ a type predicate atom with free variable o
$o \sqsubseteq \bot$ a type predicate atom for the empty type with free variable o
$\forall l.(length(l) \doteq zero \mathbin{->} isEmpty(l))$
            a closed formula with a quantifier
$o \doteq empty \mid \forall o.o \sqsubseteq \top$
            a formula with one free and one bound occurrence of o

On the other hand, the following are not formulae:

$length(l)$ *length* is not a predicate symbol.
$isEmpty(o)$ wrong argument type
$isEmpty(isEmpty(a))$
            applying predicate on formula, instead of term
$a = empty$ equality should be $\doteq$
$\forall l.length(l)$ applying a quantifier to a term

2.4 Semantics

So far, we have only discussed the syntax, the textual structure of our logic. The next step is to assign a meaning, known as a *semantics*, to the terms and formulae.

2.4.1 Models

For compound formulae involving &, |, ∀, etc., our definition of a semantics should obviously correspond to their intuitive meaning as explained in the previous section. What is not clear is how to assign a meaning in the "base case", i.e., what is the meaning of atomic formulae like $p(a)$. It seems clear that this should depend on the meaning of the involved terms, so the semantics of terms also needs to be defined.

We do this by introducing the concept of a *model*. A model assigns a meaning (in terms of mathematical entities) to the basic building blocks of our logic, i.e., the types, and the function and predicate symbols. We can then define how to combine these meanings to obtain the meaning of any term or formula, always with respect to some model.

Actually, a model fixes only the meaning of function and predicate symbols. The meaning of the third basic building block, namely the variables is given by *variable assignments* which is introduced in Def. 2.23.[5]

When we think of a method call in a JAVA program, the returned value depends not only on the values of the arguments, but possibly also on the state of some other objects. Calling the method again in a modified state might give a different result. In this chapter, we do not take into account this idea of a changing state. A model gives a meaning to any term or formula, and in the same model, this meaning never changes. Evolving states will become important in Chapter 3.

Before we state the definition, let us look into type casts, which receive a special treatment. Recall that in JAVA, the evaluation of a type cast expression `(A)o` checks whether the value of `o` has a dynamic type equal to `A` or a subtype of `A`. If this is the case, the value of the cast is the same as the value of `o`, though the expression `(A)o` has static type `A`, independently of what the static type of `o` was. If the dynamic type of the value of `o` does *not* fit the type `A`, a `ClassCastException` is thrown.

In a logic, we want *every* term to have a *value*. It would greatly complicate things if we had to take things like exceptions into account. We therefore take the following approach:

1. The value of a term $(A)t$ is the same as the value of t, provided the value of t "fits" the type A.
2. Otherwise, the term is given an arbitrary value, but still one that "fits" its static type A.

If we want to differentiate between these two cases, we can use a type predicate formula $t \sqsubseteq A$: this is defined to hold exactly if the value of t "fits" the type A.

---

[5] The reason for keeping the variables separate is that the variable assignment is occasionally modified in the semantic definitions, whereas the model stays the same.

Definition 2.20. *Given a type hierarchy and a signature as before, a model 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[6]

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

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*

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

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

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

- *for type casts, $\mathcal{I}((A))(x) = x$ if $\delta(x) \sqsubseteq A$, otherwise $\mathcal{I}((A))(x)$ is an arbitrary but fixed*[7] *element of $\mathcal{D}^A$, and*
- *for equality, $\mathcal{I}(\doteq) = \{(d, d) \mid d \in \mathcal{D}\}$,*
- *for type predicates, $\mathcal{I}(\sqsubseteq A) = \mathcal{D}^A$.*

As we promised in the beginning of Section 2.1, every domain element d has a dynamic type $\delta(d)$, just like every object created when executing a JAVA program has a dynamic type. Also, just like in JAVA, the dynamic type of a domain element cannot be an abstract type.

Example 2.21. For the type hierarchy from Example 2.6 and the signature from Example 2.14, the "intended" model $\mathcal{M}_1 = (\mathcal{D}, \delta, \mathcal{I})$ may be described as follows:

Given a state in the execution of a JAVA program, let AL be the set of all existing `ArrayList` objects. We assume that there is at least one `ArrayList` object `e` that is currently empty. We denote some arbitrary but fixed `ArrayList` object (possibly equal to `e`) by `o`. Also, let $I := \{-2^{31}, \ldots, 2^{31} - 1\}$ be the set of all possible values for a JAVA `int`.[8] Now let

---

[6] $\mathcal{D}^A$ is our formal definition of the set of all domain elements that "fit" the type A.

[7] 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.

[8] The question of how best to reason about JAVA arithmetic is actually quite complex, and is covered in Chapter 12. Here, we take a restricted range of integers for the purpose of explaining the concept of a model.

$$\mathcal{D} := AL \,\dot{\cup}\, I \,\dot{\cup}\, \{\texttt{null}\} \ .$$

We define δ by

$$\delta(d) := \begin{cases} \text{int} & \text{if } d \in I \\ \text{ArrayList} & \text{if } d \in AL \\ \text{Null} & \text{if } d = \texttt{null} \end{cases}$$

With those definitions, we get

$$\mathcal{D}^{\top} = AL \,\dot{\cup}\, I \,\dot{\cup}\, \{\texttt{null}\}$$
$$\mathcal{D}^{\text{int}} = I$$
$$\mathcal{D}^{\text{Object}} = \mathcal{D}^{\text{AbstractCollection}} = \mathcal{D}^{\text{List}} =$$
$$\mathcal{D}^{\text{AbstractList}} = \mathcal{D}^{\text{ArrayList}} = AL \,\dot{\cup}\, \{\texttt{null}\}$$
$$\mathcal{D}^{\text{Null}} = \{\texttt{null}\}$$
$$\mathcal{D}^{\perp} = \emptyset$$

Now, we can fix the interpretations of the function symbols:

$$\mathcal{I}(zero)() := 0$$
$$\mathcal{I}(plus)(x, y) := x + y \quad \text{(with Java's overflow behaviour)}$$
$$\mathcal{I}(empty)() := \texttt{e}$$
$$\mathcal{I}(length)(l) := \begin{cases} \text{the length of } l & \text{if } l \neq \texttt{null} \\ 0 & \text{if } l = \texttt{null} \end{cases}$$

Note that the choice of 0 for the *length* of `null` is arbitrary, since `null` does not represent a list. Most of the interpretation of casts is fixed, but it needs to be completed for arguments that are not of the "right" type:

$$\mathcal{I}((\top))(d) := d$$
$$\mathcal{I}((\text{int}))(d) := \begin{cases} d & \text{if } d \in I \\ 23 & \text{otherwise} \end{cases}$$
$$\mathcal{I}((\text{Object}))(d) := \begin{cases} d & \text{if } d \in AL \,\dot{\cup}\, \{\texttt{null}\} \\ \texttt{o} & \text{otherwise} \end{cases}$$
$$\vdots$$

Note how the interpretation must produce a value of the correct type for every combination of arguments, even those that would maybe lead to a `NullPointerException` or a `ClassCastException` in Java execution. For the *isEmpty* predicate, we can define:

$$\mathcal{I}(isEmpty) := \{l \in AL \mid l \text{ is an empty ArrayList}\} \ .$$

The interpretation of $\dot{=}$ and of the type predicates is fixed by the definition of a model:

$$\mathcal{I}(\doteq) := \{(d, d) \mid d \in \mathcal{D}\}$$
$$\mathcal{I}(\sqsubseteq\top) := AL \,\dot\cup\, I \,\dot\cup\, \{\texttt{null}\}$$
$$\mathcal{I}(\sqsubseteq\text{int}) := I$$
$$\mathcal{I}(\sqsubseteq\text{Object}) := AL \,\dot\cup\, \{\texttt{null}\}$$
$$\vdots$$

Example 2.22. While the model in the previous example follows the intended meanings of the types, functions, and predicates quite closely, there are also models that have a completely different behaviour. For instance, we can define a model $\mathcal{M}_2$ with

$$\mathcal{D} := \{\Box, \Diamond\} \quad \text{with} \quad \delta(\Box) := \text{int} \quad \text{and} \quad \delta(\Diamond) := \text{Null} \ .$$

This gives us:

$$\mathcal{D}^\top = \{\Box, \Diamond\}$$
$$\mathcal{D}^{\text{int}} = \{\Box\}$$
$$\mathcal{D}^{\text{Object}} = \mathcal{D}^{\text{AbstractCollection}} = \mathcal{D}^{\text{List}} =$$
$$\mathcal{D}^{\text{AbstractList}} = \mathcal{D}^{\text{ArrayList}} = \mathcal{D}^{\text{Null}} = \{\Diamond\}$$
$$\mathcal{D}^\bot = \emptyset$$

The interpretation of the functions can be given by:

$$\mathcal{I}(zero)() := \Box \qquad\qquad \mathcal{I}((\top))(d) := d$$
$$\mathcal{I}(plus)(x, y) := \Box \qquad\qquad \mathcal{I}((\text{int}))(d) := \Box$$
$$\mathcal{I}(empty)() := \Diamond \qquad\qquad \mathcal{I}((\text{Object}))(d) := \Diamond$$
$$\mathcal{I}(length)(l) := \Box \qquad\qquad\qquad \vdots$$

and the predicates by:

$$\mathcal{I}(isEmpty) := \emptyset$$
$$\mathcal{I}(\doteq) := \{(\Box, \Box), (\Diamond, \Diamond)\}$$
$$\mathcal{I}(\sqsubseteq\top) := \{\Box, \Diamond\}$$
$$\mathcal{I}(\sqsubseteq\text{int}) := \{\Box\}$$
$$\mathcal{I}(\sqsubseteq\text{Object}) := \{\Diamond\}$$
$$\vdots$$

The following definitions apply to this rather nonsensical model as well as to the one defined in the previous example. In Section 2.4.3, we introduce a way of restricting which models we are interested in.

2.4.2 The Meaning of Terms and Formulae

A model is not quite sufficient to give a meaning to an arbitrary term or formula: it says nothing about the variables. For this, we introduce the notion of a variable assignment.

Definition 2.23. *Given a model* $(\mathcal{D}, \delta, \mathcal{I})$, *a* variable assignment *is a function* $\beta : \mathrm{VSym} \to \mathcal{D}$, *such that*

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

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:*

$$\beta_x^d(y) := \begin{cases} d & \textit{if } y = x \\ \beta(y) & \textit{otherwise} \end{cases}$$

We are now ready to define the semantics of terms.

Definition 2.24. *Let* $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$ *be a model, and* β *a variable assignment. We inductively define the* valuation function $\mathrm{val}_\mathcal{M}$ *by*

- $\mathrm{val}_{\mathcal{M},\beta}(x) = \beta(x)$ *for any variable* x.
- $\mathrm{val}_{\mathcal{M},\beta}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathrm{val}_{\mathcal{M},\beta}(t_1), \ldots, \mathrm{val}_{\mathcal{M},\beta}(t_n))$.

For a ground term t, *we simply write* $\mathrm{val}_\mathcal{M}(t)$, *since* $\mathrm{val}_{\mathcal{M},\beta}(t)$ *is independent of* β.

Example 2.25. Given the signature from Example 2.14 and the models $\mathcal{M}_1$ and $\mathcal{M}_2$ from Examples 2.21 and 2.22, we can define variable assignments β_1 resp. β_2 as follows:

$$\begin{array}{ll} \beta_1(n) := 5 & \beta_2(n) := \square \\ \beta_1(o) := \texttt{null} & \beta_2(o) := \diamondsuit \\ \beta_1(l) := \texttt{e} & \beta_2(l) := \diamondsuit \\ \beta_1(a) := \texttt{e} & \beta_2(a) := \diamondsuit \end{array}$$

We then get the following values for the terms from Example 2.16:

t	$\mathrm{val}_{\mathcal{M}_1,\beta_1}(t)$	$\mathrm{val}_{\mathcal{M}_2,\beta_2}(t)$
n	5	$\square$
empty	e	$\diamondsuit$
$plus(n, n)$	10	$\square$
$plus(n, plus(n, n))$	15	$\square$
$length(a)$	0	$\square$
$length((\mathrm{List})o)$	0	$\square$
$(\mathrm{int})o$	23	$\square$

The semantics of formulae is defined in a similar way: we define a validity relation that says whether some formula is valid in a given model under some variable assignment.

Definition 2.26. *Let* $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$ *be a model, and* β *a variable assignment. We inductively define the* validity relation $\models$ *by*

- $\mathcal{M}, \beta \models p(t_1, \ldots, t_n)$ *iff* $(\mathrm{val}_{\mathcal{M},\beta}(t_1), \ldots, \mathrm{val}_{\mathcal{M},\beta}(t_n)) \in \mathcal{I}(p)$.
- $\mathcal{M}, \beta \models \mathrm{true}$.
- $\mathcal{M}, \beta \not\models \mathrm{false}$.
- $\mathcal{M}, \beta \models\ !\phi$ *iff* $\mathcal{M}, \beta \not\models \phi$.
- $\mathcal{M}, \beta \models \phi \,\&\, \psi$ *iff* $\mathcal{M}, \beta \models \phi$ *and* $\mathcal{M}, \beta \models \psi$.
- $\mathcal{M}, \beta \models \phi \mid \psi$ *iff* $\mathcal{M}, \beta \models \phi$ *or* $\mathcal{M}, \beta \models \psi$, *or both*.
- $\mathcal{M}, \beta \models \phi \rightarrow \psi$ *iff* if $\mathcal{M}, \beta \models \phi$, *then also* $\mathcal{M}, \beta \models \psi$.
- $\mathcal{M}, \beta \models \forall x.\phi$ *(for a variable x:A) iff* $\mathcal{M}, \beta_x^d \models \phi$ *for every* $d \in \mathcal{D}^A$.
- $\mathcal{M}, \beta \models \exists x.\phi$ *(for a variable x:A) iff there is some* $d \in \mathcal{D}^A$ *such that* $\mathcal{M}, \beta_x^d \models \phi$.

If $\mathcal{M}, \beta \models \phi$, *we say that* ϕ *is* valid in the model $\mathcal{M}$ *under the variable assignment* β. *For a closed formula* ϕ, *we write* $\mathcal{M} \models \phi$, *since* β *is then irrelevant.*

Example 2.27. Let us consider the semantics of the formula

$$\forall l.(length(l) \doteq zero \rightarrow isEmpty(l))$$

in the model $\mathcal{M}_1$ described in Example 2.21. Intuitively, we reason as follows: the formula states that any list l which has length 0 is empty. But in our model, `null` is a possible value for l, and `null` has length 0, but is not considered an empty list. So the statement does not hold.

Formally, we start by looking at the smallest constituents and proceed by investigating the validity of larger and larger sub-formulae.

1. Consider the term $length(l)$. Its value $\mathrm{val}_{\mathcal{M}_1,\beta}(length(l))$ is the length of the `ArrayList` object identified by $\beta(l)$, or 0 if $\beta(l) = $ `null`.
2. $\mathrm{val}_{\mathcal{M}_1,\beta}(zero)$ is 0.
3. Therefore, $\mathcal{M}_1, \beta \models length(l) \doteq zero$ exactly if $\beta(l)$ is an `ArrayList` object of length 0, or $\beta(l)$ is `null`.
4. $\mathcal{M}_1, \beta \models isEmpty(l)$ iff $\beta(l)$ is an empty `ArrayList` object.
5. Whenever the length of an `ArrayList` object is 0, it is also empty.
6. `null` is *not* an empty `ArrayList` object.
7. Hence, $\mathcal{M}_1, \beta \models length(l) \doteq zero \rightarrow isEmpty(l)$ holds iff $\beta(l)$ is not `null`.
8. For any β, we have $\mathcal{M}_1, \beta_l^{\mathtt{null}} \not\models length(l) \doteq zero \rightarrow isEmpty(l)$, because $\beta_l^{\mathtt{null}}(l) = $ `null`.
9. Therefore, $\mathcal{M}_1, \beta \not\models \forall l.(length(l) \doteq zero \rightarrow isEmpty(l))$.

In the other model, $\mathcal{M}_2$ from Example 2.22,

1. $\mathrm{val}_{\mathcal{M}_2,\beta}(length(l)) = \square$, whatever $\beta(l)$ is.
2. $\mathrm{val}_{\mathcal{M}_2,\beta}(zero)$ is also $\square$.
3. Therefore, $\mathcal{M}_2, \beta \models length(l) \doteq zero$ holds for any β.
4. There is *no* $\beta(l)$ such that $\mathcal{M}_2, \beta \models isEmpty(l)$ holds.
5. Thus, there is no β such that $\mathcal{M}_2, \beta \models length(l) \doteq zero \rightarrow isEmpty(l)$.
6. In particular, $\mathcal{M}_2, \beta_l^{\mathtt{null}} \not\models length(l) \doteq zero \rightarrow isEmpty(l)$ for all β.
7. Therefore, $\mathcal{M}_2, \beta \not\models \forall l.(length(l) \doteq zero \rightarrow isEmpty(l))$.

This result is harder to explain intuitively, since the model $\mathcal{M}_2$ is itself unintuitive. But our description of the model and the definitions of the semantics allow us to determine the truth of any formula in the model.

In the example, we have seen a formula that is valid in neither of the two considered models. However, the reader might want to check that there are also models in which the formula holds.[9] But there are also formulae that hold in all models, or in none. We have special names for such formulae.

Definition 2.28. *Let a fixed type hierarchy and signature be given.*[10]

- *A formula ϕ is* logically valid *if $\mathcal{M}, \beta \models \phi$ for any model $\mathcal{M}$ and any variable assignment β.*
- *A formula ϕ is* satisfiable *if $\mathcal{M}, \beta \models \phi$ for some model $\mathcal{M}$ and some variable assignment β.*
- *A formula is* unsatisfiable *if it is not satisfiable.*

It is important to realize that logical validity is a very different notion from the validity in a particular model. We have seen in our examples that there are many models for any given signature, most of them having nothing to do with the intended meaning of symbols. While validity in a model is a relation between a formula and a model (and a variable assignment), logical validity is a property of a formula. In Section 2.5, we show that it is even possible to check logical validity without ever talking about models.

For the time being, here are some examples where the validity/satisfiability of simple formulae is determined through explicit reasoning about models.

Example 2.29. For any formula ϕ, the formula

$$\phi \mid !\,\phi$$

is logically valid: Consider the semantics of ϕ. For any model $\mathcal{M}$ and any variable assignment β, either $\mathcal{M}, \beta \models \phi$, or not. If $\mathcal{M}, \beta \models \phi$, the semantics of $\mid$ in Def. 2.26 tells us that also $\mathcal{M}, \beta \models \phi \mid !\,\phi$. Otherwise, the semantics of the negation ! tells us that $\mathcal{M}, \beta \models !\,\phi$, and therefore again $\mathcal{M}, \beta \models \phi \mid !\,\phi$. So our formula holds in any model, under any variable assignment, and is thus logically valid.

Example 2.30. For any formula ϕ, the formula

$$\phi \,\&\, !\,\phi$$

---

[9] Hint: define a model like $\mathcal{M}_1$, but let $\mathcal{I}(length)(\texttt{null}) = -1$.

[10] It is important to fix the type hierarchy: there are formulae which are logically valid in some type hierarchies, unsatisfiable in others, and satisfiable but not valid in others still. For instance, it might amuse the interested reader to look for such type hierarchies for the formula $\exists x.\, x \in A \,\&\, !\, x \in B$.

is unsatisfiable: Consider an arbitrary, but fixed model $\mathcal{M}$ and a variable assignment β. For $\mathcal{M}, \beta \models \phi \ \& \ !\phi$ to hold, according to Def. 2.26, both $\mathcal{M}, \beta \models \phi$ and $\mathcal{M}, \beta \models !\phi$ must hold. This cannot be the case, because of the semantics of !. Hence, $\mathcal{M}, \beta \models \phi \ \& \ !\phi$ does not hold, irrespective of the model and the variable assignment, which means that the formula is unsatisfiable.

Example 2.31. The formula

$$\exists x. x \doteq x$$

for some variable $x{:}A$ with $A \in \mathcal{T}_q$ is logically valid: Consider an arbitrary, but fixed model $\mathcal{M}$ and a variable assignment β. We have required in Def. 2.20 that $\mathcal{D}^A$ is non-empty. Pick an arbitrary element $a \in \mathcal{D}^A$ and look at the modified variable assignment β_x^a. Clearly, $\mathcal{M}, \beta_x^a \models x \doteq x$, since both sides of the equation are equal terms and must therefore evaluate to the same domain element (namely a). According to the semantics of the $\exists$ quantifier in Def. 2.26, it follows that $\mathcal{M}, \beta \models x \doteq x$. Since this holds for any model and variable assignment, the formula is logically valid.

Example 2.32. The formula

$$\forall l. (length(l) \doteq zero \rightarrow isEmpty(l))$$

is satisfiable. It is not logically valid, since it does not hold in every model, as we have seen in Example 2.27. To see that it is satisfiable, take a model $\mathcal{M}$ with

$$\mathcal{I}(isEmpty) := \mathcal{D}^{\text{List}}$$

so that $isEmpty(l)$ is true for every value of l. Accordingly, in $\mathcal{M}$, the implication $length(l) \doteq zero \rightarrow isEmpty(l)$ is also valid for any variable assignment. The semantics of the $\forall$ quantifier then tells us that

$$\mathcal{M} \models \forall l. (length(l) \doteq zero \rightarrow isEmpty(l))$$

so the formula is indeed satisfied by $\mathcal{M}$.

Example 2.33. The formula

$$(A)x \doteq x \rightarrow x \in A$$

with $x{:}\top$ is logically valid for any type hierarchy and any type A: Remember that

$$\text{val}_{\mathcal{M}, \beta}((A)x) = \mathcal{I}((A))(\beta(x)) \in \mathcal{D}^A \ .$$

Now, if $\beta(x) \in \mathcal{D}^A$, then $\text{val}_{\mathcal{M}, \beta}((A)x) = \beta(x)$, so $\mathcal{M}, \beta \models (A)x \doteq x$. On the other hand, if $\beta(x) \notin \mathcal{D}^A$, then it cannot be equal to $\text{val}_{\mathcal{M}, \beta}((A)x)$, so $\mathcal{M}, \beta \not\models (A)x \doteq x$. Thus, if $(A)x \doteq x$, holds, then $\beta(x) \in \mathcal{D}^A$, and therefore $\mathcal{M}, \beta \models x \in A$.

The converse

$$x \in A \rightarrow (A)x \doteq x$$

is also logically valid for any type hierarchy and any type A: if $\mathcal{M}, \beta \models x \in A$, then $\beta \in \mathcal{D}^A$, and therefore $\mathcal{M}, \beta \models (A)x \doteq x$.

Logical Consequence

A concept that is quite central to many other introductions to logic, but that is hardly encountered when dealing with the KeY system, is that of *logical consequence*. We briefly explain it here.

Given a set of closed formulae M and a formula ϕ, ϕ is said to be a *logical consequence* of M, written $M \models \phi$, iff for all models $\mathcal{M}$ and variable assignments β such that $\mathcal{M}, \beta \models \psi$ for all $\psi \in M$, it also holds that $\mathcal{M}, \beta \models \phi$.

In other words, ϕ is not required to be satisfied in all models and under all variable assignments, but only under those that satisfy all elements of M.

For instance, for any closed formulae ϕ and ψ, $\{\phi, \psi\} \models \phi \mathrel{\&} \psi$, since $\phi \mathrel{\&} \psi$ holds for all $\mathcal{M}, \beta$ for which both ϕ and ψ hold.

Two formulae ϕ and ψ are called *logically equivalent* if for all models $\mathcal{M}$ and variable assignments β, $\mathcal{M}, \beta \models \phi$ iff $\mathcal{M}, \beta \models \psi$.

Note 2.34. The previous example shows that type predicates are not really necessary in our logic, since a sub-formula $t \sqsubseteq A$ could always be replaced by $(A)t \doteq t$. In the terminology of the above sidebar, the two formulae are logically equivalent. Another formula that is easily seen to be logically equivalent to $t \sqsubseteq A$ is

$$\exists y. y \doteq t$$

with a variable $y{:}A$. It is shown in Section 2.5.6 however, that the main way of reasoning about types, and in particular about type casts in our calculus is to collect information about dynamic types using type predicates. Therefore, adding type predicates to our logic turns out to be the most convenient approach for reasoning, even if they do not add anything to the expressivity.

2.4.3 Partial Models

Classically, the logically valid formulae have been at the centre of attention when studying a logic. However, when dealing with formal methods, many of the involved types have a fixed *intended* meaning. For instance, in our examples, the type int is certainly intended to denote the 4 byte two's complement integers of the JAVA language, and the function symbol *plus* should denote the addition of such integers.[11] On the other hand, for some types and symbols, we *are* interested in all possible meanings.

To formally express this idea, we introduce the concept of a *partial model*, which gives a meaning to parts of a type hierarchy and signature. We then define what it means for a model to extend a partial model, and look only at such models.

---

[11] We repeat that the issue of reasoning about JAVA arithmetic in the KeY system is actually more complex ($\Rightarrow$ Chap. 12).

The following definition of a partial model is somewhat more complex than might be expected. If we want to fix the interpretation of some of the functions and predicates in our signature, it is not sufficient to say which, and to give their interpretations. The interpretations must act on some domain, and the domain elements must have some type. For instance, if we want *plus* to represent the addition of Java integers, we must also identify a subset of the domain which should be the domain for the int type.

In addition, we want it to be possible to fix the interpretation of some functions only on parts of the domain. For instance, we might not want to fix the result of a division by zero.[12]

Definition 2.35. *Given a type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ and a corresponding signature $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$, we define a* partial model *to be a quintuple $(\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$ consisting of*

- *a set of* fixed types $\mathcal{T}_0 \subseteq \mathcal{T}_d$,
- *a set $\mathcal{D}_0$ called the* partial domain,
- *a* dynamic type function $\delta_0 : \mathcal{D}_0 \to \mathcal{T}_0$,
- *a* fixing function D_0, *and*
- *a* partial interpretation $\mathcal{I}_0$,

where

- $\mathcal{D}_0^A := \{d \in \mathcal{D}_0 \mid \delta_0(d) \sqsubseteq A\}$ *is non-empty for all $A \in \mathcal{T}_0$,*
- *for any $f : A_1, \ldots, A_n \to A_0 \in \mathrm{FSym}$ with all $A_i \in \mathcal{T}_0$, D_0 yields a set of tuples of domain elements*

$$D_0(f) \subseteq \mathcal{D}_0^{A_1} \times \ldots \times \mathcal{D}_0^{A_n}$$

and $\mathcal{I}_0$ yields a function

$$\mathcal{I}_0(f) : D_0(f) \to \mathcal{D}_0^{A_0} \ ,$$

and
- *for any $p : A_1, \ldots, A_n \in \mathrm{PSym}$ with all $A_i \in \mathcal{T}_0$, D_0 yields a set of tuples of domain elements*

$$D_0(p) \subseteq \mathcal{D}_0^{A_1} \times \cdots \times \mathcal{D}_0^{A_n}$$

and $\mathcal{I}_0$ yields a subset

$$\mathcal{I}_0(p) \subseteq D_0(p) \ ,$$

and

---

[12] Instead of using *partial functions* for cases like division by zero, i.e., functions which do not have a value for certain arguments, we consider our functions to be total, but we might not fix (or know, or care about) the value for some arguments. This corresponds to the under-specification approach advocated by Hähnle [2005].

- *for any $f : A_1, \ldots, A_n \to A_0 \in$ FSym, resp. $p : A_1, \ldots, A_n \in$ PSym with one of the $A_i \notin \mathcal{T}_0$, $D_0(f) = \emptyset$, resp. $D_0(p) = \emptyset$.*

This is a somewhat complex definition, so we explain the meaning of its various parts. As mentioned above, a part of the domain needs to be fixed for the interpretation to act upon, and the dynamic type of each element of that partial domain needs to be identified. This is the role of $\mathcal{T}_0$, $\mathcal{D}_0$, and δ_0. The fixing function D_0 says for which tuples of domain elements and for which functions this partial model should prescribe an interpretation. In particular, if D_0 gives an empty set for some symbol, then the partial model does not say anything at all about the interpretation of that symbol. If D_0 gives the set of all element tuples corresponding to the signature of that symbol, then the interpretation of that symbol is completely fixed. Consider the special case of a constant symbol c: there is only one 0-tuple, namely (), so the fixing function can be either $D_0(c) = \{()\}$, meaning that the interpretation of c is fixed to some domain element $\mathcal{I}_0(c)()$, or $D_0(c) = \emptyset$, meaning that it is not fixed.

Finally, the partial interpretation $\mathcal{I}_0$ specifies the interpretation for those tuples of elements where the interpretation should be fixed.

Example 2.36. We use the type hierarchy from the previous examples, and add to the signature from Example 2.14 a function symbol $div :$ int, int $\to$ int. We want to define a partial model that fixes the interpretation of *plus* to be the two's complement addition of four-byte integers that is used by JAVA. The interpretation of div should behave like JAVA's division unless the second argument is zero, in which case we do not require any specific interpretation. This is achieved by choosing

$$
\begin{aligned}
\mathcal{T}_0 &:= \{\text{int}\} \\
\mathcal{D}_0 &:= \{-2^{31}, \ldots, 2^{31} - 1\} \\
\delta_0(x) &:= \text{int} \quad \text{for all } x \in \mathcal{D}_0 \\
D_0(plus) &:= \mathcal{D}_0 \times \mathcal{D}_0 \\
D_0(div) &:= \mathcal{D}_0 \times (\mathcal{D}_0 \setminus \{0\}) \\
\mathcal{I}_0(plus)(x, y) &:= x + y \quad \text{(with JAVA overflow)} \\
\mathcal{I}_0(div)(x, y) &:= x/y \quad \text{(with JAVA overflow and rounding)}
\end{aligned}
$$

We have not yet defined exactly what it means for some model to adhere to the restrictions expressed by a partial model. In order to do this, we first define a refinement relation between partial models. Essentially, one partial model refines another if its restrictions are stronger, i.e., if it contains all the restrictions of the other, and possibly more. In particular, more functions and predicates may be fixed, as well as more types and larger parts of the domain. It is also possible to fix previously underspecified parts of the interpretation. However, any types, interpretations, etc. that were previously fixed must remain the same. This is captured by the following definition:

Definition 2.37. *A partial model* $(\mathcal{T}_1, \mathcal{D}_1, \delta_1, D_1, \mathcal{I}_1)$ *refines another partial model* $(\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$, *if*

- $\mathcal{T}_1 \supseteq \mathcal{T}_0$,
- $\mathcal{D}_1 \supseteq \mathcal{D}_0$,
- $\delta_1(d) = \delta_0(d)$ *for all* $d \in \mathcal{D}_0$,
- $D_1(f) \supseteq D_0(f)$ *for all* $f \in \mathrm{FSym}$,
- $D_1(p) \supseteq D_0(p)$ *for all* $p \in \mathrm{PSym}$,
- $\mathcal{I}_1(f)(d_1, \ldots, d_n) = \mathcal{I}_0(f)(d_1, \ldots, d_n)$ *for all* $(d_1, \ldots, d_n) \in D_0(f)$ *and* $f \in \mathrm{FSym}$, *and*
- $\mathcal{I}_1(p) \cap D_0(p) = \mathcal{I}_0(p)$ *for all* $p \in \mathrm{PSym}$.

Example 2.38. We define a partial model that refines the one in the previous example by also fixing the interpretation of *zero*, and by restricting the division of zero by zero to give one.

$$
\begin{aligned}
\mathcal{T}_1 &:= \{\mathrm{int}\} \\
\mathcal{D}_1 &:= \{-2^{31}, \ldots, 2^{31} - 1\} \\
\delta_1(x) &:= \mathrm{int} \quad \text{for all } x \in \mathcal{D}_0 \\
D_1(zero) &:= \{()\} \quad \text{(the empty tuple)} \\
D_1(plus) &:= \mathcal{D}_0 \times \mathcal{D}_0 \\
D_1(div) &:= (\mathcal{D}_0 \times (\mathcal{D}_0 \setminus \{0\})) \cup \{(0, 0)\} \\
\mathcal{I}_1(zero)() &:= 0 \\
\mathcal{I}_1(plus)(x, y) &:= x + y \quad \text{(with JAVA overflow)} \\
\mathcal{I}_1(div)(x, y) &:= \begin{cases} 1 & \text{if } x = y = 0, \\ x/y & \text{otherwise (with JAVA overflow and rounding)} \end{cases}
\end{aligned}
$$

To relate models to partial models, we can simply see models as a special kind of partial model in which all interpretations are completely fixed:

Definition 2.39. *Let* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *be a type hierarchy. Any* model $(\mathcal{D}, \delta, \mathcal{I})$ *may also be regarded as a* partial model $(\mathcal{T}_d, \mathcal{D}, \delta, D, \mathcal{I})$, *by letting* $D(f) = \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n}$ *for all function symbols* $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}$, *and* $D(p) = \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n}$ *for all predicate symbols* $p : A_1, \ldots, A_n \in \mathrm{PSym}$.

The models are special among the partial models in that they cannot be refined any further.

It is now clear how to identify models which adhere to the restrictions expressed in some partial model: we want exactly those models which are refinements of that partial model. To express that we are only interested in such models, we can *relativise* our definitions of validity, etc.

Definition 2.40. *Let a fixed type hierarchy and signature be given. Let* $\mathcal{M}_0$ *be a partial model.*

- *A formula* ϕ *is* logically valid with respect to $\mathcal{M}_0$ *if* $\mathcal{M}, \beta \models \phi$ *for any model* $\mathcal{M}$ *that refines* $\mathcal{M}_0$ *and any variable assignment* β.

- *A formula ϕ is* satisfiable with respect to $\mathcal{M}_0$ *if* $\mathcal{M}, \beta \models \phi$ *for some model $\mathcal{M}$ that refines $\mathcal{M}_0$ and some variable assignment β.*
- *A formula is* unsatisfiable with respect to $\mathcal{M}_0$ *if it is not satisfiable with respect to $\mathcal{M}_0$.*

Example 2.41. Even though division is often thought of as a partial function, which is undefined for the divisor 0, from the standpoint of our logic, a division by zero certainly produces a value. So the formula

$$\forall x. \exists y. div(x, zero) \doteq y$$

is logically valid, simply because for any value of x, one can interpret the term $div(x, zero)$ and use the result as instantiation for y.

If we add constants *zero*, *one*, *two*, etc. with the obvious interpretations to the partial model of Example 2.36, then formulae like

$$plus(one, two) \doteq three$$

and

$$div(four, two) \doteq two$$

are logically valid with respect to that partial model, though they are not logically valid in the sense of Def. 2.28. However, it is not possible to add another fixed constant c to the partial model, such that

$$div(one, zero) \doteq c$$

becomes logically valid w.r.t. the partial model, since it does not fix the interpretation of the term $div(one, zero)$. Therefore, for any given fixed interpretation of the constant c there is a model $(\mathcal{D}, \delta, \mathcal{I})$ that refines the partial model and that interprets $div(one, zero)$ to something different, i.e.,

$$\mathcal{I}(div)(1, 0) \neq \mathcal{I}(c)$$

So instead of treating *div* as a partial function, it is left *under-specified* in the partial model. Note that we handled the interpretation of "undefined" type casts in exactly the same way. See the sidebar on handling undefinedness (p. 90) for a discussion of this approach to partiality.

For the next two sections, we will not be talking about partial models or relative validity, but only about logical validity in the normal sense. We will however come back to partial models in Section 2.7.

2.5 A Calculus

We have seen in the examples after Definition 2.28 how a formula can be shown to be logically valid, using mathematical reasoning about models, the

definitions of the semantics, etc. The proofs given in these examples are however somewhat unsatisfactory in that they do not seem to be constructed in any systematic way. Some of the reasoning seems to require human intuition and resourcefulness. In order to use logic on a computer, we need a more systematic, algorithmic way of discovering whether some formula is valid. A direct application of our semantic definitions is not possible, since for infinite universes, in general, an infinite number of cases would have to be checked.

For this reason, we now present a *calculus* for our logic. A calculus describes a certain arsenal of purely syntactic operations to be carried out on formulae, allowing us to determine whether a formula is valid. More precisely, to rule out misunderstandings from the beginning, *if* a formula is valid, we are able to establish its validity by systematic application of our calculus. If it is invalid, it might be impossible to detect this using this calculus. Also note that the calculus only deals with logical validity in the sense of Def. 2.28, and not with validity w.r.t. some partial model. We will come back to these questions in Section 2.6 and 2.7.

The calculus consists of "rules" (see Fig. 2.2, 2.3, and 2.4), along with some definitions that say how these rules are to be applied to decide whether a formula is logically valid. We now present these definitions and explain most of the rules, giving examples to illustrate their use.

The basic building block to which the rules of our calculus are applied is the *sequent*, which is defined as follows:

Definition 2.42. *A* sequent *is a pair of sets of closed formulae written as*

$$\phi_1, \ldots, \phi_m \Longrightarrow \psi_1, \ldots, \psi_n \ .$$

The formulae ϕ_i on the left of the sequent arrow $\Longrightarrow$ *are called the* antecedent, *the formulae ψ_j on the right the* succedent *of the sequent. We use capital Greek letters to denote several formulae in the antecedent or succedent of a sequent, so by*

$$\Gamma, \phi \Longrightarrow \psi, \Delta$$

we mean a sequent containing ϕ in the antecedent, and ψ in the succedent, as well as possibly many other formulae contained in Γ, and Δ.

Note 2.43. Some authors define sequents using lists (sequences) or multi-sets of formulae in the antecedent or succedent. For us, sets are sufficient. So the sequent $\phi \Longrightarrow \phi, \psi$ is the same as $\phi, \phi \Longrightarrow \psi, \phi$.

Note 2.44. We do not allow formulae with free variables in our sequents. Free variables add technical difficulties and notoriously lead to confusion, since they have historically been used for several different purposes. Our formulation circumvents these difficulties by avoiding free variables altogether and sticking to closed formulae.

The intuitive meaning of a sequent

$$\phi_1, \ldots, \phi_m \implies \psi_1, \ldots, \psi_n$$

is the following:

> Whenever *all* the ϕ_i of the antecedent are true, then *at least one* of the ψ_j of the succedent is true.

Equivalently, we can read it as:

> It cannot be that *all* the ϕ_i of the antecedent are true, and *all* ψ_j of the succedent are false.

This whole statement represented by the sequent has to be shown *for all models*. If it can be shown for some model, we also say that the sequent is valid in that model. Since all formulae are closed, variable assignments are not important here. If we are simply interested in the logical validity of a single formula ϕ, we start with the simple sequent

$$\implies \phi$$

and try to construct a proof. Before giving the formal definition of what exactly constitutes a proof, we now go through a simple example.

2.5.1 An Example Proof

We proceed by applying the rules of the calculus to construct a *tree of sequents*. We demonstrate this by a proof of the validity of the formula

$$(p \ \& \ q) \rightarrow (q \ \& \ p)$$

where p and q are predicates with no arguments.[13] We start with

$$\implies (p \ \& \ q) \rightarrow (q \ \& \ p) \ .$$

In Fig. 2.2, we see a rule

$$\text{impRight} \ \frac{\Gamma, \phi \implies \psi, \Delta}{\Gamma \implies \phi \rightarrow \psi, \Delta} \ \ .$$

impRight is the name of the rule. It serves to handle implications in the succedent of a sequent. The sequent below the line is the *conclusion* of the rule, and the one above is its *premiss*. Some rules in Fig. 2.2 have several or no premisses, we will come to them later.

---

[13] Such predicates are sometimes called *propositional variables*, but they should not be confused with the variables of our logic.

The meaning of the rule is that if a sequent of the form of the premiss is valid, then the conclusion is also valid. We use it in the opposite direction: to prove the validity of the conclusion, it suffices to prove the premiss. We now apply this rule to our sequent, and write the result as follows:

$$\frac{(p \mathbin{\&} q) \Longrightarrow (q \mathbin{\&} p)}{\Longrightarrow (p \mathbin{\&} q) \mathbin{-\!>} (q \mathbin{\&} p)}$$

In this case, we take $p \mathbin{\&} q$ for the ϕ in the rule and $q \mathbin{\&} p$ for the ψ, with both Γ and Δ being empty.[14] There are now two rules in Fig. 2.2 that may be applied, namely andLeft and andRight. Let us use andLeft first. We add the result to the top of the previous proof:

$$\frac{\dfrac{p, q \Longrightarrow q \mathbin{\&} p}{p \mathbin{\&} q \Longrightarrow q \mathbin{\&} p}}{\Longrightarrow (p \mathbin{\&} q) \mathbin{-\!>} (q \mathbin{\&} p)}$$

In this case Γ contains the untouched formula $q \mathbin{\&} p$ of the succedent. Now, we apply andRight. Since this rule has two premisses, our proof *branches*.

$$\frac{\dfrac{p, q \Longrightarrow q \qquad p, q \Longrightarrow p}{p, q \Longrightarrow q \mathbin{\&} p}}{\dfrac{p \mathbin{\&} q \Longrightarrow q \mathbin{\&} p}{\Longrightarrow (p \mathbin{\&} q) \mathbin{-\!>} (q \mathbin{\&} p)}}$$

A rule with several premisses means that its conclusion is valid if all of the premisses are valid. We thus have to show the validity of the two sequents above the topmost line. We can now use the close rule on both of these sequents, since each has a formula occurring on both sides.

$$\frac{\dfrac{\overline{p, q \Longrightarrow q} \qquad \overline{p, q \Longrightarrow p}}{p, q \Longrightarrow q \mathbin{\&} p}}{\dfrac{p \mathbin{\&} q \Longrightarrow q \mathbin{\&} p}{\Longrightarrow (p \mathbin{\&} q) \mathbin{-\!>} (q \mathbin{\&} p)}}$$

The close rule has no premisses, which means that the goal of a branch where it is applied is successfully proven. We say that the branch is *closed*. We have applied the close rule on all branches, so that was it! All branches are closed, and therefore the original formula was logically valid.

---

[14] Γ, Δ, ϕ, ψ in the rule are place holders, also known as schema variables. The act of assigning concrete terms, formulae, or formula sets to schema variables is known as *matching*. See also Note 2.51 and Chapter 4 for details about pattern matching.

2.5.2 Ground Substitutions

Before discussing the workings of our calculus in a more rigorous way, we introduce a construct known as *substitution*. Substitutions are used by many of the rules that have to do with quantifiers, equality, etc.

Definition 2.45. *A* ground substitution *is a function τ that assigns a ground term to some finite set of variable symbols* $\mathrm{dom}(\tau) \subseteq \mathrm{VSym}$, *the* domain *of the substitution, with the restriction that*

> *if $v \in \mathrm{dom}(\tau)$ for a variable $v{:}B \in \mathrm{VSym}$, then $\tau(v) \in \mathrm{Trm}_A$, for some A with $A \sqsubseteq B$.*

We write $\tau = [u_1/t_1, \ldots, u_n/t_n]$ to denote the particular substitution defined by $\mathrm{dom}(\tau) = \{u_1, \ldots, u_n\}$ and $\tau(u_i) := t_i$.

We denote by τ_x the result of removing a variable from the domain of τ, i.e., $\mathrm{dom}(\tau_x) := \mathrm{dom}(\tau) \setminus \{x\}$ and $\tau_x(v) := \tau(v)$ for all $v \in \mathrm{dom}(\tau_x)$.

Example 2.46. Given the signature from the previous examples,

$$\tau = [o/empty, n/length(empty)]$$

is a substitution with

$$\mathrm{dom}(\tau) = \{o, n\} \ .$$

Note that the static type of *empty* is List, which is a subtype of Object, which is the type of the variable *o*. For this substitution, we have

$$\tau_o = [n/length(empty)]$$

and

$$\tau_n = [o/empty] \ .$$

We can also remove both variables from the domain of τ, which gives us

$$(\tau_o)_n = [] \ ,$$

the empty substitution with $\mathrm{dom}([]) = \emptyset$. Removing a variable that is not in the domain does not change τ:

$$\tau_a = \tau = [o/empty, n/length(empty)] \ .$$

The following is *not* a substitution:

$$[n/empty] \ ,$$

since the type List of the term *empty* is not a subtype of int, which is the type of the variable n.

Note 2.47. In Section 4.2.4, a more general concept of substitution is introduced, that also allows substituting terms with free variables. This can lead to various complications that we do not need to go into at this point.

We want to apply substitutions not only to variables, but also to terms and formulae.

Definition 2.48. *The application of a ground substitution τ is extended to non-variable terms by the following definitions:*

- $\tau(x) := x$ *for a variable $x \notin \mathrm{dom}(\tau)$.*
- $\tau(f(t_1, \ldots, t_n)) := f(\tau(t_1), \ldots, \tau(t_n))$.

The application of a ground substitution τ to a formula is defined by

- $\tau(p(t_1, \ldots, t_n)) := p(\tau(t_1), \ldots, \tau(t_n))$.
- $\tau(\mathrm{true}) := \mathrm{true}$ *and* $\tau(\mathrm{false}) := \mathrm{false}$.
- $\tau(!\,\phi) := !(\tau(\phi))$,
- $\tau(\phi \,\&\, \psi) := \tau(\phi) \,\&\, \tau(\psi)$, *and correspondingly for $\phi \mid \psi$ and $\phi \mathrel{-\!\!>} \psi$.*
- $\tau(\forall x.\phi) := \forall x.\tau_x(\phi)$ *and* $\tau(\exists x.\phi) := \exists x.\tau_x(\phi)$.

Example 2.49. Let's apply the ground substitution

$$\tau = [o/\mathit{empty}, n/\mathit{length}(\mathit{empty})]$$

from the previous example to some terms and formulae:

$$\tau(\mathit{plus}(n,n)) = \mathit{plus}(\mathit{length}(\mathit{empty}), \mathit{length}(\mathit{empty})) \ ,$$
$$\tau(n \doteq \mathit{length}((\mathrm{List})o)) = (\mathit{length}(\mathit{empty}) \doteq \mathit{length}((\mathrm{List})\mathit{empty})) \ .$$

By the way, this is an example of why we chose to use the symbol $\doteq$ instead of $=$ for the equality symbol in our logic. Here is an example with a quantifier:

$$\tau(\exists o.n \doteq \mathit{length}((\mathrm{List})o)) = (\exists o.(\mathit{length}(\mathit{empty}) \doteq \mathit{length}((\mathrm{List})o)) \ .$$

We see that the quantifier for o prevents the substitution from acting on the o inside its scope.

Some of our rules call for formulae *of the form* $[z/t](\phi)$ for some formula ϕ, variable z, and term t. In these cases, the rule is applicable to any formula that can be written in this way. Consider for instance the following rule from Fig. 2.3:

$$\mathsf{eqLeft} \ \frac{\Gamma, t_1 \doteq t_2, [z/t_1](\phi), [z/t_2](\phi) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2, [z/t_1](\phi) \Longrightarrow \Delta}$$
$$\text{if } \sigma(t_2) \sqsubseteq \sigma(t_1)$$

Looking at the conclusion, it requires two formulae

$$t_1 \doteq t_2 \quad \text{and} \quad [z/t_1](\phi)$$

in the antecedent. The rule adds the formula

$$[z/t_2](\phi)$$

to the antecedent of the sequent. Now consider the formulae

$$length(empty) \doteq zero \quad \text{and} \quad length(empty) \sqsubseteq int \ .$$

The right formula can also be written as

$$length(empty) \sqsubseteq int \ = \ [z/length(empty)](z \sqsubseteq int) \ .$$

In other words, in this example, we have:

$$t_1 = length(empty)$$
$$t_2 = zero$$
$$\phi = z \sqsubseteq int$$

Essentially, the z in ϕ marks an occurrence of t_1 in the formula $[z/t_1](\phi)$. The new formula added by the rule, $[z/t_2](\phi)$, is the result of replacing this occurrence by the term t_2.

We do not exclude the case that there are no occurrences of the variable z in ϕ, or that there are several occurrences. In the case of no occurrences, $[z/t_1](\phi)$ and $[z/t_2](\phi)$ are the same formula, so the rule application does not do anything. In the case of several occurrences, we replace several instances of t_1 by t_2 simultaneously.

Note that this is just an elegant, yet precise way of formulating our calculus rules. In the implementation of the KeY system, it is more convenient to replace one occurrence at a time.

2.5.3 Sequent Proofs

As we saw in the example of Section 2.5.1, a sequent proof is a tree that is constructed according to a certain set of rules. This is made precise by the following definition:

Definition 2.50. *A* proof tree *is a finite tree (shown with the root at the bottom), such that*

- *each node of the tree is annotated with a sequent*
- *each inner node of the tree is additionally annotated with one of those rules shown in Figs. 2.2, 2.3, and 2.4 that have at least one premiss. This rule relates the node's sequent to the sequents of its descendants. In particular, the number of 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).*

Note 2.51. A really rigorous definition of the concept of a proof would require a description of the *pattern matching* and replacement process that underlies the application of the rules. This is done to a certain extent in Chapter 4. For the time being, we assume that the reader understands that the Latin and Greek letters Γ, t_1, ϕ, z, A are actually place holders for arbitrary terms, formulae, types, etc. according to their context.

In a sense, models and proofs are complementary: to show that a formula is satisfiable, one has to describe a single model that satisfies it, as we did for instance in Example 2.32. To show that a formula is logically valid, we have previously shown that it is valid in any model, like for instance in Example 2.33. Now we can show logical validity by constructing a single proof.

2.5.4 The Classical First-Order Rules

Two rules in Fig. 2.2 carry a strange requirement: allRight and exRight require the choice of "$c : \to A$ a new constant, if $x{:}A$". The word "new" in this requirement means that the symbol c has not occurred in any of the sequents of the proof tree built so far. The idea is that to prove a statement for all x, one chooses an arbitrary but fixed c and proves the statement for that c. The symbol needs to be new since we are not allowed to assume anything about c (except its type).

If we use the calculus in the presence of a partial model in the sense of Section 2.4.3, we may only take a symbol c that is not fixed, i.e., $D_0(c) = \emptyset$. The reason is again to make sure that no knowledge about c can be assumed.

In order to permit the construction of proofs of arbitrary size, it is sensible to start with a signature that contains enough constant symbols of every type. We call signatures where this is the case "admissible":

Definition 2.52. *For any given type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$, an admissible signature is a signature that contains an infinite number of constant symbols $c :\to A$ for every non-empty type $A \in \mathcal{T}_q$.*

Since the validity or satisfiability of a formula cannot change if symbols are added to the signature, it never hurts to assume that our signature is admissible. And in an admissible signature, it is always possible to pick a new constant symbol of any type.

$$\text{andLeft } \frac{\Gamma, \phi, \psi \Longrightarrow \Delta}{\Gamma, \phi \,\&\, \psi \Longrightarrow \Delta} \qquad \text{andRight } \frac{\Gamma \Longrightarrow \phi, \Delta \quad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \,\&\, \psi, \Delta}$$

$$\text{orRight } \frac{\Gamma \Longrightarrow \phi, \psi, \Delta}{\Gamma \Longrightarrow \phi \mid \psi, \Delta} \qquad \text{orLeft } \frac{\Gamma, \phi \Longrightarrow \Delta \quad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \mid \psi \Longrightarrow \Delta}$$

$$\text{impRight } \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \rightarrow \psi, \Delta} \qquad \text{impLeft } \frac{\Gamma \Longrightarrow \phi, \Delta \quad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \rightarrow \psi \Longrightarrow \Delta}$$

$$\text{notLeft } \frac{\Gamma \Longrightarrow \phi, \Delta}{\Gamma, !\,\phi \Longrightarrow \Delta} \qquad \text{notRight } \frac{\Gamma, \phi \Longrightarrow \Delta}{\Gamma \Longrightarrow !\,\phi, \Delta}$$

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

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

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

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

$$\text{close } \frac{}{\Gamma, \phi \Longrightarrow \phi, \Delta}$$

$$\text{closeFalse } \frac{}{\Gamma, \mathrm{false} \Longrightarrow \Delta} \qquad \text{closeTrue } \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

Fig. 2.2. Classical first-order rules

We start our demonstration of the rules with some simple first-order proofs. We assume the minimal type hierarchy that consists only of $\bot$ and $\top$, see Example 2.7.

Example 2.53. Let the signature contain a predicate $p : \top$ and two variables $x{:}\top, y{:}\top$. We also assume an infinite set of constants $c_1, c_2, \ldots{:}\top$. We construct a proof for the formula

$$\exists x.\forall y.(p(x) \rightarrow p(y)) \ .$$

We start with the sequent

$$\Longrightarrow \exists x.\forall y.(p(x) \rightarrow p(y))$$

for which only the exRight rule is applicable. We need to choose a term t for the instantiation. For lack of a better candidate, we take c_1:[15]

$$\frac{\Longrightarrow \forall y.(p(c_1) -> p(y)), \exists x.\forall y.(p(x) -> p(y))}{\Longrightarrow \exists x.\forall y.(p(x) -> p(y))}$$

Note that the original formula is left in the succedent. This means that we are free to choose a more suitable instantiation later on. For the time being, we apply the allRight rule, picking c_2 as the new constant.

$$\frac{\dfrac{\Longrightarrow p(c_1) -> p(c_2), \exists x.\forall y.(p(x) -> p(y))}{\Longrightarrow \forall y.(p(c_1) -> p(y)), \exists x.\forall y.(p(x) -> p(y))}}{\Longrightarrow \exists x.\forall y.(p(x) -> p(y))}$$

Next, we apply impRight:

$$\frac{\dfrac{\dfrac{p(c_1) \Longrightarrow p(c_2), \exists x.\forall y.(p(x) -> p(y))}{\Longrightarrow p(c_1) -> p(c_2), \exists x.\forall y.(p(x) -> p(y))}}{\Longrightarrow \forall y.(p(c_1) -> p(y)), \exists x.\forall y.(p(x) -> p(y))}}{\Longrightarrow \exists x.\forall y.(p(x) -> p(y))}$$

Since the closing rule close cannot be applied to the leaf sequent (nor any of the other closing rules), our only choice is to apply exRight again. This time, we choose the term c_2.

$$\frac{\dfrac{\dfrac{\dfrac{p(c_1) \Longrightarrow \forall y.(p(c_2) -> p(y)), p(c_2), \exists x.\forall y.(p(x) -> p(y))}{p(c_1) \Longrightarrow p(c_2), \exists x.\forall y.(p(x) -> p(y))}}{\Longrightarrow p(c_1) -> p(c_2), \exists x.\forall y.(p(x) -> p(y))}}{\Longrightarrow \forall y.(p(c_1) -> p(y)), \exists x.\forall y.(p(x) -> p(y))}}{\Longrightarrow \exists x.\forall y.(p(x) -> p(y))}$$

Another application of allRight (with the new constant c_3) and then impRight give us:

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(c_1), p(c_2) \Longrightarrow p(c_3), p(c_2), \exists x.\forall y.(p(x) -> p(y))}{p(c_1) \Longrightarrow p(c_2) -> p(c_3), p(c_2), \exists x.\forall y.(p(x) -> p(y))}}{p(c_1) \Longrightarrow \forall y.(p(c_2) -> p(y)), p(c_2), \exists x.\forall y.(p(x) -> p(y))}}{p(c_1) \Longrightarrow p(c_2), \exists x.\forall y.(p(x) -> p(y))}}{\Longrightarrow p(c_1) -> p(c_2), \exists x.\forall y.(p(x) -> p(y))}}{\Longrightarrow \forall y.(p(c_1) -> p(y)), \exists x.\forall y.(p(x) -> p(y))}}{\Longrightarrow \exists x.\forall y.(p(x) -> p(y))}$$

---

[15] There are two reasons for insisting on admissible signatures: one is to have a sufficient supply of new constants for the allRight and exLeft rules. The other is that exRight and allLeft sometimes need to be applied although there is no suitable ground term in the sequent itself, as is the case here.

Finally, we see that the atom $p(c_2)$ appears on both sides of the sequent, so we can apply the close rule

$$\frac{\begin{array}{c} \overline{p(c_1), p(c_2) \Longrightarrow p(c_3), p(c_2), \exists x.\forall y.(p(x) \mathrel{-\!\!>} p(y))} \\ \overline{p(c_1) \Longrightarrow p(c_2) \mathrel{-\!\!>} p(c_3), p(c_2), \exists x.\forall y.(p(x) \mathrel{-\!\!>} p(y))} \\ \overline{p(c_1) \Longrightarrow \forall y.(p(c_2) \mathrel{-\!\!>} p(y)), p(c_2), \exists x.\forall y.(p(x) \mathrel{-\!\!>} p(y))} \\ \overline{p(c_1) \Longrightarrow p(c_2), \exists x.\forall y.(p(x) \mathrel{-\!\!>} p(y))} \\ \overline{\Longrightarrow p(c_1) \mathrel{-\!\!>} p(c_2), \exists x.\forall y.(p(x) \mathrel{-\!\!>} p(y))} \\ \overline{\Longrightarrow \forall y.(p(c_1) \mathrel{-\!\!>} p(y)), \exists x.\forall y.(p(x) \mathrel{-\!\!>} p(y))} \end{array}}{\Longrightarrow \exists x.\forall y.(p(x) \mathrel{-\!\!>} p(y))}$$

This proof tree has only one branch, and a closing rule has been applied to the leaf of this branch. Therefore, all branches are closed, and this is a proof for the formula $\exists x.\forall y.(p(x) \mathrel{-\!\!>} p(y))$.

Example 2.54. We now show an example of a branching proof. In order to save space, we mostly just write the leaf sequents of the branch we are working on.

We take again the minimal type hierarchy. The signature contains two predicate symbols $p, q : \top$, as well as the infinite set of constants $c_1, c_2, \ldots : \top$ and a variable $x{:}\top$. We show the validity of the formula

$$(\exists x.p(x) \mathrel{-\!\!>} \exists x.q(x)) \mathrel{-\!\!>} \exists x.(p(x) \mathrel{-\!\!>} q(x)) \ .$$

We start with the sequent

$$\Longrightarrow (\exists x.p(x) \mathrel{-\!\!>} \exists x.q(x)) \mathrel{-\!\!>} \exists x.(p(x) \mathrel{-\!\!>} q(x))$$

from which the impRight rule makes

$$\exists x.p(x) \mathrel{-\!\!>} \exists x.q(x) \Longrightarrow \exists x.(p(x) \mathrel{-\!\!>} q(x)) \ .$$

We now apply impLeft, which splits the proof tree. The proof tree up to this point is:

$$\frac{\dfrac{\Longrightarrow \exists x.p(x), \exists x.(p(x) \mathrel{-\!\!>} q(x)) \qquad \exists x.q(x) \Longrightarrow \exists x.(p(x) \mathrel{-\!\!>} q(x))}{\exists x.p(x) \mathrel{-\!\!>} \exists x.q(x) \Longrightarrow \exists x.(p(x) \mathrel{-\!\!>} q(x))}}{\Longrightarrow (\exists x.p(x) \mathrel{-\!\!>} \exists x.q(x)) \mathrel{-\!\!>} \exists x.(p(x) \mathrel{-\!\!>} q(x))}$$

On the left branch, we have to choose a term to instantiate one of the existential quantifiers. It turns out that any term will do the trick, so we apply exRight with c_1 on $\exists x.p(x)$, to get

$$\Longrightarrow p(c_1), \exists x.p(x), \exists x.(p(x) \mathrel{-\!\!>} q(x))$$

and then on $\exists x.(p(x) \mathrel{-\!\!>} q(x))$, which gives

$$\Longrightarrow p(c_1), p(c_1) \mathbin{-}\!> q(c_1), \exists x.p(x), \exists x.(p(x) \mathbin{-}\!> q(x)) \ .$$

We now apply impRight to get

$$p(c_1) \Longrightarrow p(c_1), q(c_1), \exists x.p(x), \exists x.(p(x) \mathbin{-}\!> q(x))$$

to which the close rule applies.

On the right branch, we apply exLeft using c_2 as the new constant, which gives us

$$q(c_2) \Longrightarrow \exists x.(p(x) \mathbin{-}\!> q(x)) \ .$$

We now use exRight with the instantiation c_2, giving

$$q(c_2) \Longrightarrow p(c_2) \mathbin{-}\!> q(c_2), \exists x.(p(x) \mathbin{-}\!> q(x)) \ .$$

impRight now produces

$$q(c_2), p(c_2) \Longrightarrow q(c_2), \exists x.(p(x) \mathbin{-}\!> q(x)) \ ,$$

to which close may be applied.

To summarise, for each of !, &, |, $\mathbin{-}\!>$, $\forall$, and $\exists$, there is one rule to handle occurrences in the antecedent and one rule for the succedent. The only "indeterminisms" in the calculus are 1. the order in which the rules are applied, and 2. the instantiations chosen for allRight and exRight.

Both of these indeterminisms are of the kind known as *don't care indeterminism*. What this means is that any choice of rule application order or instantiations can at worst delay (maybe infinitely) the closure of a proof tree. If there is a closed proof tree for a formula, any proof tree can be completed to a closed proof tree. It is not necessary in principle to backtrack over rule applications, there are no "dead ends" in the search space. A calculus with this property is known as *proof confluent*.

It should be noted that an unfortunate choice of applied rules can make the resulting proof much larger in practice, so that it can be worthwhile to remove part of a proof attempt and to start from the beginning.

2.5.5 The Equality Rules

The essence of reasoning about equality is the idea that if one entity equals another, then any occurrence of the first may be replaced by the second. This idea would be expressed by the following (in general incorrect) rule:

$$\mathsf{eqLeftWrong} \ \ \frac{\Gamma, t_1 \doteq t_2, [z/t_1](\phi), [z/t_2](\phi) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2, [z/t_1](\phi) \Longrightarrow \Delta}$$

Unfortunately in the presence of subtyping, things are not quite that easy. Assume for instance a type hierarchy with two types $B \sqsubseteq A$, but $B \neq A$, and

a signature containing constants $a : \to A$ and $b : \to B$, and a predicate $p : B$. If we apply the above rule on the sequent

$$b \doteq a, p(b) \Longrightarrow$$

we get the new "sequent"

$$b \doteq a, p(b), p(a) \Longrightarrow \quad .$$

This is in fact not a sequent, since $p(a)$ is not a formula, because p cannot be applied to a term of static type A.

$$\text{eqLeft} \quad \frac{\Gamma, t_1 \doteq t_2, [z/t_1](\phi), [z/t_2](\phi) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2, [z/t_1](\phi) \Longrightarrow \Delta}$$
$$\text{if } \sigma(t_2) \sqsubseteq \sigma(t_1)$$

$$\text{eqRight} \quad \frac{\Gamma, t_1 \doteq t_2 \Longrightarrow [z/t_2](\phi), [z/t_1](\phi), \Delta}{\Gamma, t_1 \doteq t_2 \Longrightarrow [z/t_1](\phi), \Delta}$$
$$\text{if } \sigma(t_2) \sqsubseteq \sigma(t_1)$$

$$\text{eqLeft}' \quad \frac{\Gamma, t_1 \doteq t_2, [z/t_1](\phi), [z/(A)t_2](\phi) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2, [z/t_1](\phi) \Longrightarrow \Delta}$$
$$\text{with } A := \sigma(t_1)$$

$$\text{eqRight}' \quad \frac{\Gamma, t_1 \doteq t_2 \Longrightarrow [z/(A)t_2](\phi), [z/t_1](\phi), \Delta}{\Gamma, t_1 \doteq t_2 \Longrightarrow [z/t_1](\phi), \Delta}$$
$$\text{with } A := \sigma(t_1)$$

$$\text{eqSymmLeft} \quad \frac{\Gamma, t_2 \doteq t_1 \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2 \Longrightarrow \Delta} \qquad \text{eqClose} \quad \frac{}{\Gamma \Longrightarrow t \doteq t, \Delta}$$

Fig. 2.3. Equality rules

There are two ways to solve this problem. The first way is embodied by the rules eqLeft and eqRight in Fig. 2.3: The static type of the new term t_2 is required to be a subtype of the type of the original t_1. This guarantees that the resulting formula is still well-typed. Indeed, it would have forbidden the erroneous rule application of our example since $\sigma(t_2) \not\sqsubseteq \sigma(t_1)$.

The other solution is to insert a cast. If $t_1 \doteq t_2$ holds, and A is the static type of t_1, then $t_2 \doteq (A)t_2$ also holds, and therefore $t_1 \doteq (A)t_2$, so we can rewrite t_1 to $(A)t_2$, which still has the static type A, so again, the formula remains well-typed. This is what the rules eqLeft' and eqRight' do.[16]

---

[16] As is illustrated in Example 2.58, any application of these two rules may be replaced by a series of applications of other rules, so it would be possible to do

Note that the equation $t_1 \doteq t_2$ has to be on the left of the sequent for all four of these rules. The difference between the Left and Right versions is the position of the formula on which the equation is applied. The only way of handling an equation in the succedent, i.e., of *showing* that an equation holds is to apply other equations on both sides until they become identical, and then to apply eqClose.

In general, one might want to apply equations in both directions, i.e., also rewrite t_2 to t_1. We allow this by the rule eqSymmLeft. Equivalently we could have given variants of the four rewriting rules, that apply equations from right to left, but that would have doubled the number of rules.

Example 2.55. Assume a type hierarchy containing two types $B \sqsubseteq A$ in addition to $\bot$ and $\top$. We need two constants $a : \rightarrow A$ and $b : \rightarrow B$, functions $f : B \rightarrow B$ and $g : A \rightarrow B$, and a variable $x{:}A$. We show the validity of

$$(\forall x.f(g(x)) \doteq g(x) \ \& \ b \doteq g(a)) \rightarrow f(f(b)) \doteq f(g(a)) \ .$$

Starting from the initial sequent

$$\Longrightarrow (\forall x.f(g(x)) \doteq g(x) \ \& \ b \doteq g(a)) \rightarrow f(f(b)) \doteq f(g(a)) \ ,$$

applying impRight and andLeft leads to

$$\forall x.f(g(x)) \doteq g(x), b \doteq g(a) \Longrightarrow f(f(b)) \doteq f(g(a)) \ .$$

We now apply allLeft for the instantiation a. Since we do not need any more instances of the $\forall$ formula, we abbreviate it by "$\dots$" in the rest of this example:

$$\dots, f(g(a)) \doteq g(a), b \doteq g(a) \Longrightarrow f(f(b)) \doteq f(g(a)) \ .$$

Consider the equation $b \doteq g(a)$. The static type of both sides is B, so we could apply the equation in both directions. We would like to rewrite occurrences of $g(a)$ to the smaller term b, so we apply eqSymmLeft to turn the equation around:

$$\dots, f(g(a)) \doteq g(a), g(a) \doteq b \Longrightarrow f(f(b)) \doteq f(g(a)) \ .$$

Now we apply $g(a) \doteq b$ on the left side of the equation $f(g(a)) \doteq g(a)$. As we explained at the end of Section 2.5.2, this is done by marking the place where the equality should be applied by a variable z and "pulling out" the term t_1 into a substitution, i.e., $(f(z) \doteq g(a))[z/t_1]$. In other words, we apply eqLeft with

$$t_1 = g(a) \qquad t_2 = b \qquad \phi = f(z) \doteq g(a)$$

to get

---

without them. Still, it is sometimes convenient to have them, since they allow to do all equality reasoning first, possibly inserting casts, and taking care of the type reasoning later.

$$\dots, f(g(a)) \doteq g(a), g(a) \doteq b, f(b) \doteq g(a) \Longrightarrow f(f(b)) \doteq f(g(a)) \ .$$

The next step is to apply the new equation $f(b) \doteq g(a)$ on the occurrence of $f(b)$ in the succedent, i.e., we apply eqRight with

$$t_1 = f(b) \qquad t_2 = g(a) \qquad \phi = f(z) \doteq f(g(a))$$

to get

$$\dots, f(g(a)) \doteq g(a), g(a) \doteq b, f(b) \doteq g(a)$$
$$\Longrightarrow f(g(a)) \doteq f(g(a)), f(f(b)) \doteq f(g(a))$$

which can be closed using the eqClose rule.

The eqLeft′/eqRight′ rules introduce casts which can only be treated by using some additional rules. We therefore postpone an example of their use to the following section.

The equality rules allow much more freedom in their application than the previously shown rules in Fig. 2.2. As a general guideline, it is often best to apply equations in the direction that makes terms smaller or simpler, provided this is allowed by the types.

It should be mentioned at this point that the equality rules in the implementation of the KeY system are organised in a slightly different way. Instead of letting the user decide between the rules eqLeft and eqLeft′, or between eqRight and eqRight′ for an occurrence in the succedent, the system checks whether $\sigma(t_2) \sqsubseteq \sigma(t_1)$. If this is the case, no cast is needed and eqLeft, resp. eqRight is applied, otherwise a cast is inserted, corresponding to an application of eqLeft′, resp. eqRight′. This combined behaviour is achieved by a rule named applyEq (see Fig. 4.5).

2.5.6 The Typing Rules

The remaining rules, shown in Fig. 2.4, all concern type casts and type predicates. In problems where all terms are of the same type, and no casts or type predicates occur, these rules are not needed.

Given two terms $t_1 \in \mathrm{Trm}_A$ and $t_2 \in \mathrm{Trm}_B$ of static types A and B, the first rule allows deriving $t_2 \sqsubseteq A$ and $t_1 \sqsubseteq B$. Why is this allowed? Given some model $\mathcal{M}$, and variable assignment β, if $\mathcal{M}, \beta \models t_1 \doteq t_2$, then $\mathrm{val}_{\mathcal{M},\beta}(t_1) = \mathrm{val}_{\mathcal{M},\beta}(t_2)$. Therefore, the dynamic types of the terms' values are also equal: $\delta(\mathrm{val}_{\mathcal{M},\beta}(t_1)) = \delta(\mathrm{val}_{\mathcal{M},\beta}(t_2))$. Now, the dynamic type of each term is a subtype of the static type of the term. Since the dynamic types are the same, we additionally know that the dynamic type of each term is a subtype of the static type of *the other* term. Hence, $\mathcal{M}, \beta \models t_2 \sqsubseteq A$ and $\mathcal{M}, \beta \models t_1 \sqsubseteq B$. In combination with the typeStatic and typeGLB rules, we can go on by deriving $t_1 \sqsubseteq A \sqcap B$ and $t_2 \sqsubseteq A \sqcap B$.

The typeAbstract rule handles type predicate literals for abstract types. The underlying reasoning is that if the dynamic type of a value cannot be

typeEq
$$\frac{\Gamma, t_1 \doteq t_2, t_2 \sqsubseteq \sigma(t_1), t_1 \sqsubseteq \sigma(t_2) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2 \Longrightarrow \Delta}$$

typeGLB
$$\frac{\Gamma, t \sqsubseteq A, t \sqsubseteq B, t \sqsubseteq A \sqcap B \Longrightarrow \Delta}{\Gamma, t \sqsubseteq A, t \sqsubseteq B \Longrightarrow \Delta}$$

typeStatic
$$\frac{\Gamma, t \sqsubseteq \sigma(t) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

typeAbstract
$$\frac{\Gamma, t \sqsubseteq A, t \sqsubseteq B_1 \mid \cdots \mid t \sqsubseteq B_k \Longrightarrow \Delta}{\Gamma, t \sqsubseteq A \Longrightarrow \Delta}$$
with $A \in \mathcal{T}_a$ and $B_1, \ldots, B_k$ the direct subtypes of A

castAddLeft
$$\frac{\Gamma, [z/t](\phi), t \sqsubseteq A, [z/(A)t](\phi) \Longrightarrow \Delta}{\Gamma, [z/t](\phi), t \sqsubseteq A \Longrightarrow \Delta}$$
where $A \sqsubseteq \sigma(t)$.

castAddRight
$$\frac{\Gamma, t \sqsubseteq A \Longrightarrow [z/(A)t](\phi), [z/t](\phi), \Delta}{\Gamma, t \sqsubseteq A \Longrightarrow [z/t](\phi), \Delta}$$
where $A \sqsubseteq \sigma(t)$.

castDelLeft
$$\frac{\Gamma, [z/t](\phi), [z/(A)t](\phi) \Longrightarrow \Delta}{\Gamma, [z/(A)t](\phi) \Longrightarrow \Delta}$$
where $\sigma(t) \sqsubseteq A$.

castDelRight
$$\frac{\Gamma \Longrightarrow [z/t](\phi), [z/(A)t](\phi), \Delta}{\Gamma \Longrightarrow [z/(A)t](\phi), \Delta}$$
where $\sigma(t) \sqsubseteq A$.

castTypeLeft
$$\frac{\Gamma, (A)t \sqsubseteq B, t \sqsubseteq A, t \sqsubseteq B \Longrightarrow \Delta}{\Gamma, (A)t \sqsubseteq B, t \sqsubseteq A \Longrightarrow \Delta}$$

castTypeRight
$$\frac{\Gamma, t \sqsubseteq A \Longrightarrow t \sqsubseteq B, (A)t \sqsubseteq B, \Delta}{\Gamma, t \sqsubseteq A \Longrightarrow (A)t \sqsubseteq B, \Delta}$$

closeSubtype
$$\frac{}{\Gamma, t \sqsubseteq A \Longrightarrow t \sqsubseteq B, \Delta}$$
with $A \sqsubseteq B$

closeEmpty
$$\frac{}{\Gamma, t \sqsubseteq \bot \Longrightarrow \Delta}$$

Fig. 2.4. Typing rules

equal to an abstract type, so if $t \sqsubseteq A$ holds for an abstract type A, then $t \sqsubseteq B$ holds for some subtype B of A. Since we require type hierarchies to be finite, we can form the disjunction $t \sqsubseteq B_1 \mid \cdots \mid t \sqsubseteq B_k$ for all direct subtypes B_i of A.[17] If one of the direct subtypes B_i is itself abstract, the rule can be applied again on $t \sqsubseteq B_i$.

The castAdd, castType, and castDel rules can be used to close proof trees that involve formulae with type casts. More specifically, we need to deal with the situation that a branch can almost be closed, using for instance close or eqClose, but the involved formulae or terms are not quite equal, they differ by some of the casts. In general, the sequent also contains type predicates that allow to decide whether the casts are "successful" or not.

---

[17] B is a direct subtype of A if A and B are distinct types, $B \sqsubseteq A$, and there is no type C that is distinct from A and B with $B \sqsubseteq C \sqsubseteq A$, Def. 3.1.

The basic observation is that if $t \sqsubseteq A$ holds, then the cast $(A)t$ does not change the value of t, so $(A)t \doteq t$ also holds. It is tempting to introduce rules like the following, which allows to remove casts in such situations:

$$\text{wrongCastDelLeft} \quad \frac{\Gamma, [z/(A)t](\phi), t \sqsubseteq A, [z/t](\phi) \Longrightarrow \Delta}{\Gamma, [z/(A)t](\phi), t \sqsubseteq A \Longrightarrow \Delta}$$

Unfortunately, the new formula $[z/t](\phi)$, in which the cast was removed, is possibly no longer well-typed: In general, the static type of t is a supertype of that of $(A)t$. Our solution to this problem is to *add* casts to the involved terms or formulae until they become equal. This is the purpose of the castAdd rules.

There are also castDel rules to delete casts, but these are only available if the static type of a term is a subtype of the type being cast to. In that case, the cast is obviously redundant, and removing it preserves the well-typedness of terms.

The two castType rules can be considered valid special cases of our wrongCastDelLeft rule: If we know $t \sqsubseteq A$, then we may remove the cast in $(A)t \sqsubseteq B$ to obtain $t \sqsubseteq B$. There is no problem with the static types here, since the type predicate $\sqsubseteq B$ may be applied to terms of arbitrary type. These rules are occasionally needed to derive the most specific type information possible about the term t.

We now illustrate these rules in some examples.

Example 2.56. We start by the formula from Example 2.33. In any type hierarchy that contains some type A, and a signature with a variable $x{:}\top$ and a constant $c{:}\top$, we show the validity of

$$\forall x.((A)x \doteq x \rightarrow x \sqsubseteq A) \ .$$

The initial sequent is

$$\Longrightarrow \forall x.((A)x \doteq x \rightarrow x \sqsubseteq A) \ ,$$

on which we apply the allRight and impRight to get

$$(A)c \doteq c \Longrightarrow c \sqsubseteq A \ .$$

The static type of c is $\top$, and the static type of $(A)c$ is A. We apply the typeEq rule to get

$$(A)c \doteq c, c \sqsubseteq A, (A)c \sqsubseteq \top \Longrightarrow c \sqsubseteq A \ .$$

Since $c \sqsubseteq A$ appears on both sides, this can be closed using the close rule.

Example 2.57. With the same type hierarchy and signature as the previous example, we now show the converse implication:

$$\forall x.(x \sqsubseteq A \rightarrow (A)x \doteq x) \ .$$

Again, we apply allRight and impRight on the initial sequent, to obtain

$$c \sqsubseteq A \Longrightarrow (A)c \doteq c \ .$$

We now apply castAddRight with

$$t = c \qquad \text{and} \qquad \phi = (A)c \doteq z$$

to obtain

$$c \sqsubseteq A \Longrightarrow (A)c \doteq (A)c, (A)c \doteq c \ ,$$

which can be closed using eqClose.

Example 2.58. Here is a more complicated example of type reasoning. We return to the type hierarchy from Example 2.6, p. 24. Remember that

$$\text{AbstractList} = \text{AbstractCollection} \sqcap \text{List} \ .$$

The following functions are used:

$$ord : \text{AbstractCollection} \rightarrow \text{AbstractList}$$
$$rev : \text{List} \rightarrow \text{List}$$

Maybe *ord* takes a collection and puts it into some order, whereas *rev* reverses a list. We also use a constant *a*:AbstractCollection. The problem is to show the validity of

$$ord(a) \doteq a \rightarrow rev(ord(a)) \doteq rev((\text{List})a) \ .$$

In this example, we silently omit some formulae from sequents, if they are not needed any more, to make it easier to follow the development. After applying impRight on the initial sequent, we get

$$ord(a) \doteq a \Longrightarrow rev(ord(a)) \doteq rev((\text{List})a) \ . \qquad (*)$$

Next, we would like to rewrite $ord(a)$ to a in the succedent. However, the static type of a is AbstractCollection, which is not a subtype of the static type of $ord(a)$, namely AbstractList. Therefore, we must use eqRight$'$, which introduces a cast and gives us:

$$ord(a) \doteq a \Longrightarrow rev((\text{AbstractList})a) \doteq rev((\text{List})a) \ .$$

Our goal must now be to make the two casts in the succedent equal. To deduce more information about the type of a, we apply typeEq on the left to get

$$a \sqsubseteq \text{AbstractList} \Longrightarrow rev((\text{AbstractList})a) \doteq rev((\text{List})a)$$

(we omit the other, uninteresting formula $ord(a) \sqsubseteq \text{AbstractCollection}$). Now, how do we replace the cast to List by a cast to AbstractList? We use a combination of two rules: First, we apply castAddRight to insert a cast:

$$a \sqsubseteq \text{AbstractList} \Longrightarrow rev((\text{AbstractList})a) \doteq rev((\text{List})(\text{AbstractList})a) \ .$$

Since $\text{AbstractList} \sqsubseteq \text{List}$, the outer cast has become redundant, so we use castDelRight to remove it:

$$a \sqsubseteq \text{AbstractList} \Longrightarrow rev((\text{AbstractList})a) \doteq rev((\text{AbstractList})a) \ .$$

This sequent can be closed using eqClose.

It turns out that applications of the eqRight′/eqLeft′ rules can always be replaced by sequences of applications of the other rules. They were only added because they are sometimes more convenient. We demonstrate this by showing an alternative way of proceeding from the sequent (∗) above. We first apply the typeEq rule, which gives us

$$ord(a) \doteq a, a \sqsubseteq \text{AbstractList} \Longrightarrow rev(ord(a)) \doteq rev((\text{List})a) \ .$$

We can then use castAddRight on the right side of the equation in the antecedent, yielding

$$ord(a) \doteq (\text{AbstractList})a, a \sqsubseteq \text{AbstractList} \Longrightarrow rev(ord(a)) \doteq rev((\text{List})a) \ .$$

Now, the static types are the same on both sides and we can use eqRight to obtain

$$a \sqsubseteq \text{AbstractList} \Longrightarrow rev((\text{AbstractList})a) \doteq rev((\text{List})a) \ .$$

From this sequent, we continue as before.

Example 2.59. Using the type hierarchy from Example 2.6 once again, a variable l : List and a constant c:List, we show validity of

$$\forall l.l \sqsubseteq \text{ArrayList} \ .$$

This is of course due to the fact that ArrayList is the top-most non-abstract subtype of List. Starting from

$$\Longrightarrow \forall l.l \sqsubseteq \text{ArrayList} \ ,$$

we apply the rule allRight to obtain

$$\Longrightarrow c \sqsubseteq \text{ArrayList} \ .$$

We can use the typeStatic rule for c to get

$$c \sqsubseteq \text{List} \Longrightarrow c \sqsubseteq \text{ArrayList} \ .$$

Now typeAbstract produces

$$c \sqsubseteq \text{AbstractList} \Longrightarrow c \sqsubseteq \text{ArrayList} \ ,$$

since AbstractList is the only direct subtype of the abstract type List. Since AbstractList is also abstract, we apply typeStatic again, to get

$$c \sqsubseteq \text{ArrayList} \Longrightarrow c \sqsubseteq \text{ArrayList} \ ,$$

which can be closed by close.

Example 2.60. In the type hierarchy from Example 2.6, using a variable i : int, and constants c:int and *null*:Null, we show

$$!\, \exists i. i \doteq \mathit{null} \ .$$

On the initial sequent

$$\Longrightarrow\ !\, \exists i. i \doteq \mathit{null} \ ,$$

we apply notRight to obtain

$$\exists i. i \doteq \mathit{null} \Longrightarrow\ ,$$

and exRight, which gives

$$c \doteq \mathit{null} \Longrightarrow\ .$$

Using typeStatic and typeEq for c produces

$$c \doteq \mathit{null}, c \sqsubseteq \text{int}, c \sqsubseteq \text{Null} \Longrightarrow\ .$$

The intersection of int and Null is the empty type, so we can use typeGLB to derive

$$c \doteq \mathit{null}, c \sqsubseteq \bot \Longrightarrow\ ,$$

which can be closed using closeEmpty.

To summarise, the general idea of type reasoning is to start by identifying the interesting terms t. For these terms, one tries to derive the most specific type information, i.e., a type predicate literal $t \sqsubseteq A$ where the type A is as small as possible with respect to $\sqsubseteq$, by using typeStatic and typeEq, etc. Then, add a cast to the most specific known type in front of the interesting occurrences of t. On the other hand, delete redundant casts using the castDel rules. Sometimes, branches can be closed due to contradictory type information using closeSubtype and closeEmpty.

2.6 Soundness, Completeness

At first sight, the rules given in Section 2.5 might seem like a rather haphazard collection. But in fact, they enjoy two important properties. First, it is not possible to close a proof tree for a formula that is not logically valid. This is known as *soundness*. Second, if a formula is logically valid, then there is always a proof for it. This property is known as *completeness*. These two properties are so important that we state them as a theorem.

Theorem 2.61. *Let a fixed type hierarchy and an admissible signature be given. Then any formula ϕ is logically valid if and only if there is a sequent proof for ϕ constructed according to Def. 2.50.*

A proof of this result has been given by Giese [2005].

It is important to note that the theorem does not state soundness and completeness for our notion of validity with respect to partial models. This issue is discussed further in Section 2.7.

Soundness is much more important than completeness, in the sense that more harm is usually done if a wrong statement is considered correct, than if a valid statement cannot be shown. For instance, if a proof for the correctness of a piece of critical software is produced, and the software is used in the belief that it is correct, the consequences might be catastrophic.

On the other hand, not being able to prove the correctness of a correct piece of software with a given method might delay its deployment. Maybe the verification can be done by some other method. Maybe the formal proof is not considered to be crucial.

In practice, however, when a proof attempt fails, it is good to know that there can be only two reasons: either the statement to be shown is not valid, or one has not looked hard enough for a proof. The possibility that the statement is valid, but no proof exists, would make the whole situation more confusing.

Since we ultimately intend to use our logic and calculus on a computer, where a program should help us to find the proofs, let us consider some of the computational aspects of our calculus.

We already mentioned that the calculus is proof confluent: If a formula ϕ is valid, then any proof tree for ϕ can be completed to a closed proof. No rule application can lead into a "dead end". However, it is still possible to expand a proof tree for a valid formula indefinitely without finding a closed proof, just by regularly performing the "wrong" rule applications.

The good news is that it *is* possible to apply rules systematically in such a way that a closed proof is eventually found if it exists. This leads to a "computational" version of the previous version:

Theorem 2.62. *Let a fixed type hierarchy and an admissible signature be given. There is a program with the following property: if it is given a formula as input, it terminates stating the validity of the input formula if and only if that formula is logically valid.*

What if the formula is not valid? In general, the program will search indefinitely, and never give any output. It is possible to show that this must be so: It is a property of our logic that there can be no program that terminates on both valid and invalid formulae and correctly states whether the input is valid.

The technical way of describing this situation is to say that the validity of formulae in our logic is *undecidable*. This means that there is no program that terminates on all inputs and answers the question of validity.

More precisely, validity is *semi-decidable*, which means that there *is* a program that at least gives a positive answer for valid formulae. We equivalently say that the set of valid formulae is *recursively enumerable*, which means that it is possible to write a program that prints a list of all valid formulae.

For the practical use of a theorem proving program, this means that if the program runs for a very long time, there is no way of knowing whether the statement we are trying to prove is wrong, or whether we just have to wait for an even longer time.

There are logics (propositional logic and some modal logics) that have a better behaviour in this respect: there are theorem provers which terminate on all input and answer whether the input is a valid formula. However, these logics are a lot less expressive, and therefore not suited for detailed descriptions of complex software systems. Any logic that is expressive enough for that purpose has an undecidable validity problem.

The interested reader can find much more information on all aspects of the mechanisation of reasoning in the Handbook of Automated Reasoning edited by Robinson and Voronkov [2001].

2.7 Incompleteness

The soundness and completeness properties stated in the previous section do not apply to validity relative to some partial model. Indeed, it is hard to give general results about relative validity since any set of operations with fixed interpretation would require its own set of additional rules.

In this section, we discuss a particularly important case, namely the operations of addition and multiplication on the natural numbers. We do not include subtraction or division, since they can be expressed in terms of addition and multiplication.

Let us assume that the type hierarchy contains a sort N and the signature contains function symbols

$$zero : \rightarrow N$$
$$succ : N \rightarrow N$$
$$plus : N, N \rightarrow N$$
$$times : N, N \rightarrow N$$

The partial model that interests us is defined as

$$
\begin{aligned}
\mathcal{T}_0 &:= \{N\} \\
\mathcal{D}_0 &:= \mathbb{N} = \{0, 1, 2, \ldots\} \\
\delta_0(x) &:= N \quad \text{for all } x \in \mathcal{D}_0 \\
D_0(zero) &:= \{()\} \\
D_0(succ) &:= \mathbb{N} \\
D_0(plus) &:= D_0(times) := \mathbb{N} \times \mathbb{N} \\
\mathcal{I}_0(zero)() &:= 0 \\
\mathcal{I}_0(succ)(x) &:= x + 1 \\
\mathcal{I}_0(plus)(x, y) &:= x + y \\
\mathcal{I}_0(times)(x, y) &:= xy
\end{aligned}
$$

We call this model "arithmetic". Note that our domain $\mathcal{D}_0$ now contains all the mathematical integers, and not only the JAVA integers as in previous examples. With this signature and partial model, individual natural numbers can be expressed as *zero* for 0, *succ(zero)* for 1, *succ(succ(zero))* for 2, etc. The operations of addition and multiplication are sufficient to express many interesting mathematical concepts. For instance the following formula with free variable x expresses that x is either zero, or one, or a prime number:

$$\forall y.\forall z.(times(y, z) \doteq x \mathrel{-\!\!>} y \doteq x \mid z \doteq x) \ ,$$

and the following the fact that there are infinitely many prime numbers:[18]

$$\forall x.\exists u.\forall y.\forall z.(times(y, z) \doteq plus(x, u) \mathrel{-\!\!>} y \doteq plus(x, u) \mid z \doteq plus(x, u)) \ .$$

Due to their expressivity, these basic operations on numbers are among the first things one might want to fix in a partial model. The bad news is that there can be no complete calculus for validity with respect to arithmetic.

Theorem 2.63. *There is no set of sequent rules suitable for mechanisation[19] such that a formula ϕ is valid w.r.t arithmetic if and only if there is a closed sequent proof for ϕ using these rules.*

Indeed, there is no program with the following property: if it is given a formula as input, it terminates stating the validity of the input formula if and only if that formula is logically valid w.r.t. arithmetic.

This is essentially the famous Incompleteness Theorem of Gödel [1931]. It means that if we want the expressive power of arithmetic, there are always some theorems that are true, but cannot be shown in our calculus. Another way of expressing this is that any sound calculus is necessarily incomplete. Therefore, one also calls a logic with this property incomplete.

---

[18] It expresses that for any x, one can find a u, such that $x + u$ is prime, in other words there are primes of arbitrary magnitude.

[19] By this, we mean that the rules may not perform operations which are so complicated that it cannot be checked by a computer program whether a given rule application is correct.

$$\text{succZero} \ \frac{\Gamma, \forall n.\,!\,zero \doteq succ(n) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\text{succEq} \ \frac{\Gamma, \forall m.\forall n.(succ(m) \doteq succ(n) \mathbin{-}> m \doteq n) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\text{pluZero} \ \frac{\Gamma, \forall n.plus(zero, n) \doteq n \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\text{plusSucc} \ \frac{\Gamma, \forall m.\forall n.plus(succ(m), n) \doteq succ(plus(m, n)) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\text{timesZero} \ \frac{\Gamma, \forall n.times(zero, n) \doteq zero \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\text{timesSucc} \ \frac{\Gamma, \forall m.\forall n.times(succ(m), n) \doteq plus(n, times(m, n)) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

natInduct
$$\frac{\Gamma \Longrightarrow [n/zero](\phi), \Delta \qquad \Gamma \Longrightarrow \forall n.(\phi \mathbin{-}> [n/succ(n)](\phi)), \Delta \quad \Gamma, \forall n.\phi \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

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

Fig. 2.5. Rules for arithmetic, using variables $m{:}N, n{:}N$

In practice, the way of dealing with this problem is to add a number of rules to the calculus that capture the behaviour of *plus* and *times*, as well as one particular rule called the *induction rule* (see also Chapter 11). For instance, we can add the rules in Fig. 2.5. Most of these rules just add simple properties of the involved operations to the sequent. The interesting rule is natInduct: It expresses that if you can show that a statement holds for *zero*, and that if it holds for some number n, it also holds for the next number $succ(n)$, then it must hold for all numbers.

These rules are still subject to the incompleteness theorem. But it turns out that using these rules, it is possible to prove almost any arithmetical statement that occurs *in practice*. Virtually any theorem about natural numbers that occurs in mathematics is ultimately proven using some variation of these few rules.[20]

---

[20] There are exceptions to this. For instance, there is a number theoretical theorem known as Goodstein's theorem that can only be proven by using more powerful methods [Goodstein, 1944, Kirby and Paris, 1982].

It is interesting to note that many of the data structures appearing in computer programs, like for instance lists, strings, or trees have the same properties. In fact their behaviour can be encoded using numbers, and on the other hand, they can be used to simulate arithmetic. Therefore, for these data types the same observation holds, namely that validity relative to them makes the logic incomplete, but adding an appropriate induction rule (structural induction) allows proving almost all *practically interesting* statements. Induction is discussed in much greater detail in Chapter 11.

Fortunately however, this is in a sense the *only* kind of incompleteness one has to confront: as explained in Section 3.4.2, the calculus used in KeY to reason about programs is complete *relative to arithmetic*, meaning that it is possible to prove any valid statement about programs if one can prove statements about arithmetic. The observation about practically interesting statements applies also here, which means that despite the theoretical incompleteness, we can prove almost all interesting statements about almost all interesting programs.

3

Dynamic Logic

by

Bernhard Beckert
Vladimir Klebanov
Steffen Schlager

3.1 Introduction

In the previous chapter, we have introduced a variant of classical predicate logic that has a rich type system and a sequent calculus for that logic. This predicate logic can easily be used to describe and reason about data structures, the relations between objects, the values of variables—in short: about the states of (Java) programs.

Now, we extend the logic and the calculus such that we can describe and reason about the behaviour of programs, which requires to consider not just one but several program states. As a trivial example, consider the Java statement x++;. We want to be able to express that this statement, when started in a state where x is zero, terminates in a state where x is one.

We use an instance of dynamic logic (DL) [Harel, 1984, Harel et al., 2000, Kozen and Tiuryn, 1990, Pratt, 1977] as the logical basis of the KeY system's software verification component [Beckert, 2001]. The principle of DL is the formulation of statements about program behaviour by integrating programs and formulae within a single language. To this end, the operators (modalities) $\langle p \rangle$ and $[p]$ can be used in formulae, where p can be any sequence of legal Java Card statements (i.e., DL is a multi-modal logic). These operators that refer to the final state of p 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 $\mathtt{x} \doteq 0 \to \langle \mathtt{x++} \rangle (\mathtt{x} \doteq 1)$.

In general, there can be more than one final state because the programs can be non-deterministic; but here, since Java Card programs are deterministic, there is exactly one such 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). "Deterministic" here means

that a program, for the same initial state and the some inputs, always has the same behaviour—in particular, the same final state (if it terminates) and the same outputs. When we do not (exactly) know what the initial state resp. the inputs are, we may not know what (exactly) the behaviour is. But that does not contradict determinism of the programming language JAVA CARD.

Deduction in DL, and in particular in JAVA CARD DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of JAVA ($\Rightarrow$ Sect. 3.4.5).

Dynamic Logic and Hoare Logic

Dynamic logic can be seen as an extension of Hoare logic. The DL formula $\phi \mathbin{-}\!\!> [p]\psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$. But in contrast to Hoare logic, the set of formulae of DL is closed under the usual logical operators: In Hoare logic, the formulae ϕ and ψ are pure first-order formulae, whereas in DL they can contain programs.

DL allows to involve programs in the descriptions ϕ resp. ψ of states. For example, using a program, it is easy to specify that a data structure is not cyclic, which is impossible in pure first-order logic. Also, all JAVA constructs are available in our DL for the description of states (including `while` loops and recursion). It is, therefore, not necessary to define an abstract data type *state* and to represent states as terms of that type; instead DL formulae can be used to give a (partial) description of states, which is a more flexible technique and allows to concentrate on the relevant properties of a state.

Structure of This Chapter

The structure of this chapter is similar to that of the chapter on first-order logic. We first define syntax and semantics of our JAVA CARD dynamic logic in Sections 3.2 and 3.3. Then, in Section 3.4–3.9, we present the JAVA CARD DL calculus, which is used in the KeY system for verifying JAVA CARD programs. Section 3.4 gives an overview, while Sect. 3.5–3.9 describe the main components of the calculus: non-program rules (Sect. 3.5), rules for reducing JAVA CARD programs to combinations of state updates and case distinctions (Sect. 3.6), rules for handling loops with the help of loop invariants (Sect. 3.7), rules for handling method calls with the help of method contracts (Sect. 3.8), and the simplification and normalisation of state updates (Sect. 3.9). Finally, Sect. 3.10 discusses related work.

In addition, some important aspects of JAVA CARD DL and the calculus are discussed in other chapters of this book, including the first-order part (Chapter 2), proof construction and search (Chapter 4), induction (Chapter 11), handling integers (Chapter 12), and handling the particularities of JAVA CARD such as JAVA CARD's transaction mechanism (Chapter 9). An introduction to using the implementation of the calculus in the KeY system is given in Chapter 10.

3.2 Syntax

In general, a dynamic logic is constructed by extending some non-dynamic logic with parameterised 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 the typed first-order predicate logic described in Chapter 2. Not surprisingly, the programming language we consider is JAVA CARD, i.e., the programs p within the modal operators are in our case JAVA CARD programs. The logic we define in this section is called JAVA CARD Dynamic Logic or, for short, JAVA CARD DL.

The structure of this section follows the structure of Chapter 2. Sect. 3.2.1 defines the notions of type hierarchy and signature for JAVA CARD DL. However, we are more restrictive here than in the corresponding definitions of Chapter 2 (Def. 2.1 and 2.8) since we want the JAVA CARD DL type hierarchy to reflect the type hierarchy of JAVA CARD. A JAVA CARD DL type hierarchy must, e.g., always contain a type Object. Then, we define the syntax of JAVA CARD DL which consists of terms, formulae, and a new category of expressions called *updates* (Sect. 3.2). In the subsequent Sect. 3.3, we present a model-theoretic semantics of JAVA CARD DL based on Kripke structures.

3.2.1 Type Hierarchy and Signature

We start with the definition of the underlying type hierarchies and the signatures of JAVA CARD DL. Since the logic we define is tailored to the programming language JAVA CARD, we are only interested in type hierarchies containing a set of certain types that are part of every JAVA CARD program. First, we define a direct subtype relation that is needed in the subsequent definition.

Definition 3.1 (Direct subtype). *Assume a first-order logic type system* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$. *Then the* direct subtype relation $\sqsubseteq^0 \subseteq \mathcal{T} \times \mathcal{T}$ *between two types* $A, B \in \mathcal{T}$ *is defined as:*

$$A \sqsubseteq^0 B$$
$$\textit{iff}$$
$$A \sqsubseteq B \textit{ and } A \neq B \textit{ and}$$
$$C = A \textit{ or } C = B \textit{ for any } C \in \mathcal{T} \textit{ with } A \sqsubseteq C \textit{ and } C \sqsubseteq B.$$

Intuitively, A is a direct subtype of B if there is no type C that is between A and B.

Definition 3.2 (JAVA CARD DL type hierarchy). *A* JAVA CARD DL *type hierarchy is a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ ($\Rightarrow$ *Def. 2.1*) *such that:*

- $\mathcal{T}_d$ *contains (at least) the types:*

       integerDomain, boolean, Object, Error, Exception,
RuntimeException, NullPointerException, ClassCastException,
ExceptionInInitializerError, ArrayIndexOutOfBoundsException,
ArrayStoreException, ArithmeticException, Null;

- $\mathcal{T}_a$ *contains (at least) the types:* integer, byte, short, int, long, char, *Serializable, Cloneable, Throwable;*
- *if* $A \sqsubseteq$ Object, *then* Null $\sqsubseteq A$ *for all* $A \neq \bot \in \mathcal{T}$;
- integerDomain $\sqsubseteq^0 A$ *and* $A \sqsubseteq^0$ integer *for all* $A \in \{\text{byte}, \text{short}, \text{int},$ long, char$\}$;
- $\bot \sqsubseteq^0$ integerDomain;
- $\bot \sqsubseteq^0$ Null;
- $\bot \sqsubseteq^0$ boolean;
- $A \sqcap B = \bot$ *for all* $A \in \{$integerDomain, integer, byte, short, int, long, char, boolean$\}$ *and* $B \sqsubseteq$ Object.

In the remainder of this chapter, with type hierarchy *we always mean a* JAVA CARD *DL type hierarchy, unless stated otherwise.*

A JAVA CARD DL type hierarchy is a type hierarchy containing the types that are built into JAVA CARD like boolean, the root reference type Object, and the type Null, which is a subtype of all reference types (Null exists implicitly in JAVA CARD). As in the first-order case, the type hierarchy contains the special types $\top$ and $\bot$ ($\Rightarrow$ Def. 2.1). Moreover, it contains a set of abstract and dynamic (i.e., non-abstract) types reflecting the set of JAVA CARD interfaces and classes necessary when dealing with arrays. These are Cloneable, Serializable, Throwable, and some particular sub-sorts of the latter which are the possible exceptions and errors that may occur during initialisation and when working with arrays.

    Finally, a type hierarchy includes the types boolean, byte, short, int, long, and char representing the corresponding primitive JAVA CARD types. Note, that these types (except for boolean) are abstract and are subtypes of the likewise abstract type integer. The common subtype of these types is the non-abstract type integerDomain, thus satisfying the requirement that any abstract type must have a non-abstract subtype. Later we define that the domain of integerDomain is the (infinite) set $\mathbb{Z}$ of integer numbers. Since the domain of a type by definition ($\Rightarrow$ Def. 2.20) includes the domains of its subtypes, all the abstract supertypes of integerDomain share the common domain $\mathbb{Z}$. The typing of the usual functions on the integers, like e.g., addition, is defined as integer, integer $\rightarrow$ integer.

Reasons for the Complicated Integer Type Hierarchy

The reasons behind the somewhat complicated looking integer type hierarchy are twofold. First, we want to have mathematical integers in the logic instead of integers with finite range as in JAVA CARD (the advantage of this decision is explained in Chapter 12). As a consequence, the

type `integer` is defined. Second, we need a dedicated type for each primitive JAVA CARD integer type. That is necessary for a correct handling of `ArrayStoreException`s in the calculus which requires the mapping between types in JAVA CARD and sorts in JAVA CARD DL to be an injection.

The reason why we introduce the common subtype `integerDomain` is that `integer`, `short`, ..., `char` are supposed to share the same domain, namely the integer numbers $\mathbb{Z}$. As a consequence of Def. 2.20, which requires that any domain element $d \in \mathcal{D}$ has a unique dynamic type $\delta(d)$, the only possibility to obtain the same domain for several types is to declare these types as abstract and introduce a common non-abstract subtype holding the domain.

The definition of type hierarchies (Def. 3.2) partly fixes the subtype relation. It requires that type Null is a common subtype of all subtypes of Object (except $\bot$). That is necessary to correctly reflect the JAVA CARD reference type hierarchy. Besides reference types, JAVA CARD has primitive types (e.g., `boolean`, `byte`, or `int`) which have no common sub- or supertype with any reference type. Def. 3.2 guarantees that we only consider type hierarchies where there is no common subtype (except $\bot$, which does not exist in JAVA CARD) of primitive and reference types, thus correctly reflecting the type hierarchy of the JAVA CARD language.

However, Def. 3.2 does not fix the set of user-defined types and the subtype relation between them. A JAVA CARD type hierarchy can contain additional user-defined types, e.g., those types that are declared in a concrete JAVA CARD program ($\Rightarrow$ Def. 3.10).

Fig. 3.1 shows the basic type hierarchy without any user-defined types. Due to space restrictions the types `short`, `int`, and the built-in API reference types like Serializable, Cloneable, Exception, etc. are omitted from the figure. Abstract types are written in italics ($\bot$ is of course also abstract). The subtype relation $A \sqsubseteq B$ is illustrated by an arrow from A to B (reflexive arrows are omitted).

Example 3.3. Consider the type hierarchy in Fig. 3.2 which is an extension of the first-order type hierarchy from Example 2.6. The types AbstractCollection, List, AbstractList, ArrayList, and the array type Object[] are user-defined, i.e., are not required to be contained in any type hierarchy. As we define later, any array type must be a subtype of the built-in types Object, Serializable, and Cloneable.

Due to space restrictions some of the built-in types ($\Rightarrow$ Fig. 3.1) are omitted.

As in Chapter 2, 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
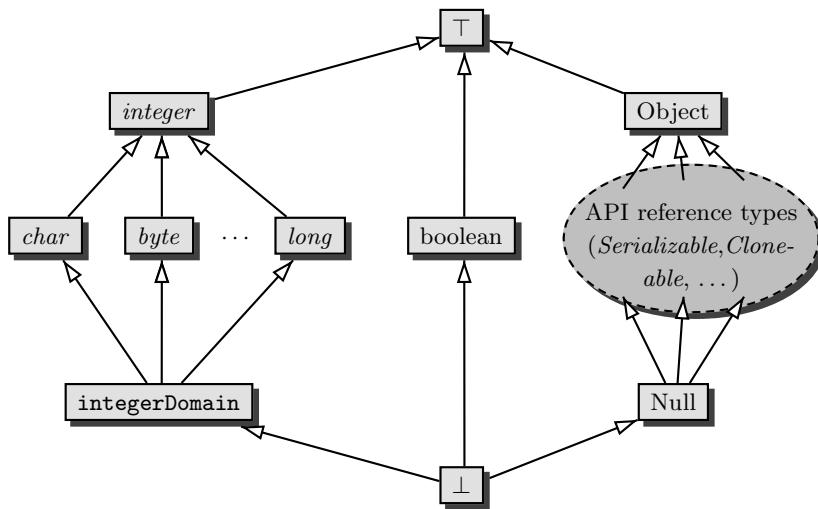
Fig. 3.1. Basic Java Card DL type hierarchy without user-defined types
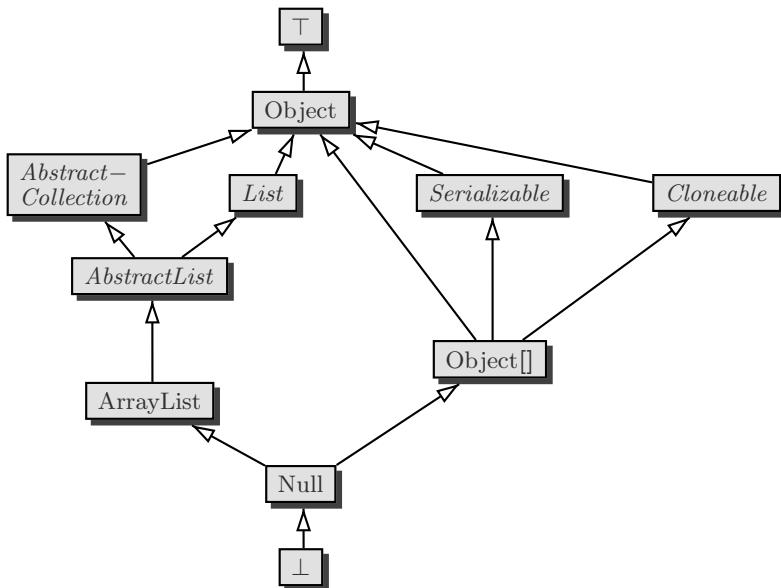
Fig. 3.2. Example for a Java Card DL type hierarchy (built-in types partly omitted)

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 (local) variables, attributes, and arrays outside of modalities, i.e., they occur as terms in JAVA CARD DL. Local 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 (constants).

We only allow signatures that contain certain function and predicate symbols. For example, we require that a JAVA CARD DL signature contains constants $0, 1, \dots$ representing the integer numbers, function symbols for arithmetical operations like addition, subtraction, etc., and the typical ordering predicates on the integers.

Definition 3.4 (JAVA CARD DL signature). *Let T be a type hierarchy, and let FSym_r^0 , FSym_{nr}^0, PSym_r^0 , and PSym_{nr}^0 be the sets of rigid and non-rigid function and predicate symbols from App. 4.*

Then, a JAVA CARD DL signature (for T) 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 (as in the first-order case, Def. 2.8),*
- *a set FSym_r of rigid function symbols and a set FSym_{nr} of non-rigid function symbols such that*

$$\mathrm{FSym}_r \cap \mathrm{FSym}_{nr} = \emptyset$$
$$\mathrm{FSym}_r^0 \subseteq \mathrm{FSym}_r$$
$$\mathrm{FSym}_{nr}^0 \subseteq \mathrm{FSym}_{nr},$$

- *a set PSym_r of rigid predicate symbols and a set PSym_{nr} of non-rigid predicate symbols such that*

$$\mathrm{PSym}_r \cap \mathrm{PSym}_{nr} = \emptyset$$
$$\mathrm{PSym}_r^0 \subseteq \mathrm{PSym}_r$$
$$\mathrm{PSym}_{nr}^0 \subseteq \mathrm{PSym}_{nr}, \text{ and}$$

- *a typing function α (as in the first-order case, Def. 2.8).*

In the remainder of this chapter, with signature *we always mean a JAVA CARD DL signature, unless stated otherwise.*

Example 3.5. Given the type hierarchy from Example 3.3 ($\Rightarrow$ Fig. 3.2), an example for a signature is the following:

$$\mathrm{VSym} = \{a, n, x\}$$

with

$$a:\text{ArrayList},\quad n:\texttt{integer},\quad x:\texttt{integer}$$

$$\text{FSym}_r = \{f, g\} \cup \text{FSym}_r^0$$

with

$$f : \texttt{integer} \rightarrow \texttt{integer}$$
$$g : \texttt{integer}$$

$$\text{FSym}_{nr} = \{al, arg, c, data, i, j, length, para1, sal, v\} \cup \text{FSym}_{nr}^0$$

with

al :	ArrayList
arg :	int
c :	integer
$data$:	ArrayList $\rightarrow$ Object[]
i :	int
j :	short
$length$:	ArrayList $\rightarrow$ int
$para1$:	ArrayList
sal :	ArrayList
v :	int

and

$$\text{PSym}_r = \text{PSym}_r^0$$

Note 3.6. In the KeY system the user never has to explicitly define the whole type hierarchy and signature but the system automatically derives large parts of both from the JAVA CARD program under consideration. Only types and symbols that do not appear in the program must be declared manually. Note, however, that from a logical point of view the type hierarchy and the signature are fixed *a priori*, and formulae (and thus programs in modal operators being part of a formula) must only contain types and symbols declared in the type hierarchy and signature.

The syntactic categories of first-order logic are terms and formulae. Here, we need an additional category called *updates* [Beckert, 2001], which are used to (syntactically) represent state changes.

In contrast to first-order logic, the definition of terms and formulae (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 formulae (see Example 3.9). Nevertheless, in order to improve readability we give separate definitions of updates, terms, and formulae in the following.

3.2.2 Syntax of Java Card DL Terms

Definition 3.7 (Terms of Java Card DL). *Given a Java Card DL signature* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *for a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$*, the system* $\{Terms_A\}_{A \in \mathcal{T}}$ *of sets of* terms of static type A *is inductively defined as the least 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* $\mathrm{FSym}_r \cup \mathrm{FSym}_{nr}$ *and terms* $t_i \in Terms_{A'_i}$ *with* $A'_i \sqsubseteq A_i$ *($1 \le i \le n$)*;
- *(if* ϕ *then* t_1 *else* t_2*)* $\in Terms_A$ *for all* $\phi \in Formulae$ *($\Rightarrow$ Def. 3.14) and all terms* $t_1 \in Terms_{A_1}, t_2 \in Terms_{A_2}$ *with* $A = A_1 \sqcup A_2$;
- *(ifExMin* $x.\phi$ *then* t_1 *else* t_2*)* $\in Terms_A$ *for all variables* $x \in \mathrm{VSym}$, *all formulae* $\phi \in Formulae$ *($\Rightarrow$ Def. 3.14), 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* $u \in Updates$ *($\Rightarrow$ Def. 3.8) 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* $[\,](a, i)$.[1]

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 (if ϕ then t_1 else t_2) and (ifExMin $x.\phi$ then t_1 else t_2) (which could be defined for first-order logic as well). Informally, if ϕ holds, a conditional term (if ϕ then t_1 else t_2) denotes the domain element t_1 evaluates to. Otherwise, if ϕ does not hold, t_2 is evaluated. The meaning of a term (ifExMin $x.\phi$ then t_1 else t_2) is a bit more involved. If there is some d such that ϕ holds, then the whole term evaluates to the value denoted by t_1 under the variable assignment $\beta_x^{d'}$, where d' is the least element satisfying ϕ. Otherwise, if ϕ does not hold for any x, then t_2 is evaluated.

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

3.2.3 Syntax of Java Card DL Updates

Definition 3.8 (Syntactic updates of Java Card DL). *Given a Java Card DL signature* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *for a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$*, the set* Updates *of* syntactic updates *is inductively defined as the least set such that:*

Function update: $(f(t_1, \ldots, t_n) := t) \in Updates$ *for all terms* $f(t_1, \ldots, t_n) \in Terms_A$ *($\Rightarrow$ Def. 3.7) with* $f \in \mathrm{FSym}_{nr}$ *and* $t \in Terms_{A'}$ *s.t.* $A' \sqsubseteq A$;
Sequential update: $(u_1 \,;\, u_2) \in Updates$ *for all* $u_1, u_2 \in Updates$;
Parallel update: $(u_1 \,||\, u_2) \in Updates$ *for all* $u_1, u_2 \in Updates$;

---

[1] Note, that $[\,]$ is a normal function symbol declared in the signature.

Quantified update: ($\mathtt{for}$ x; ϕ; u) $\in$ *Updates for all* $u \in$ *Updates, $x \in$ VSym,
and $\phi \in$ Formulae ($\Rightarrow$ Def. 3.14);*
Update application: ($\{u_1\}$ u_2) $\in$ *Updates for all* $u_1, u_2 \in$ *Updates.*

Syntactic updates can be seen as a language for describing program transitions. Informally speaking, function updates correspond to assignments in an imperative programming language and sequential and parallel updates correspond to sequential and parallel composition, respectively. Quantified updates are a generalisation of parallel updates. A quantified update ($\mathtt{for}$ x; ϕ; u) can be understood as (the possibly infinite) sequence of updates

$$\cdots \,||\, [x/t_n]u \,||\, \cdots \,||\, [x/t_0]u$$

put in parallel. The individual updates $[x/t_n]u, \dots, [x/t_0]u$ are obtained by substituting the free variable x in the update u with all terms $t_n, \dots, t_0$ such that $[x/t_i]\phi$ holds (it is assumed that all terms t_i evaluate to different domain elements). For parallel updates, the order matters. In case of a clash, i.e., if two updates put in parallel modify the same location, the latter one dominates the earlier one (if read from left to right). Coming back to our approximation of quantified updates by parallel updates, this means, that the order of the updates $[x/t_i]u$ put in parallel is crucial. As we see in Def. 3.27, the order depends on a total order $\preceq$, that is imposed on the domain, such that for all $[x/t_i]u$ the following holds: t_i evaluates to a domain element that is less than all the elements t_j ($j > i$) evaluate to (with respect to $\preceq$).

Updates vs. Other State Transition Languages

The idea of describing state changes by a (syntactically quite restrictive) language like JAVA CARD DL updates is not new and appears in slightly different ways in other approaches as well. For example, abstract state machines (ASMs) [Gurevich, 1995] are also based on updates which, however, have a different clash resolution strategy (clashing updates have no effect).

Another concept that is similar to updates are generalised substitutions in the B language [Abrial, 1996].

According to the semantics we define below, JAVA CARD DL terms are (like first-order terms) evaluated in a first-order interpretation that fixes the meaning of function and predicate symbols. However, in JAVA CARD DL models, we have many first-order interpretations (representing program states) rather than only one. Programs occurring in modal operators describe a state transition to the state in which the formula following the modal operator is evaluated. Updates serve basically the same purpose, but they are simpler in many respects.

A simple function update describes a transition from one state to exactly one successor state (i.e., the update process always "terminates normally" in our terminology). Exactly one "memory location" is changed during this transition. None of the above holds in general for a JAVA assignment. Furthermore,

the syntax of updates generated by the calculus that we define ($\Rightarrow$ Sect. 3.6) is restricted even further, 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 formulae.

Example 3.9. Given the type hierarchy and the signature from Examples 3.3 and 3.5, respectively, the following are JAVA CARD DL terms:

n	a variable
c	a non-rigid 0-ary function (constant)
$\{c := 0\}(c)$	a term with a function update
$\{c := 0 \,\|\, c := 1\}(c)$	a term with a parallel update
$\{\mathtt{for}\ x;\ x \doteq 0 \mid x \doteq 1;\ c := x\}(c)$	
	a term with a quantified update
$\{\mathtt{for}\ a;\ a \sqsubseteq \text{ArrayList};\ length(x) := 0\}(length(al))$	
	a term with a quantified update

In contrast, the following are *not* terms:

f	wrong number of arguments
$\{n := 0\}(c)$	update tries to change the value of a variable
$\{g := 0\}(c)$	update tries to change the value of a rigid function symbol
$\{\mathtt{for}\ i;\ i \doteq 0 \mid i \doteq 1;\ c := i\}(c)$	
	an update quantifying over a term instead of a variable
	(i was declared to be a function symbol)

Updates vs. Substitutions

In classical dynamic logic [Harel et al., 2000] and Hoare logic [Hoare, 1969] there are no updates. Modifications of states are expressed using equations and syntactic substitutions. Consider, for example, the following instance of the assignment rule for classical dynamic logic

$$\frac{\Gamma,\ x' \doteq x + 1 \Longrightarrow [x/x']\phi,\ \Delta}{\Gamma \Longrightarrow \langle x = x + 1 \rangle \phi,\ \Delta}$$

where the fresh variable x' denotes the new value of x. The formula ϕ must be evaluated with the new value x' and therefore x is substituted with x'. The equation $x' \doteq x + 1$ establishes the relation between the old and the new value of x.

In principle, updates are not more expressive than substitutions. However, for reasoning about programs in an object-oriented programming language like JAVA CARD updates have some advantages.

The main advantage of updates is that they are part of the syntax of the (object-level) logic, while substitutions are only used on the meta-level to

describe and manipulate formulae. Thus, updates can be collected and need not be applied until the whole program has been symbolically executed (and, thus, has disappeared). The collected updates can be simplified before they are actually applied, which often helps to avoid case distinctions in proofs. Substitutions in contrast are applied immediately and thus there is no chance of simplification (for more details see Sect. 3.6.1).

Another point in favour for updates is that substitutions—as usually defined for first-order logic—replace variables with terms. This, however, does not help to handle JAVA CARD assignments since they modify non-rigid functions rather than variables. Thus, in order to handle assignments with updates, a more general notion of substitution would become necessary.

3.2.4 Syntax of JAVA CARD DL Formulae

Before we define the syntax of JAVA CARD DL formulae, we first define *normalised* JAVA CARD programs, which are allowed to appear in formulae (within modalities). The normal form can be established automatically by a simple program transformation and/or extension of the type hierarchy and the signature and does not constitute a real restriction.

Normalised JAVA CARD Programs

The definition of normalised JAVA CARD programs is necessary for two reasons. First, we do not want to handle certain features of JAVA CARD (like, e.g., inner classes) in the calculus ($\Rightarrow$ Sect. 3.4.6), because including them would require many rules to be added to our calculus. The approach we pursue is to remove such features by a simple program transformation. The second reason is that a JAVA CARD program must only contain types and symbols declared in the type hierarchy and signature. Note, that this does not restrict the set of possible JAVA CARD programs. It is always possible to adjust the type hierarchy and signature to a given program.

Since JAVA CARD code can appear in formulae, we actually have to give a formal definition of the syntax of JAVA CARD. This however goes beyond the scope of this book and we refer the reader to the JAVA CARD language specification [Chen, 2000, Sun, 2003d,e].

Definition 3.10 (Normalised JAVA CARD programs). *Given a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *for a signature* (VSym, $\mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r,$ $\mathrm{PSym}_{nr}, \alpha$), *a normalised* JAVA CARD *program* P *is a set of (abstract) class and interface definitions satisfying the following constraints:*

1. P is compile-correct and compile-time constants of an integer type do not cause overflow.[2]
2. P does not contain inner classes.
3. Identifiers in declarations of local variables, attributes, and parameters of methods (and constructors) are unique.
4. $A \in \mathcal{T}_a$ for all interface and abstract class types A declared in or imported into P.
5. $A \in \mathcal{T}_d$ for all non-abstract class types A declared in or imported into P.
6. $C \sqsubseteq D$ 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.
7. For all array types $A\underbrace{[\,]\cdots[\,]}_{n\ times}$ (or $A[\,]^n$ for short where $A[\,]^0 = A$) occurring in P and $1 \le i \le n$:
   - $A \in \mathcal{T}$,
   - $B[\,]^m \sqsubseteq A[\,]^n$ iff $B[\,]^{m-1} \sqsubseteq A[\,]^{n-1}$ for all $B[\,]^m \in \mathcal{T}$ $(m \ge 1)$,
   - $A[\,]^i \in \mathcal{T}_d$,
   - $A[\,]^i \sqsubseteq \mathrm{Object}$,
   - $A[\,]^i \sqsubseteq \mathrm{Serializable}$,
   - $A[\,]^i \sqsubseteq \mathrm{Cloneable}$, and
   - $B \not\sqsubseteq A[\,]^i \in \mathcal{T}_d$ for all non-array types $B \in \mathcal{T} \setminus \{\bot, \mathrm{Null}\}$,
   - $\mathrm{Null} \sqsubseteq A[\,]^i$.
8. 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:$(A'[\,]^n) \in \mathrm{FSym}_{nr}$.
9. 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 $id : (C \to A'[\,]^n) \in \mathrm{FSym}_{nr}$.

Let Π denote the set of all normalised JAVA CARD programs.

Not surprisingly, we require that the programs P we consider are compile-correct (Constraint (1)). Constraint (2) requires that P does not contain inner classes like, e.g., anonymous classes. In principle, this restriction could be dropped. This however would result in a bunch of extra rules for the calculus to be defined in Sect. 3.4.6.

In contrast to the programming language JAVA CARD, in JAVA CARD DL we do not have overloading of function or predicate symbols. Therefore we require that the identifiers used in declarations in P are unique (Constraint (3)). For instance, it is not allowed to have two local variables with the same name occurring in P. Field hiding is disallowed by the same token.

---

[2] The second condition can be checked statically. For example, a compile time constant like `final byte b=(byte)500;` is not allowed since casting the literal `500` of type `int` to type `byte` causes overflow.

Note, that the Constraints (2) and (3) are harmless restrictions in the sense that any JAVA CARD program can easily be transformed into an equivalent program satisfying the constraints.

Constraints (4) and (5) make sure that all non-array reference types declared in and imported into P are contained in the type hierarchy. This in particular applies to all classes that are automatically imported into any program like the classes in package `java.lang` (in particular `Object`).

Constraint (6) guarantees that the inheritance hierarchy of the JAVA CARD program P is correctly reflected by the subtype relation in the type hierarchy.

Array reference types are addressed in Constraint (7). Array types are not declared explicitly in JAVA CARD like class or interface types but nevertheless they still must be part of the type hierarchy and the subtype relation must match the inheritance hierarchy in JAVA CARD, i.e., array types are subtypes of Serializable, Cloneable, and Object.

The first condition requires the element type A of an array type $A[]^n$ to be part of the type hierarchy ($A \in \mathcal{T}$). The subtype relation between two array types $B[]^m$ and $A[]^n$ is recursively defined on the component types $B[]^{m-1}$ and $A[]^{n-1}$. Therefore, we additionally postulate that all array types up to dimension n with element type A are contained in the set of dynamic types ($A[]^i \in \mathcal{T}_d$) as well. Then the recursive definition is well-founded since eventually we arrive at non-array types and we require that $A[]^i \sqsubseteq$ Object. Finally, we stipulate that non-array types $B \in \mathcal{T} \setminus \{\bot, \text{Null}\}$ must not be a subtype of any array type $A[]^i$.

Local variables and static fields in JAVA CARD occur as non-rigid 0-ary functions in the logic (i.e., as constants). Therefore, we require for any such element a corresponding function to be present in the signature (Constraint (8)).

Finally, in Constraint (9) we consider non-static fields which are represented by non-rigid unary functions that map instances of the class declaring the field to elements of the field type.

In order to normalise a JAVA CARD program, Constraints (2) and (3) can always be satisfied by performing a program transformation. For example, inner classes can be transformed into top-level classes; identifiers (e.g., attributes or local variables) can be renamed. On the other hand, meeting the Constraints (4)–(9) may require an extension of the underlying type hierarchy and signature, since only declared types and symbols may be used in a normalised JAVA CARD program. However, such an extension is harmless and is done automatically by the KeY system, i.e., the user does not have to explicitly declare all the types and symbols occurring in the JAVA CARD program to be considered.

Example 3.11. Given the type hierarchy and signature from Examples 3.3 and 3.5, respectively, the following set of classes and interfaces constitute a normalised JAVA CARD program.

```
──── JAVA ──────────────────────────────────────────────
   abstract class AbstractCollection {
 2 }

 4 interface List {
   }
 6
   abstract class AbstractList
 8   extends AbstractCollection implements List {
   }
10
   class ArrayList extends AbstractList {
12
     static ArrayList sal;
14   static int v;

16   Object[] data;
     int length;
18
     public static void demo1() {
20     int i=0;
     }
22
     public static void demo2(ArrayList para1) {
24     // int i=1; violates Constraint (3)
       short j;
26     if (para1==null)
         j=0;
28     else
         j=1;
30     para1.demo3();
     }
32
     void demo3() {
34     this.length=this.length+1;
     }
36
     int inc(int arg) {
38     return arg+1;
     }
40
   }
42
   // class Violate { violates Constraint (5)
44 //   int k; violates Constraint (8)
   // }
────────────────────────────────────────────────── JAVA ──
```

The above program satisfies all the constraints from Def. 3.10. All interface types, (abstract) class types, and array types are contained in the corresponding JAVA CARD DL type hierarchy, and for all identifiers in the program there is a type correct function in the signature.

The statement in line 24 (commented out) would violate Constraint (3) since method `demo1` already declares a local variable with identifier `i`.

Similarly, declaring a class type `Violate` that is not contained in the type hierarchy (as in line 43) violates Constraint (5). Also not allowed is declaring a local variable if the signature does not contain the corresponding function symbol (line 44).

Within modal operators we exclusively allow for sequences of statements. A so-called *program statement* is either a normal JAVA statement, a *method-body statement*, or a *method-frame statement*. Note that logical variables, in contrast to non-rigid function symbols reflecting local program variables, attributes, and arrays, must not occur in programs.

Intuitively, a method-body statement is a shorthand notation for the precisely identified implementation of method $m(\ldots)$ in class T. That is, in contrast to a normal method call in JAVA CARD where the implementation to be taken is determined by dynamic binding, a method-body statement is a call to a method declared in a type that is precisely identified by the method-body statement.

A method-frame statement is required when handling a method call by syntactically replacing it with the method's implementation ($\Rightarrow$ Sect. 3.6.5). To handle the return statement in the right way, it is necessary

1. to record the object field or variable x that the result is to be assigned to, and
2. to mark the boundaries of the implementation body when it is substituted for the method call.

For that purpose, we allow a method-frame statement to occur as a JAVA CARD DL program statement.

Definition 3.12 (JAVA CARD DL program statement). *Let $P \in \Pi$ be a normalised JAVA CARD program. Then a JAVA CARD DL program statement is*

- *a JAVA statement as defined in the JAVA language specification [Gosling et al., 2000, § 14.5] (except `synchronized`),*
- *a method-body statement*

$$retvar\texttt{=}target\texttt{.}m\texttt{(}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{)}\texttt{@}T\texttt{;}$$

where
- *$target\texttt{.}m\texttt{(}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{)}$ is a method invocation expression,*
- *the type T points to a class declared in P (from which the implementation is taken),*

– the result of the method is assigned to *retvar* after return (if the method is not void), or
• a method-frame statement

```
method-frame(result->retvar, source=T, this=target) : { body }
```

where
– *the return value of the method is assigned to retvar when body has been executed (if the method is not void),*
– *the type T points to the class in P providing the particular method implementation,*
– *target is the object the method was invoked on,*
– *body is the body of the invoked method.*

Thus, all JAVA statements that are defined in the official language specification can be used (except for `synchronized` blocks), and there are two additional ones: a `method-body` statement and a `method-frame` statement.

Another extension is that we do not require *definite assignment*. In JAVA, the value of a local variable or `final` field must have a definitely assigned value when any access of its value occurs [Gosling et al., 2000, § 16]. In JAVA CARD DL we allow sequences of statements that violate this condition (the variable then has a well-defined but unknown value).

Note, that the additional constructs and extensions are a "harmless" extension as they are only used for proof purposes and never occur in the verified JAVA CARD programs.

Definition 3.13 (Legal sequence of JAVA CARD DL program statements). *Let $P \in \Pi$ be normalised JAVA CARD program.*
A sequence $st_1 \cdots st_n$ ($n \geq 0$) of JAVA CARD DL program statements is legal w.r.t. to P if P enriched with the class declaration

```
public class DefaultClass {
  public static void defaultMethod() {
    st₁
    ⋮
    stₙ
  }
}
```

where DefaultClass and defaultMethod are fresh identifiers—is a normalised JAVA CARD program, except that $st_1 \cdots st_n$ do not have to satisfy the definite assignment condition [Gosling et al., 2000, § 16].

Now we can define the set of JAVA CARD DL formulae:

Definition 3.14 (Formulae of JAVA CARD DL). *Let a signature* (VSym, $\text{FSym}_r, \text{FSym}_{nr}, \text{PSym}_r, \text{PSym}_{nr}, \alpha$) *for a type hierarchy* T *a normalised* JAVA CARD *program* $P \in \Pi$ *be given.*

Then, the set Formulae of JAVA CARD DL *formulae is inductively defined as the least set such that:*

- $r(t_1, \ldots, t_n) \in$ *Formulae for all predicate symbols* $r : A_1, \ldots, A_n \in \text{PSym}_r \cup \text{PSym}_{nr}$ *and terms* $t_i \in \text{Terms}_{A'_i}$ ($\Rightarrow$ *Def. 3.7*) *with* $A'_i \sqsubseteq A_i$ $(1 \leq i \leq n)$,
- true, false $\in$ *Formulae,*
- $!\phi, (\phi \mid \psi), (\phi \ \& \ \psi), (\phi \rightarrow \psi), (\phi \leftrightarrow \psi) \in$ *Formulae for all* $\phi, \psi \in$ *Formulae,*
- $\forall x.\phi, \exists x.\phi \in$ *Formulae for all* $\phi \in$ *Formulae and all variables* $x \in$ VSym,
- $\{u\}\,\phi \in$ *Formulae for all* $\phi \in$ *Formulae and* $u \in$ *Updates* ($\Rightarrow$ *Def. 3.8*),
- $\langle p \rangle \phi, [p]\phi \in$ *Formulae for all* $\phi \in$ *Formulae and any legal sequence* p *of* JAVA CARD DL *program statements.*

In the following we often abbreviate formulae of the form $(\phi \rightarrow \psi) \ \& \ (!\phi \rightarrow \xi)$ *by* if ϕ then ψ else ξ.

Example 3.15. Given the type hierarchy and the signature from Examples 3.3 and 3.5, respectively, and the normalised JAVA CARD DL program from Example 3.11, the following are JAVA CARD DL formulae:

$\{c := 0\}(c \doteq 0)$	a formula with an update
$(\{c := 0\}c) \doteq c$	a formula containing a term with an update
`sal` $!\doteq$ `null` $\rightarrow \langle$`ArrayList.demo2(sal);`$\rangle$`j` $\doteq 1$	
	a formula with a modal operator
$\{$`sal` $:=$ `null`$\}\langle$`ArrayList.demo2(sal);`$\rangle$`j` $\doteq 0$	
	a formula with a modal operator and an update
$\{$`v` $:= g\}\langle$`ArrayList al=new ArrayList(); v=al.inc(v);`$\rangle(v \doteq g + 1)$	
	a formula with a modal operator and an update
$\{$`v` $:= g\}\langle$ `ArrayList al=new ArrayList();` `v=al.inc(v)@ArrayList`$\rangle(v \doteq g + 1)$	
	a method-body statement within a modal operator
$\langle$`int i=0; v=i;`$\rangle(v \doteq 0)$	local variable declaration and assignment within a modal operator

Note 3.16. 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. For example, the following is not a syntactically correct JAVA CARD DL formula:

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

To express the desired property, there are two possibilities. The first one is using an update to bind the program variable to the quantified logical variable:

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

The second possibility is to use an equation:

$$\forall n.(n \doteq \texttt{v} \rightarrow \langle \, \texttt{ArrayList al=new ArrayList();}$$
$$\texttt{v=al.inc(v);}\rangle(\texttt{v} \doteq n+1))$$

Both possibilities are equivalent with respect to validity: the first one is valid iff the second one is valid.[3]

Before we define the semantics of JAVA CARD DL in the next section, we extend the definition of free variables from Chap. 2 to the additional syntactical constructs of JAVA CARD DL.

Definition 3.17. *We define the set $fv(u)$ of free variables of an update u by:*

- $fv(f(t_1,\ldots,t_n) := t) = fv(t) \cup \bigcup_{i=1}^{n} fv(t_i),$
- $fv(u_1 \, ; u_2) = fv(u_1) \cup fv(u_2),$
- $fv(u_1 \, || \, u_2) = fv(u_1) \cup fv(u_2),$
- $fv(\texttt{for } x; \, \phi; \, u) = (fv(\phi) \cup fv(u)) \setminus \{x\}.$

For terms and formulae we extend Def. 2.18 as follows:

- $fv(\text{if } \phi \text{ then } t_1 \text{ else } t_2) = fv(\phi) \cup fv(t_1) \cup fv(f_2)$
- $fv(\text{ifExMin } x.\phi \text{ then } t_1 \text{ else } t_2) = ((fv(\phi) \cup fv(t_1)) \setminus \{x\}) \cup fv(f_2)$
- $fv(\{u\}t) = fv(u) \cup fv(t)$ *for a term* t,
- $fv(\{u\}\phi) = fv(u) \cup fv(\phi)$ *for a formula* ϕ,
- $fv(\langle p \rangle \phi) = fv(\phi)$ *for a formula* ϕ,
- $fv([p]\phi) = fv(\phi)$ *for a formula* ϕ.

3.3 Semantics

We have seen that 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 the different program states. Traditionally, in modal logics the different models are called *worlds*. But here we call them *states*, which better fits the intuition.

---

[3] Please note that both formulae ϕ_1, ϕ_2 are not logically equivalent in the sense that $\phi_1 \leftrightarrow \phi_2$ is logically valid.

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

- a partial first-order model $\mathcal{M}$ fixing the meaning of *rigid* function and predicate symbols,
- an (infinite) set $\mathcal{S}$ of states where a state is any first-order model refining $\mathcal{M}$, thus assigning meaning to the non-rigid function and predicate symbols (which are not interpreted by $\mathcal{M}$), and
- a program relation ρ fixing the meaning of programs occurring in modalities: $(S_1, p, S_2) \in \rho$ iff the sequence p of statements when started in state S_1 terminates in state S_2, assuming it is executed in some static context, i.e., in some static method declared in some public class.

3.3.1 Kripke Structures

Definition 3.18 (Java Card DL Kripke structure). *Let a signature* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *for a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *be given and let* $P \in \Pi$ *be a normalised Java Card program.*

A Java Card DL Kripke structure $\mathcal{K}$ *for that signature, type hierarchy, and program is a tuple* $(\mathcal{M}, \mathcal{S}, \rho)$ *consisting of a partial first-order model* $\mathcal{M} = (\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$, *a set* $\mathcal{S}$ *of states, and a program relation* ρ *such that:*

- $\mathcal{T}_0 = \mathcal{T}$;
- *the partial domain* $\mathcal{D}_0$ *is a set satisfying*
  - $\mathbb{Z} = \mathcal{D}_0^{\mathtt{integerDomain}}$,
  - $\{tt, ff\} = \mathcal{D}_0^{\mathtt{boolean}}$,
  - $\{null\} = \mathcal{D}_0^{\mathrm{Null}}$,
  - *for all dynamic types* $A \in \mathcal{T}_d \setminus \{\mathrm{Null}\}$ *with* $A \sqsubseteq \mathrm{Object}$ *there is a countably infinite set* $\mathcal{D}_0' \subseteq \mathcal{D}_0$ *such that* $\delta_0(d) = A$ *for all* $d \in \mathcal{D}_0'$,
  - *for all* $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}_r \cup \mathrm{FSym}_{nr}$

$$D_0(f) = \begin{cases} \emptyset & \text{if } f \in \mathrm{FSym}_{nr} \\ \mathcal{D}_0^{A_1} \times \cdots \times \mathcal{D}_0^{A_n} & \text{if } f \in \mathrm{FSym}_r \end{cases}$$

  - *for all* $p : A_1, \ldots, A_n \in \mathrm{PSym}_r \cup \mathrm{PSym}_{nr}$

$$D_0(p) = \begin{cases} \emptyset & \text{if } p \in \mathrm{PSym}_{nr} \\ \mathcal{D}_0^{A_1} \times \cdots \times \mathcal{D}_0^{A_n} & \text{if } p \in \mathrm{FSym}_r \end{cases}$$

  - $\mathcal{I}_0(f)$ *for* $f \in \mathrm{FSym}_r^0$ *(see App. A.2.1),*
  - $\mathcal{I}_0(p)$ *for* $p \in \mathrm{PSym}_r^0$ *(see App. A.2.2);*
- *the set* $\mathcal{S}$ *of Java Card DL states consists of all first-order models* $(\mathcal{D}, \delta, \mathcal{I})$ *refining* $\mathcal{M}$ *with*
  - $\mathcal{D} = \mathcal{D}_0$,
  - $\delta = \delta_0$;

- 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:

$$\rho(S_1, p, S_2)$$
$$iff$$

p started in S_1 in a static context terminates normally in S_2 according to the JAVA language specification [Gosling et al., 2000].

The partial model $\mathcal{M}$ is called Kripke seed *since it determines the set of states of a JAVA CARD DL Kripke structure.*

Note 3.19. In the above definition we require that the partial domain $\mathcal{D}_0$ (which is equal to the domain $\mathcal{D}$) is a set satisfying the mentioned properties. This guarantees in particular that the domain contains exactly the two elements tt and ff with dynamic type boolean and that *null* is 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 subset $\mathcal{D}_0' \subseteq \mathcal{D}_0$ with $\delta_0(d) = A$. These domain elements represent the JAVA CARD objects of dynamic type A. Objects can be created dynamically during the execution of a JAVA CARD program and therefore we do not know the exact number of objects in advance. Since for a smooth handling of quantifiers in a calculus it is advantageous to have a constant domain for all JAVA CARD DL states (see below), we cannot extend the domain on demand if a new object is created. Therefore, we simply require an infinite number of domain elements with an appropriate dynamic type making sure that there is always an unused domain element available to represent a newly created object.

Constant-Domain Assumption

Def. 3.18 requires that both the domain and the dynamic type function are the same for all states in a Kripke structure. This so-called *constant-domain assumption* is a harmless but reasonable and useful restriction as the following example shows.

If we assume a constant domain, then the formula

$$\forall x. p(x) \rightarrow [\pi] \forall x. p(x) \ ,$$

where p is a rigid predicate symbol, is valid in all Kripke structures because p cannot be affected by the program π. Without the constant domain assumption there could be states where this formula does not hold since the domain of the state reached by π may have more elements than the state where $\forall x. p(x)$ holds and in particular might include elements for which p does not hold.

A problem similar to the one above already appears in classical modal logics. In this setting constant-domain Kripke structures are characterised by the so-called *Barcan formula* $\forall x. \Box p(x) \rightarrow \Box \forall x. p(x)$ (see, e.g., the book by Fitting and Mendelsohn [1999]).

The Kripke seed $\mathcal{M}$ of a Kripke structure (Def. 3.18) fixes the interpretation of the rigid function and predicate symbols, i.e., of those symbols that have the same meaning in all states. Moreover, it is "total" for these symbols in the sense that it assigns meaning to rigid symbols for all argument tuples, i.e., $D_0(s) = \mathcal{D}_0^{A_1} \times \cdots \times \mathcal{D}_0^{A_n}$ for any rigid function or predicate symbol s. That means, for example, that division by zero is defined, i.e., $\mathcal{I}_0(x/y)$ (we use infix notation for better readability) yields some (fixed but unknown) element $d \in \mathbb{Z} \subseteq \mathcal{D}_0$ for $y = 0$.

Handling Undefinedness

The way we deal with undefinedness is based on *underspecification* as proposed by Gries and Schneider [1995], Constable and O'Donnell [1978]. Hähnle [2005] argues that this approach is superior to other approaches (at least in the context of specification and verification of programs).

The basic idea is that any function f that is undefined for certain argument tuples (like, e.g., / which is undefined for $\{(x, 0) \mid x \in \mathbb{Z}\}$) is made total by assigning a fixed but unknown result value for those arguments where it is undefined. This is achieved using a dedicated (semantic) choice function $choice_f$ which has the same arity as f. For example, for / the choice function $choice_/$ could be defined as $choice_/(x, 0) = x$.

In the presence of choice functions, the definition of validity needs to be revised such that a formula ϕ is said to be valid in a model $\mathcal{M}$ iff it is valid in $\mathcal{M}$ for all possible definitions of the choice functions. That is, it is crucial that all possibilities for a choice function are considered rather than relying on just one particular possibility.

In Example 2.41 we explained how this can be achieved in classical first-order logic making use of partial models, leaving open the interpretation of functions for critical argument tuples. A formula is then valid in a (partial) model $\mathcal{M}$ iff it is valid in all (total) models refining $\mathcal{M}$—making sure that all possibilities for choice functions are considered.

In order to carry over this approach to the Kripke semantics of JAVA CARD DL there are two options:

1. Leaving open the interpretation of functions for critical argument tuples in the Kripke seed. Then all possibilities for the choice functions are considered in the states of the Kripke structure, which are defined as the set of all models refining the Kripke seed.
2. Fixing a particular choice function in the Kripke seed. Then in order to consider all choice functions all possible Kripke seeds need be to taken into account.

In Def. 3.18 we chose the second of these two options, and there are good reasons for this decision. Since a formula is defined to be valid iff it is valid in all Kripke structures ($\Rightarrow$ Def. 3.38), we need to consider all Kripke

structures (and thus all Kripke seeds) anyway. The second and more important argument is that, if we chose the first option, each single Kripke structure contains states in which the same "undefined" term would evaluate to different values. Such a term would then not be rigid anymore (Lemma. 3.33)—even if the function symbol is declared to be rigid and the arguments are rigid. That would heavily complicate the definitions of the semantics of JAVA CARD DL formulae containing modal operators and of update simplification in Sect. 3.9. For example, without modifying the semantics of JAVA CARD DL formulae, the formula

$$g \doteq 5/0 \mathbin{-\!\!>} \langle \texttt{int i=0;} \rangle g \doteq 5/0 \ ,$$

where g is a rigid constant, would no longer be valid since the program might terminate in a state where $5/0$ has a meaning different from that in the initial state (whereas g has the same meaning in all states since it is rigid). Hence it is beneficial to fix the semantics of all rigid functions for all argument tuples already in the Kripke seed.

Please note, that—besides underspecification—there are several other ways to deal with undefinedness in formal languages. One possibility is to introduce an explicit value *undefined*. That approach is pursued, e.g., in OCL [OCL 2.0]. It has the disadvantage that the user needs to know non-standard semantics in order to evaluate expressions. Further approaches, such as allowing for partially defined functions, are discussed in the article by Hähnle [2005].

The Kripke seed does not provide an interpretation of the non-rigid symbols, which is done by the models refining the seed, i.e., the states of the Kripke structure.

The semantics of normalised JAVA CARD programs is given by the relation ρ, where ρ holds for (S_1, p, S_2) iff the sequence p of statements, when started in S_1, terminates *normally* in S_2. *Normal* termination means that the program does not terminate abruptly (e.g., because of an uncaught exception). Otherwise ρ does not hold, i.e., if the program terminates *abruptly* or does not terminate at all.

Non-reachable States

According to Def. 3.18, the set S of JAVA CARD DL states contains *all* possible states (i.e., all structures refining the Kripke seed). That implies that a JAVA CARD DL Kripke structure also contains states that are not reachable by any JAVA CARD program (e.g., states in which a class is both marked as initialised and erroneous). The main reason for not excluding such states is that even if they cannot be reached by any JAVA CARD program, they can still be reached (or better: described) by updates. For example, the update

> T.`<classInitialised>` := TRUE $||\, T$.`<erroneous>` := TRUE
>
> describes a state in which a class $T \sqsubseteq$ Object is both initialised and erroneous. Such states do not exist in Java Card.
>
>   There is no possibility to syntactically restrict the set of updates such that only states reachable by Java Card programs can be described. Of course, one could modify the semantics of updates such that updates leading to non-reachable states do not terminate or yield an unspecified state. That however would make the semantics and simplification of updates much more complicated.

Note 3.20. The definition that abrupt termination and non-termination are treated the same is not a necessity but a result of the answer to the question of when we consider a program to be correct. On the level of Java Card DL we say that a program is totally correct if it terminates normally and if it satisfies the postcondition (assuming it satisfies the precondition). Thus, if something unexpected happens and the program terminates abruptly then it is not considered to be totally correct—even if the postcondition holds in the state in which the execution of the program abruptly stops.

Other languages like, e.g., the Java Modeling Language (JML) have a more fine-grained interpretation of correctness with respect to (abrupt) termination. JML distinguishes between normal termination and abrupt termination by an uncaught exception, and it allows to specify different postcondition for each of the two cases. Since in the KeY tool we translate JML expressions into Java Card DL formulae we somehow have to mimic the distinction between non-termination and abrupt termination in Java Card DL.

This is done by performing a program transformation such that the resulting program catches all exceptions at top-level and thus always terminates normally. The fact, that the original program would have terminated abruptly is indicated by the value of a new Boolean variable. For example, in the following formula, the program within the modal operator terminates normally, independently of the value of `j`.

—— KeY ——

```
\<{
Throwable thrown = null;
  try {
    i = i / j;
 } catch (Exception e) {
   thrown = e;
 }
}\> (thrown != null)
```

In the postcondition the formula `thrown !≐ null` holds if and only if the original program (without the `try-catch` block) terminates abruptly.

For a more detailed account of this issue the reader is referred to Sects. 3.7.1 and 8.2.3.

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

3.3.2 Semantics of JAVA CARD DL Updates

Similar to the first-order case we inductively define a valuation function $\text{val}_{\mathcal{M}}$ assigning meaning to updates, terms, and formulae. Since non-rigid function and predicate symbols can have different meanings in different states, the valuation function is parameterised with a JAVA CARD DL state, i.e., for each state $\mathcal{S}$, 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 semantics of programs contained in modal operators and indeed updates are used to describe the effect of programs.

In parallel updates $u_1 \| u_2$ (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 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 $\preceq$ 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 $min_{\preceq}(\mathcal{D}_{sub})$). The parallel application of unbounded sets of updates can then be well-ordered as well, and clashes can be resolved by giving precedence to the update assigning the smallest value. For this reasons, we first equip JAVA CARD DL Kripke structures with a well-ordering on the domain.

Definition 3.21 (JAVA CARD DL Kripke structure with ordered domain). *A* JAVA CARD *DL Kripke structure with ordered domain $\mathcal{K}_{\preceq}$ is a* JAVA CARD *DL Kripke structure $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$ with a well-ordering on $\mathcal{D}$, i.e., a binary relation $\preceq$ with the following properties:*

- *$x \preceq x$ for all $x \in \mathcal{M}$ (reflexivity),*
- *$x \preceq y$ and $y \preceq x$ implies $x = y$ (antisymmetry),*
- *$x \preceq y$ and $y \preceq z$ implies $x \preceq z$ (transitivity), and*
- *any non-empty subset $\mathcal{D}_{sub} \subseteq \mathcal{D}$ has a least element $min_{\preceq}(\mathcal{D}_{sub})$, i.e., $min_{\preceq}(\mathcal{D}_{sub}) \preceq y$ for all $y \in \mathcal{D}_{sub}$ (well-orderedness).*

As every set can be well-ordered (based on Zermelo-Fraenkel set theory [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 depending on the problem. In the implementation of the KeY system, we have chosen the following order as it allows to capture the effects of a particular class of loops in quantified updates in a rather nice way Gedell and Hähnle [2006]. Note however, that the order can be modified without having to adapt other definitions of the logic except for the predicate *quanUpdateLeq* that allows to access the order on the object level (it is required for update simplification ($\Rightarrow$ Sect. 3.9)).

Definition 3.22 (KeY Java Card DL Kripke structure). *A* KeY Java Card DL Kripke structure *is a Java Card DL Kripke structure with ordered domain, where the order $\preceq$ is defined for any $x, y \in \mathcal{D}^\top$ as follows:*

- *If $\delta_0(x) \neq \delta_0(y)$ then*
$$\begin{cases} x \preceq y & \text{if } \delta_0(x) \sqsubseteq \delta_0(y) \\ y \preceq x & \text{if } \delta_0(y) \sqsubseteq \delta_0(x) \\ x \preceq y & \text{if } \delta_0(x) \leq_{lex} \delta_0(y) \text{ and neither} \\ & \quad \delta_0(x) \sqsubseteq \delta_0(y) \text{ nor } \delta_0(y) \sqsubseteq \delta_0(x) \end{cases}$$
  where $\leq_{lex}$ is the usual lexicographic order on the names of types.
- *If $\delta_0(x) = \delta_0(y)$ then*
  - *if $\delta_0(x) =$ boolean then $x \preceq y$ iff $x = \textit{ff}$*
  - *if $\delta_0(x) =$ integerDomain then $x \preceq y$ iff*
    $x \geq 0$ and $y < 0$ or
    $x \geq 0$ and $y \geq 0$ and $x \leq y$, or
    $x < 0$ and $y < 0$ and $y \leq x$
  - *if Null $\neq A = \delta_0(x) \sqsubseteq$ Object then $x \preceq y$ iff $index_A(x) \preceq index_A(y)$ where $index_T : \mathcal{D}^T \to$ integer is some arbitrary but fixed bijective mapping for all dynamic types $T \in \mathcal{T}_d \setminus \{\text{Null}\}$*
  - *if $\delta_0(x) =$ Null then $x = y$.*

The semantics of an update is defined—relative to a given Java Card DL state—as a partial first-order model $(\mathcal{T}_0, \mathcal{D}_0, \delta, D, \mathcal{I}_0)$ that is defined exactly on those tuples of domain elements that are affected by the update, i.e., the partial model describes only the modifications induced by the update. Thus, the semantics of updates is given by a special class of partial models that differ only in D and $\mathcal{I}_0$ from a Java Card DL state $(\mathcal{T}_0, \mathcal{D}_0, \delta, D', \mathcal{I}_0')$ (here seen as a partial model), i.e., an update neither modifies the set $\mathcal{T}_0$ of fixed types, nor the partial domain $\mathcal{D}_0$, nor the dynamic type function δ. In order to improve readability, we therefore introduce so-called *semantic updates* to capture the semantics of (syntactic) updates.

Definition 3.23. *Let $(\text{VSym}, \text{FSym}_r, \text{FSym}_{nr}, \text{PSym}_r, \text{PSym}_{nr}, \alpha)$ be a signature for a type hierarchy. 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 3.24. *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. 3.27, 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, formulae, or updates following an update u are evaluated, CU is applied to S yielding a state S'.

Definition 3.25 (Application of semantic updates). *Let* $(\mathrm{VSym}, \mathrm{FSym}_r,$ $\mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *be a signature for a given type hierarchy and let* $\mathcal{M} = (\mathcal{D}_0, \delta, \mathcal{I}_0)$ *be a first-order model for that signature.*

For any set $CU \in \mathcal{CU}$ of consistent semantics updates, the modification $CU(\mathcal{M})$ *is defined as the model $(\mathcal{D}'_0, \delta', \mathcal{I}'_0)$ with*

$$\mathcal{D}'_0 = \mathcal{D}_0$$
$$\delta' = \delta$$
$$\mathcal{I}'_0(f)(d_1, \ldots, d_n) = \begin{cases} d & \text{if } (f, (d_1, \ldots, d_n), d) \in CU \\ \mathcal{I}_0(f)(d_1, \ldots, d_n) & \text{otherwise} \end{cases}$$

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

Intuitively, a set CU of consistent semantic updates modifies the interpretation of $\mathcal{M}$ for the locations that are contained in CU.

Note 3.26. The consistency condition in Def. 3.24 guarantees that the interpretation function $\mathcal{I}'$ in Def. 3.25 is well-defined.

Definition 3.27 (Semantics of JAVA CARD DL updates). *Given a signature for a type hierarchy, let $\mathcal{K}_{\preceq} = (\mathcal{M}, \mathcal{S}, \rho)$ be a JAVA CARD DL Kripke structure with ordered domain, let β be a variable assignment, and let $P \in \Pi$ be a normalised JAVA CARD program.*

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

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

$$d_i = \mathrm{val}_{S,\beta}(t_i) \qquad (1 \leq i \leq n)$$
$$d = \mathrm{val}_{S,\beta}(s) \ ,$$

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

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

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

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

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

$\quad U = \{(f, (d_1, \ldots, d_n), d) \mid \text{ there is } a \in \mathcal{D}^A \text{ such that}$
$\qquad\qquad\qquad\qquad\qquad ((f, (d_1, \ldots, d_n), d), a) \in dom \text{ and}$
$\qquad\qquad\qquad\qquad\qquad b \not\preceq a \text{ for all } ((f, (d_1, \ldots, d_n), d'), b) \in dom\}$

$\quad$ *with* $dom = \bigcup_{a \in \{d \in \mathcal{D}^A \mid S, \beta_x^d \models \phi\}} (\mathrm{val}_{S,\beta_x^a}(u) \times \{a\})$, *and* A *is the type of* x,
- $\mathrm{val}_{S,\beta}(\{u_1\}\, u_2) = \mathrm{val}_{S',\beta}(u_2)$ *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.

In both sequential and parallel 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 3.28. Consider the updates

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

and

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

where c is a non-rigid constant. We stepwise evaluate these updates in a Java Card DL state $S_1 = (\mathcal{D}, \delta, \mathcal{I}_1)$ with $\mathcal{I}_1(c) = 0$.

$$\mathrm{val}_{S_1}(c := c + 1 \,;\, c := c + 2) = (U_1 \cup U_2) \setminus C$$

where

$U_1 = \mathrm{val}_{S_1,\beta}(c := c + 1) = \{(c, (), 1)\}$
$U_2 = \mathrm{val}_{S_2,\beta}(c := c + 2) = \{(c, (), 3)\} \qquad$ with $S_2 = \mathrm{val}_{S_1,\beta}(c := c + 1)(S_1)$
$C = \{(f, (d_1, \ldots, d_n), d) \mid (f, (d_1, \ldots, d_n), d) \in U_1$ and
$\qquad\qquad\qquad\qquad\qquad (f, (d_1, \ldots, d_n), d') \in U_2 \text{ for some } d' \neq d\}$
$\quad = \{(c, (), 1)\}$

That is, we first evaluate the sub-update $c := c + 1$ in state S_1 yielding U_1. In order to evaluate the second sub-update, we first have to apply U_1 to state S_1, which results in the state S_2 that coincides with S_1 except for the interpretation of c, which is $\mathcal{I}_2(c) = 1$. The evaluation of $c := c + 2$ in S_2 yields U_2. From the union $U_1 \cup U_2$ we have to remove the set C of conflicting semantic updates and finally obtain the result

$$\text{val}_{S_1}(c := c + 1 \,; c := c + 2) = \{(c, (), 3)\} \;,$$

i.e., a semantic update that fixes the interpretation of the 0-ary function symbol c to be the value 3.

On the other hand, the semantics of the parallel update in state S_1 is defined as

$$\text{val}_{S_1}(c := c + 1 \,||\, c := c + 2) = (U_1 \cup U_2) \setminus C$$

where

$$
\begin{aligned}
U_1 &= \text{val}_{S_1}(c := c + 1) \\
U_2 &= \text{val}_{S_1}(c := c + 2) \\
C &= \{(f, (d_1, \ldots, d_n), d) \mid (f, (d_1, \ldots, d_n), d) \in U_1 \text{ and} \\
&\qquad\qquad\qquad\quad (f, (d_1, \ldots, d_n), d') \in U_2 \text{ for some } d' \neq d\}
\end{aligned}
$$

That is, we first evaluate the two parallel sub-updates, resulting in the sets $U_1 = \{(c, (), 1)\}$ and $U_2 = \{(c, (), 2)\}$ of consistent semantic updates. Both U_1 and U_2 fix the interpretation of c for the same (and only) argument tuple $()$ but in an inconsistent way; U_1 and U_2 assign c the values 1 and 2, respectively. Such a situation is called a *clash*. As a consequence, the union $U_1 \cup U_2$ is not a set of consistent semantics updates. To regain consistency we have to remove those elements from the union that cause the clash. In the example, that is the set

$$C = \{(c, (), 1)\} \;,$$

and we obtain as the result

$$\text{val}_{S_1}(c := c + 1 \,||\, c := c + 2) \;=\; \{(c, (), 2)\} \;.$$

This example shows that in case of a clash within a parallel update, the later sub-update dominates the earlier one such that the evaluation of the second sub-update is not affected by the first one. In contrast, with sequential updates the first sub-update affects the second sub-update.

Not surprisingly, defining the semantics of quantified updates is rather complicated and proceeds in two steps.

First, we determine the set $D = \{d \in \mathcal{D}^A \mid S, \beta_x^d \models \phi\}$ of domain elements $d \in \mathcal{D}^A$ satisfying the guard formula ϕ with the variable assignment β_x^d. Then, for each $a \in D$ the sub-update u is evaluated with β_x^a resulting in a

set of consistent semantic updates. If the quantified update is clash-free, its semantics is simply the union of all these sets of semantic updates.

In general though, a quantified update might contain clashes which must be resolved. For example, performing the steps described above for the quantified update

$$\texttt{for } x;\ x \doteq 0 \mid x \doteq 1;\ c := 5 - x$$

results in the two sets $U_1 = \{(c, (), 5)\}$ and $U_2 = \{(c, (), 4)\}$ of consistent semantic updates (the set of values satisfying the guard formula is $\{0, 1\}$). The set $U_1 \cup U_2$ is inconsistent, i.e., the quantified update is not clash-free and we have to resolve the clashes. For this purpose it is important to remember the value $a \in D$ from the variable assignment β_x^a under which the sub-update u was evaluated. Therefore, in Def. 3.27, we define a set $dom = \bigcup_{a \in \{d \in \mathcal{D}^A \mid S, \beta_x^d \models \phi\}} (\mathrm{val}_{S,\beta_x^a}(u) \times \{a\})$ consisting of pairs of (possibly inconsistent) semantics updates (resulting from $\mathrm{val}_{S,\beta_x^a}(u)$) and the appropriate value a (from β_x^a). In our example, the set dom is given as

$$dom = \{((c, (), 5), 0), ((c, (), 4), 1)\}$$

which we use for clash resolution. The clash is resolved by considering only one of the two semantic updates and discarding the other one. In general, to determine which one is kept the second components a, a' (here 0 and 1) from the elements in dom come into play: The semantic update $(f, (d_1, \ldots, d_n), d)$ with appropriate a is kept if $a \preceq a'$ for all $(f', (d_1', \ldots, d_n'), d')$ with $f = f'$, $n = m$, $d_i = d_i'$ ($1 \leq i \leq n$), and appropriate a'. That is, the semantic update arising from the least element satisfying the guard dominates. In our example we keep $(c, (), 5)$ and discard $(c, (), 4)$ since $0 \preceq 1$.

This "least element" approach for clash resolution in a sense carries over the last-win semantics of parallel updates to quantified updates. Note, that this is not the only possibility for clash resolution (see Note 3.30).

Example 3.29. In this example we show how clashes for quantified updates are resolved using the ordering predicate $\preceq$ on the domain.

Consider the update

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

It attempts to simultaneously assign the values 0 and 1 to the location $h(0)$. To keep the example simple, we assume that x ranges over the positive integers, which allows us to chose the usual "less than or equal" ordering relation $\leq$.

The semantics of a clashing quantified update is given by the least (second component of the) elements in the set dom with respect to the ordering. First, we determine the set dom (for an arbitrary state S):

$$
\begin{aligned}
dom &= \bigcup_{a \,\in\, \{d \in \mathcal{D}^A \mid S, \beta_x^d \models (x \doteq 0 \mid x \doteq 1)\}} (\mathrm{val}_{S,\beta_x^a}(h(0) := x) \times \{a\}) \\
&= \{\mathrm{val}_{S,\beta_x^0}(h(0) := x) \times \{0\}, \mathrm{val}_{S,\beta_x^1}(h(0) := x) \times \{1\}\} \\
&= \{((h, (0), 0), 0), ((h, (0), 1), 1)\}
\end{aligned}
$$

since $\mathrm{val}_{S,\beta_x^0}(h(0) := x) = \{(h,(0),0)\}$ and $\mathrm{val}_{S,\beta_x^1}(h(0) := x) = \{(h,(0),1)\}$. Then, in the second step, we remove those elements from *dom* that cause clashes. In the example, the two elements are inconsistent, and we keep only $((h,(0),0),0)$ since it has the smaller second component with respect to the ordering $\leq$. That is, the result of evaluating the quantified update is the singleton set $\{(h,(0),0)\}$ of consistent semantics updates.

As an example for a quantified update without clash consider

$$\texttt{for } x;\ x \doteq 0 \mid x \doteq 1;\ h(x) := 0\ ,$$

which we evaluate in some arbitrary state S. Again we assume that x ranges over the non-negative integers. Then,

$$\begin{aligned} dom &= \bigcup_{a \in \{d \in \mathcal{D}^A \mid S, \beta_x^d \models x \doteq 0 \mid x \doteq 1\}} (\mathrm{val}_{S,\beta_x^a}(h(x) := 0) \times \{a\}) \\ &= \{\mathrm{val}_{S,\beta_x^0}(h(x) := 0) \times \{0\}, \mathrm{val}_{S,\beta_x^1}(h(x) := 0) \times \{1\}\} \\ &= \{((h,(0),0),0), ((h,(1),0),1)\} \end{aligned}$$

This set *dom* does not contain inconsistencies and, thus, the semantics of the quantified update is $\{(h,(0),0),(h,(1),0)\}$.

Note 3.30. As already mentioned before there are several possibilities for defining the semantics of updates in case of a clash.

The crucial advantage of using a last-win clash semantics is that the transformation of sequential updates into parallel ones becomes almost trivial and can in practice be carried out very efficiently ($\Rightarrow$ Sect. 3.9.2). A last-win semantics allows to postpone case distinctions resulting from the possibility of aliasings/clashes to a later point in the proof.

Other possible strategies for handling clashes in quantified updates are (the basic ideas are mostly taken from the thesis by Platzer [2004b]):

- Leaving the semantics of updates undefined in case of a clash. This approach is similar to how partial functions (e.g., /) are handled in KeY. Then, a clashing update leads to a state where the location affected by the clash has a fixed but unknown value.
- Using the notion of *consistent* (syntactic) updates (as it is done in ASMs) in which no clashes occur. Following this idea, inconsistent updates would have no effect. However, according to the experiences with the existing version of KeY for ASMs [Nanchen et al., 2003], proving the consistency of updates tends to be tedious.
- Making the execution of updates containing clashes indeterministic (an arbitrary one of the clashing sub-updates is chosen). Then, however, updates would no longer be deterministic modal operators. Apart from the fact that the determinism of updates is utilised in a number of places in KeY, transformation rules for updates become much more involved for this clash semantics.

3.3.3 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 3.31 (Semantics of JAVA CARD DL terms). *Given a signature for a type hierarchy, let* $\mathcal{K}_{\preceq} = (\mathcal{M}, \mathcal{S}, \rho)$ *be a KeY JAVA CARD DL Kripke structure, let* β *be a variable assignment, and let* $P \in \Pi$ *be a normalised JAVA CARD program.*

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

$$\mathrm{val}_{S,\beta}(x) = \beta(x) \quad \text{for variables } x$$

$$\mathrm{val}_{S,\beta}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathrm{val}_{S,\beta}(t_1), \ldots, \mathrm{val}_{S,\beta}(t_n))$$

$$\mathrm{val}_{S,\beta}(\text{if } \phi \text{ then } t_1 \text{ else } t_2)) = \begin{cases} \mathrm{val}_{S,\beta}(t_1) & \text{if } S, \beta \models \phi \\ \mathrm{val}_{S,\beta}(t_2) & \text{if } S, \beta \not\models \phi \end{cases}$$

$$\mathrm{val}_{S,\beta}(\text{ifExMin } x.\phi \text{ then } t_1 \text{ else } t_2)) =$$

$$\begin{cases} \mathrm{val}_{S,\beta_x^d}(t_1) & \text{if there is some } d \in \mathcal{D}^A \text{ such that } S, \beta_x^d \models \phi \text{ and} \\ & d \preceq d' \text{ for any } d' \in \mathcal{D}^A \text{ with } S, \beta_x^{d'} \models \phi \\ & (\text{where } A \text{ is the type of } x) \\ \mathrm{val}_{S,\beta}(t_2) & \text{otherwise} \end{cases}$$

$$\mathrm{val}_{S,\beta}(\{u\}(t)) = \mathrm{val}_{S_1,\beta}(t) \quad \text{with } S_1 = \mathrm{val}_{S,\beta}(u)(S)$$

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. From Def. 3.18 follows, that 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 3.32. *A JAVA CARD DL term* t *is* rigid

- *if* $t = x$ *and* $x \in \mathrm{VSym}$,
- *if* $t = f(t_1, \ldots, t_n)$, $f \in \mathrm{FSym}_r$ *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. 3.35) and the sub-terms* t_1, t_2 *are rigid*,
- *if* $t = (\text{ifExMin } x.\phi \text{ then } t_1 \text{ else } t_2)$ *and the formula* ϕ *is rigid (Def. 3.35) and the sub-terms* t_1, t_2 *are rigid*.

Intuitively, 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).

Lemma 3.33. *Let $\mathcal{K}_\preceq = (\mathcal{M}, \mathcal{S}, \rho)$ be a KeY* Java Card *DL Kripke structure, let $P \in \Pi$ be a normalised* Java Card *program, and let β be a variable assignment.*

If Java Card *DL term t is rigid, then*

$$\mathrm{val}_{S_1,\beta}(t) = \mathrm{val}_{S_2,\beta}(t)$$

for any two states $S_1, S_2 \in \mathcal{S}$.

The proof of the above lemma proceeds by induction on the term structure and makes use of the fact, that by definition the leading function symbol f of a term to be updated must be from the set FSym_{nr}.

3.3.4 Semantics of Java Card DL Formulae

Definition 3.34 (Semantics of Java Card DL formulae). *Given a signature for a type hierarchy, let $\mathcal{K}_\preceq = (\mathcal{M}, \mathcal{S}, \rho)$ be a KeY* Java Card *DL Kripke structure, let β be a variable assignment, and let $P \in \Pi$ be a normalised* Java Card *program.*

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

- $S, \beta \models p(t_1, \ldots, t_n)$ *iff* $(\mathrm{val}_{S,\beta}(t_1), \ldots, \mathrm{val}_{S,\beta}(t_n)) \in \mathcal{I}(p)$
- $S, \beta \models \mathrm{true}$
- $S, \beta \not\models \mathrm{false}$
- $S, \beta \models\, !\,\phi$ *iff* $S, \beta \not\models \phi$
- $S, \beta \models (\phi\ \&\ \psi)$ *iff* $S, \beta \models \phi$ *and* $S, \beta \models \psi$
- $S, \beta \models (\phi\ |\ \psi)$ *iff* $S, \beta \models \phi$ *or* $S, \beta \models \psi$ *(or both)*
- $S, \beta \models (\phi \mathbin{-\!\!>} \psi)$ *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* $d \in \mathcal{D}^A$ *(where A is the type of x)*
- $S, \beta \models \exists x.\phi$ *iff* $S, \beta_x^d \models \phi$ *for some* $d \in \mathcal{D}^A$ *(where A is the type of x)*
- $S, \beta \models \{u\}(\phi)$ *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 $(S, p, S') \in \rho$ and $S', \beta \models \phi$*
- $S, \beta \models [p]\phi$ *iff $S', \beta \models \phi$ for every state $S' \in \mathcal{S}$ with $(S, p, S') \in \rho$*

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

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

Definition 3.35. *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 $(1 \leq i \leq n)$,*
- *if $\phi = \mathrm{true}$ or $\phi = \mathrm{false}$,*
- *if $\phi =\, !\,\psi$ and ψ is rigid,*
- *$\phi = (\psi_1\ |\ \psi_2)$, $\phi = (\psi_1\ \&\ \psi_2)$, or $\phi = (\psi_1 \mathbin{-\!\!>} \psi_2)$, 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.*

Note 3.36. 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 behaviour 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.

Intuitively, rigid formulae—in contrast to non-rigid formulae—have the same meaning in all JAVA CARD DL states.

Lemma 3.37. *Let $\mathcal{K}_{\preceq} = (\mathcal{M}, \mathcal{S}, \rho)$ be a KeY JAVA CARD DL Kripke structure and let $P \in \Pi$ be a normalised JAVA CARD program, and let β be a variable assignment.*

If a JAVA CARD DL formula ϕ is rigid, then

$$S_1, \beta \models \phi \text{ if and only if } S_2, \beta \models \phi$$

for any two states $S_1, S_2 \in \mathcal{S}$.

Finally, we define what it means for a formula to be valid or satisfiable. A first-order formula is satisfiable (resp., valid) if it holds in some (all) model(s) for some (all) variable assignment(s) ($\Rightarrow$ Def. 2.40). Similarly, a JAVA CARD DL formula is satisfiable (resp. valid) if it holds in some (all) state(s) of some (all) Kripke structure(s) $\mathcal{K}_{\preceq}$ for some (all) variable assignment(s).

Definition 3.38. *Given a signature for a type hierarchy and a normalised JAVA CARD program $P \in \Pi$, let ϕ be a JAVA CARD DL formula.*

ϕ is satisfiable if there is a KeY JAVA CARD DL Kripke structure $\mathcal{K}_{\preceq} = (\mathcal{M}, \mathcal{S}, \rho)$ such that $S, \beta \models \phi$ for some state $S \in \mathcal{S}$ and some variable assignment β.

Given a KeY JAVA CARD DL Kripke structure $\mathcal{K}_{\preceq} = (\mathcal{M}, \mathcal{S}, \rho)$, the formula ϕ is $\mathcal{K}_{\preceq}$-valid, denoted by $\mathcal{K}_{\preceq} \models \phi$, if $S, \beta \models \phi$ for all states $S \in \mathcal{S}$ and all variable assignments β

ϕ is logically valid, denoted by $\models \phi$, if $\mathcal{K}_{\preceq} \models \phi$ for all KeY JAVA CARD DL Kripke structures $\mathcal{K}_{\preceq}$.

Note 3.39. Satisfiability and validity for JAVA CARD DL coincide with the corresponding notions in first-order logic (Def. 2.40), i.e., if a first-order formula ϕ is satisfiable (valid) in first-order logic then ϕ is satisfiable (valid) in JAVA CARD DL.

Note 3.40. The notions of satisfiability, $\mathcal{K}_{\preceq}$-validity, and logical validity of a formula depend on the given type hierarchy and normalised JAVA CARD program, and are not preserved if one of the two is modified—as the following simple example shows.

Suppose a type hierarchy contains an abstract type A with Null as its only subtype. Then the formula $\forall x.x \doteq \texttt{null}$, where x is of type A, is valid since $\mathcal{D}^A$ consists only of the element *null* to which the term $\texttt{null}$ evaluates.

Now we modify the type hierarchy and add a dynamic type B that is a subtype of A and a supertype of Null. By definition, the domain of a dynamic type is non-empty, and, since B is a subtype of A, $\mathcal{D}^A$ contains at least one element $d \neq null$ with $d \in \mathcal{D}^B$. As a consequence, ϕ is not valid in the modified type hierarchy.

The following example shows that validity does not only depend on the given type hierarchy but also on the JAVA CARD DL program (which is, of course, more obvious). Suppose the type hierarchy contains the dynamic type *Base* with dynamic subtype *SubA*, and the normalised JAVA CARD program shown below is given:

─── JAVA ───

```
   class Base {
2    int m() {
       return 0;
4    }

6    public static void start(Base o) {
       int i=o.m();
8    }
   }
10
   class SubA extends Base {
12   int m() {
       return 0;
14   }
   }
```

─── JAVA ───

Consider the method invocation `o.m();` in line 7. Both the implementation of `m` in class `Base` and the one in class `SubA` may be chosen for execution. The choice depends on the dynamic type of the domain element that `o` evaluates to—resulting in a case distinction.

Nevertheless, the formula

$$\langle \texttt{i=Base.start(o);} \rangle \texttt{i} \doteq 0 \ ,$$

where $o{:}Base \in \mathrm{FSym}_{nr}$, is valid because both implementations of `m()` return 0.

Now we modify the implementation of method `m()` in class `SubA` by replacing the statement `return 0;` with `return 1;`. Then, ϕ is no longer valid since now there are values of `o` for which method invocation `o.m();` yields the return value 1 instead of 0.

Instead of modifying the implementation of method `m()` in class `SubA` one could also add another class `SubB` extending `Base` with an implementation of `m()` that return a "wrong" result, making ϕ invalid.

The examples given here clearly show that in general the validity of a formula depends on the given type hierarchy and on the program that is considered. As is explained in Sect. 8.5, as a consequence, any proof is in principle invalidated if the program is modified (e.g., by adding a new subclass) and needs to be redone. However, validity is not always lost if the program is modified and Sect. 8.5 presents methods for identifying situations where it is preserved.

Example 3.41. We now check the formulae from Example 3.15 for validity.

$\models \{c := 0\}\,(c \doteq 0)$ since in the state in which $c \doteq 0$ is evaluated, c is indeed 0 (due to the update).

$\not\models (\{c := 0\}\,c) \doteq c$ since $(\{c := 0\}c)$ evaluates to 0 in any state but there are states in which c (the right side) is different from 0.

$\models$ `sal !`$\doteq$ `null` $\rightarrow$ $\langle$`ArrayList.demo2(sal);`$\rangle$`j` $\doteq 1$ since after invocation of `ArrayList.demo2(sal)` with an argument `sal` different from `null` the local variable `j` has the value 1.

$\not\models \{$`sal` $:=$ `null`$\}\langle$`ArrayList.demo2(sal);`$\rangle$`j` $\doteq 1$ since `j` has the value 0 when the program terminates if started in a state with `sal` $\doteq$ `null`. Due to the update `sal :=` only such states are considered and, thus, this formula is even unsatisfiable.

$\not\models \{$`v` $:= g\}(\langle$`ArrayList al=new ArrayList(); v=al.inc(v);`$\rangle$`v` $\doteq g+1)$ since the JAVA CARD addition `arg+1` in method `inc` causes a so-called *overflow* for $n = 2147483647$ ($\Rightarrow$ Chap. 12), in which case `v` has the negative value $-2147483648 \neq 2147483647 + 1$.

$\not\models \{$`v` $:= g\}(\langle$`ArrayList al=new ArrayList(); v=al.inc(v)@ArrayList`$\rangle$`v` $\doteq g+1)$ This formula has the same semantics as the previous one since in this case the method-body statement `v=al.inc(v)@ArrayList` is equivalent to the method call `v=al.inc(v)` (because there are no subclasses of `ArrayList` overriding method `inc`).

$\models \langle$`int v=0;`$\rangle$`v` $\doteq 0$ since the program always terminates in state with `v` $\doteq 0$.

Example 3.42. The following two examples deal with functions that have a predefined fixed semantics (i.e., the same semantics in all Kripke seeds) only on parts of the domain. We consider the division function / (written infix in the following), which has a predefined interpretation for $\{(x,y) \in$ `integer` $\times$ `integer` $\mid y \neq 0\}$ while the interpretation for $\{(x,y) \in$ `integer` $\times$ `integer` $\mid y = 0\}$ depends on the particular Kripke seed.

$\mathcal{K}_{\preceq} \models 5/c \doteq 1$ in any $\mathcal{K}_{\preceq}$ with Kripke seed $\mathcal{M} = (\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$
                       such that
                       • $\mathcal{I}_0(c) = 5$ or
                       • $\mathcal{I}_0(c) = 0$ and $\mathcal{I}_0(5/0) = 1$.
$\models 5/c \doteq 5/c$ since $5/c \doteq 5/c$ holds in any state S of any $\mathcal{K}_{\preceq}$ (even
                       if $\text{val}_S(c) = 0$).

3.3.5 Java Card-Reachable States

As mentioned before, the set of states that are reachable by Java Card
programs is a subset of the states of a Java Card DL Kripke structure.
Indeed, a state is (only) Java Card-reachable if it satisfies the following
conditions:

1. A finite number of objects are created.[4]
2. Reference type attributes of non-null objects are either null or point to
   some other created object. Similarly, all entries of reference-type arrays
   different from null are either null or point to some created object.
3. For any array a the dynamic type $\delta(a[i])$ of the array entries is a subtype
   of the element type A of the dynamic type $A[\,] = \delta(a)$ of a (violating this
   condition in Java leads to an `ArrayStoreException`).

Given a type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$, a signature, and a normalised Java
Card program $P \in \Pi$ the above conditions can be expressed with Java
Card DL formulae as follows (the implicit fields like, e.g., `<nextToCreate>`
and the function T::get() used in the following formulae are defined formally
in Sect. 3.6.6):

1. For all dynamic types $T \in \mathcal{T}_d \setminus \{\text{Null}\}$ with $T \sqsubseteq \text{Object}$ that occur in P:

$$T.\texttt{<nextToCreate>} >= 0 \;\&$$
$$\forall x. (x >= 0 \;\&\; x < T.\texttt{<nextToCreate>} <->$$
$$\quad \text{T::get}(x).\texttt{<created>} \doteq \text{TRUE})$$

   where $x \in \text{VSym}$ is of type `integer`.
   By definition, an object o is created iff its index (which is the in-
   teger i for which the equation $o \doteq \text{C::get}(i)$ holds) is in the inter-
   val $[0, C.\texttt{<nextToCreate>}[$. This guarantees that only a finite number
   of objects are created. The above formula expresses the consistency be-
   tween the implicit attribute T::get(x).`<created>` and the definition of
   createdness for any type T.

---

[4] In Java Card DL, objects are represented by domain elements, and the domain
is assumed to be constant (see Note 3.19 on page 89). Whether an object is
created or not is indicated by a Boolean function `<created>` ($\Rightarrow$ Sect. 3.6.6).

2. (i) For all types $T \in \mathcal{T} \setminus \{\bot, \text{Null}\}$ with $T \sqsubseteq$ Object that occur in P and for all non-rigid function symbols $f : T \to T' \in \text{FSym}_{nr}$ with $T' \sqsubseteq$ Object that are declared as an attribute of T in P:

$$\forall o.(o.\texttt{<created>} \doteq \text{TRUE} \ \& \ o \mathbin{!\dot{=}} \mathit{null}) \to$$
$$(o.f \doteq \mathit{null} \mid o.f.\texttt{<created>} \doteq \text{TRUE}))$$

where $o \in \text{VSym}$ is of type T.

(ii) For all array types $T[\,] \in \mathcal{T}$ that occur in P

$$\forall a.\forall x.(a.\texttt{<created>} \doteq \text{TRUE} \ \& \ a \mathbin{!\dot{=}} \mathit{null}) \to$$
$$(a[x] \doteq \mathit{null} \mid a[x].\texttt{<created>} \doteq \text{TRUE}))$$

where $a \in \text{VSym}$ is of type $T[\,]$ and $x \in \text{VSym}$ of type `integer`.

3. For all array types $T[\,] \in \mathcal{T}$ that occur in P:

$$\forall a.\forall x.((a.\texttt{<created>} \doteq \text{TRUE} \ \& \ a \mathbin{!\dot{=}} \mathit{null}) \to$$
$$\mathit{arrayStoreValid}(a, a[x]))$$

where $a \in \text{VSym}$ is of type $T[\,]$ and $x \in \text{VSym}$ of type `integer` (see App. A.2.2 for the semantics of the predicate *arrayStoreValid*).

Thus, there is a reachability formula for each type T, each reference type attribute, and each array type that occurs in the program P. Since the conjunction of all these may result in a quite lengthy formula, we introduce the non-rigid predicate *inReachableState* which, by definition, holds in a state S iff all the above formulae hold in S.

There are some more constraints restricting the set of Java Card-reachable states dealing with class initialisation. For example, an initialised class is not erroneous. However, since class initialisation is not handled in this book we do not go into details here and omit the corresponding constraints (for a detailed account on class initialisation the reader is referred to [Bubel, 2001]). Please note that the KeY system can handle class initialisation.

Example 3.43. Consider the following Java Card program

```
——— Java ———
class Control {
  Data data;
}

class Data {
  int d;
}
                                            ——— Java ———
```

We assume a specification of class `Data` that consists of the invariant

$$\forall data.(data.\texttt{<created>} \doteq \text{TRUE} \rightarrow data.\texttt{d} >= 0)$$

(where $data \in \text{VSym}$ is of type `Data`), stating that, for all created objects of type `Data`, the value of the attribute `d` is non-negative.

Now, we would like to prove that for any object c of type `Control` that is created and different from `null`, the value of the attribute `c.data.d` is non-negative. With the semantics of JAVA CARD in mind, this seems to be a valid property given the invariant of class `Data`. However, the corresponding JAVA CARD DL formula

$$\forall data.(data.\texttt{<created>} \doteq \text{TRUE} \rightarrow data.\texttt{d} >= 0) \rightarrow$$
$$(\texttt{c.<created>} \doteq \text{TRUE} \ \& \ \texttt{c} \ != \ \texttt{null} \ \& \ \texttt{c.data} \ != \ \texttt{null} \rightarrow$$
$$\texttt{c.data.d} >= 0)$$

is (surprisingly) not valid. The reason is that we cannot establish the assumption of the invariant of class `Data`, since we cannot prove that the equation `c.data.<created>` $\doteq$ TRUE holds, i.e., that `c.data` refers to a created object (even if we know that `c.data` is different from `null`). In JAVA CARD it is always true that a non-null attribute of a created non-null object points to a created object. In our logic, however, we have to make this explicit by adding the assumption *inReachableState* stating that we are in a JAVA CARD-reachable state. We obtain

$$(\textit{inReachableState} \ \&$$
$$\forall data.(data.\texttt{<created>} \doteq \text{TRUE} \rightarrow data.\texttt{d} >= 0)) \rightarrow$$
$$(\texttt{c.<created>} \doteq \text{TRUE} \ \& \ \texttt{c} \ != \ \texttt{null} \ \& \ \texttt{c.data} \ != \ \texttt{null} \rightarrow$$
$$\texttt{c.data.d} >= 0)$$

One conjunct of *inReachableState* is the formula

$$\forall o.(o.\texttt{<created>} \doteq \text{TRUE} \ \& \ o \ != \ \texttt{null}) \rightarrow$$
$$(o.\texttt{data} \doteq \texttt{null} \ | \ o.\texttt{data.<created>} \doteq \text{TRUE}))$$

where $o \in \text{VSym}$ is of type `Data`. If we instantiate the universal quantifier in this formula with `c` we can derive the desired equation

$$\texttt{c.data.<created>} \doteq \text{TRUE} \ \ .$$

This example shows that there are formulae 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 the predicate *inReachableState* to the invariants of the program to be verified. Then, states that are not reachable by any JAVA CARD program are excluded from consideration.

Dealing with the inReachableState Predicate in Proofs

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*:

(i) The proof can only be closed by employing (parts of) the properties stated in *inReachableState* (as in Example 3.43).
(ii) A state (described by some update) must be shown to satisfy the predicate *inReachableState*. Such a situation occurs, for example, when using a method contract (i.e., the specification of a method) in a proof. Then it is necessary to establish the precondition of the method specification, which usually contains the predicate *inReachableState*, in the invocation state (⇒ Sect. 3.8).

In both situations, expanding *inReachableState* into its components is not feasible since in practice the resulting formula would consist of hundreds or thousands of conjuncts.

To deal with situation (i), the KeY calculus provides rules that allow the user to extract parts of *inReachableState* that are necessary to close the proof.

To prevent full expansion of *inReachableState* in the case that

$$inReachableState \rightarrow \{u\}\ inReachableState$$

must be shown for some update u (situation (ii)), the KeY system performs a syntactic analysis of the update u and expands only those parts of *inReachableState* that possibly are affected by u.

Note, that in general an update u that results from the symbolic execution of some program cannot describe a state that violates *inReachableState*. However, the user might provide such a malicious update that leads to an unreachable state by, for example, applying the cut rule (⇒ Sect. 3.5.2).

3.4 The Calculus for Java Card DL

3.4.1 Sequents, Rules, and Proofs

The KeY system's calculus for Java Card DL is a sequent calculus. It extends the first-order calculus from Chapter 2.

Sequents are defined as in the first-order case (Def. 2.42). The only difference is that now the formulae in the sequents are Java Card DL formulae.

Definition 3.44. *A sequent is of the form $\Gamma \Longrightarrow \Delta$, where Γ, Δ are sets of closed Java Card DL formulae.*

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

As in the first-order case, the semantics of a sequent

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

is the same as that of the formula

$$(\phi_1 \,\&\, \ldots \,\&\, \phi_m) \mathbin{-\!\!>} (\psi_1 \mid \ldots \mid \psi_m) \ .$$

In Chapter 2 we have used an informal notion of what a rule is, and what a rule application is. Now, we give a more formal definition.

Definition 3.45. *A rule R is a binary relation between (a) the set of all tuples of sequents and (b) the set of all sequents.*

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

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

A *calculus*—in particular our Java Card DL calculus—is formally a set of rules.

Proof trees are defined as in the first-order case (Def. 2.50), except that now the rules of the Java Card DL calculus (as described in Sections 3.5–3.9) are used for derivation instead of the first-order rules. 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.

3.4.2 Soundness and Completeness of the Calculus

Soundness

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

Proposition 3.46 (Soundness). *If a sequent $\Gamma \Longrightarrow \Delta$ is derivable in the Java Card DL calculus (Def. 3.45), then it is valid, i.e., the formula $\bigwedge \Gamma \mathbin{-\!\!>} \bigvee \Delta$ is logically valid (Def. 3.38).*

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 ($\Rightarrow$ Sect. 4.5).

Validating the Soundness of the Java Card DL Calculus

So far, we have no intention of formally proving the soundness of the Java Card DL calculus, i.e., the core rules that are not user-defined (the soundness of user-defined rules can be verified within the KeY system, see Sect. 4.5). Doing so would first require a formal specification of the Java Card language. No *official* formal semantics of Java or Java Card is available though. Furthermore, proving soundness of the calculus requires the use of a higher-order theorem proving tool, and it is a tedious task due to the high number of rules. Resources saved on a formal soundness proof were instead spent on further improvement of the KeY system. We refer to [Beckert and Klebanov, 2006] for a discussion of this policy and further arguments in its favour. On the other hand, the KeY project performs ongoing cross-verification against other Java formalisations to ensure the faithfulness of the calculus.

One such effort compares the KeY calculus with the Bali semantics [von Oheimb, 2001a], which is a Java Hoare logic formalised in Isabelle/HOL. KeY rules are translated manually into Bali rules. These are then shown sound with respect to the rules of the standard Bali calculus. The published result [Trentelman, 2005] describes in detail the examination of the rules for local variable assignment, field assignment and array assignments.

Another validation was carried out by Ahrendt et al. [2005b]. A reference Java semantics from [Farzan et al., 2004] was used, which is formalised in Rewriting Logic [Meseguer and Rosu, 2004] and mechanised in the input language of the Maude system. This semantics is an executable specification, which together with Maude provides a Java interpreter. Considering the nature of this semantics, we concentrated on using it to verify our program transformation rules. 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 calculus rules. 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 calculus is constantly extended with new features.

Furthermore, the KeY calculus is regularly tested against the compiler test suite Jacks (available at `www.sourceware.org/mauve/jacks.html`). 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.

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] (discussed in Sect. 2.7). 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 formulae 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 3.47 (Relative Completeness). *If a sequent $\Gamma \Longrightarrow \Delta$ is valid, i.e., the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is logically valid (Def. 3.38), then there is a finite set Γ_{FOL} of logically valid first-order formulae such that the sequent*

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

is derivable in the JAVA CARD DL calculus.

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 formulae there is an equivalent purely first-order formula. A completeness proof for the object-oriented dynamic logic ODL [Beckert and Platzer, 2006], which captures the essence of JAVA CARD DL, is given by Platzer [2004b].

3.4.3 Rule Schemata and Schema Variables

The following definition makes use of the notion of *schema variables*. They represent concrete syntactical elements (e.g., terms, formulae or programs). Every schema variable is assigned a kind that determines which class of concrete elements is represented by such a schema variable.

Definition 3.48. *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.

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, formulae, 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.
- Formulae are added to a sequent. The number of formulae 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 formulae in the conclusion.

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

Thus, all our rules preserve the "context" in a sequent, i.e., the formulae 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 formalised in Def. 3.49). 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.

Later in the book (e.g., Sect. 3.7) we will present a few rules where the context cannot be omitted. Such rules are indicated with the $(*)$ symbol. These rules will be shown for comparison only; they are not part of the JAVA CARD DL calculus.

Definition 3.49. *If*

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

is an instance of a rule schema, then

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

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 formulae.

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.

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 3.1 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 formulae and Γ, Δ to represent sets of formulae. Schema variables of corresponding kinds occur also in the *taclets* used to implement rules in the KeY system ($\Rightarrow$ Sect. 4.2).

If a schema variable T representing a type expression is indexed with the name of another schema variable, say e, then it only maches 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).

3.4.4 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,

Table 3.1. Correspondence between names of schema variables and their kinds

π	non-active prefix of JAVA code (Sect. 3.4.4)
ω	"rest" of JAVA code after the active statement (Sect. 3.4.4)
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 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, …	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

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:

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

> **No Rule for Sequential Composition**
>
> 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 $\langle \pi pq\omega \rangle \phi$.

3.4.5 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 formulae. Since JAVA CARD DL formulae 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

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

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:

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

This way, we have reduced the 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 analysing 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:

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

Thus, we can state the essence of symbolic execution: the JAVA code in the formulae 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 ($\Rightarrow$ Chap. 11) or loop invariants ($\Rightarrow$ Sect. 3.7.1). 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.

3.4.6 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. 3.7) and some of the rules for handling abrupt termination (Sect. 3.6.7).

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. Rules that handle loops for which no fixed upper bound on the number of iterations exists. In this chapter, we only consider rules that handle loops using loop invariants (Sect. 3.7). A separate chapter is dedicated to handling loops by induction (Chapter 11).
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 formulae. 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 formulae. These are then handled by Component 1.

The symbolic execution process is for the most part done automatically by the KeY system. Usually, only handling loops and methods may require user interaction. Also, for solving the first-order problem that is left at the end of the symbolic execution process, the KeY system often needs support from the user (or from the decision procedures integrated into KeY, see Chapter 10).

3.5 Calculus Component 1: Non-program Rules

3.5.1 First-Order Rules

Since first-order logic is part of JAVA CARD DL, all the rules of the first-order calculus introduced in Chapter 2 are also part of the JAVA CARD DL calculus. That is, all rules from Fig. 2.2 (classical first-order rules), Fig. 2.3 (equality rules), Fig. 2.4 (typing rules), and Fig. 2.5 (arithmetic rules) can be applied to JAVA CARD DL sequents—even if the formulae that they are applied to are not purely first-order.

Consider, for example, the rule impRight. In Chapter 2, the rule schema for this rule takes the following form:

$$\text{impRight (Chapter 2 notation)} \quad \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \rightarrow \psi, \Delta}$$

In this chapter, we omit the context Γ, Δ from rule schemata ($\Rightarrow$ Sect. 3.4.3), i.e., the same rule is schematically written as:

$$\text{impRight} \quad \frac{\phi \Longrightarrow \psi}{\Longrightarrow \phi \rightarrow \psi}$$

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 formulae that are not purely first-order. For example, the following is an instance of impRight:

$$\frac{x \doteq 1, \ \{x := 0\}(x \doteq y) \Longrightarrow \{x := 0\}\langle \mathtt{m()};\rangle(y \doteq 0)}{x \doteq 1 \Longrightarrow \{x := 0\}(x \doteq y \rightarrow \langle \mathtt{m()};\rangle(y \doteq 0))}$$

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

Due to the presence of modalities and non-rigid functions, which do not exist in purely first-order formulae, different parts of a formula may have to be evaluated in different states. Therefore, the application of some first-order rules that rely on the identity of terms in different parts of a formula needs to be restricted. That affects rules for universal quantification and equality rules.

Restriction of Rules for Universal Quantification

The rules for universal quantification have the following form:

$$\text{allLeft } \frac{\forall x.\phi, \ [x/t](\phi) \Longrightarrow}{\forall x.\phi \Longrightarrow} \qquad \text{exRight } \frac{\Longrightarrow \exists x.\phi, \ [x/t](\phi)}{\Longrightarrow \exists x.\phi}$$

where $t \in \text{Trm}_{A'}$ is a rigid ground term
whose type A' is a sub-type of the type A of x

In the first-order case, the term t that is instantiated for the quantified variable x can be an arbitrary ground term. In JAVA CARD DL, however, we have to add the restriction that t is a *rigid* ground term (Def. 3.32). The reason is that, though an arbitrary value can be instantiated for x as it is universally quantified, in each individual instantiation, all occurrences of x must have the same value.

Example 3.50. The formula $\forall x.(x \doteq 0 \rightarrow \langle \texttt{i++;}\rangle (x \doteq 0))$ is logically valid, but instantiating the variable x with the non rigid constant $\texttt{i}$ is wrong as it leads to the unsatisfiable formula $\texttt{i} \doteq 0 \rightarrow \langle \texttt{i++;}\rangle (\texttt{i} \doteq 0)$.

In practice, it is often very useful to instantiate a universally quantified variable x with the value of a non-rigid term t. That, however, is not easily possible as x must not be instantiated with a non-rigid term. In that case, one can add the logically valid formula $\exists y.(y \doteq t)$ to the left of the sequent, Skolemise that formula, which yields $c_{sk} \doteq t$, and then instantiate x with the rigid constant c_{sk} (this is done using the rule substToEq).

Rules for existential quantification do not have to be restricted because they introduce *rigid* Skolem constants anyway.

Restriction of Rules for Equalities

The equality rules (Fig. 2.3) are part of the JAVA CARD DL calculus but an equality $t_1 \doteq t_2$ may only be used for rewriting if

- both t_1 and t_2 are rigid terms (Def. 3.32), or
- the equality $t_1 \doteq t_2$ 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 syntactically

identical updates (in fact, it is also sufficient if the two updates are only semantically identical, i.e., have the same effect). This same-update-level property is explained in more detail in Sect. 4.4.1.

Example 3.51. The sequent

$$x \doteq v + 1 \Longrightarrow \{v := 2\}(v + 1 \doteq 3)$$

is valid. But applying the equality to the occurrence of $v + 1$ on the right side of the sequent is wrong, as it would lead to the sequent

$$x \doteq v + 1 \Longrightarrow \{v := 2\}(x \doteq 3)$$

that is satisfiable but *not valid.*

In the sequent

$$\{v := 2\}(x \doteq v + 1) \Longrightarrow \{v := 2\}(v + 1 \doteq 3) \ ,$$

however, both the equality and the term being replaced occur in the scope of identical updates and, thus, the equality rule can be applied.

3.5.2 The Cut Rule and Lemma Introduction

The cut rule

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

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 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 $\Longrightarrow ! \phi$.

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 his or her 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 ($\Rightarrow$ Sect. 4.5). That way, a lemma ϕ can be introduced that can be used in any context (provided that ϕ is shown to be logically valid).

3.5.3 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 like JAVA CARD.

3.6 Calculus Component 2: Reducing JAVA Programs

3.6.1 The Basic Assignment Rule

In JAVA—like in other object-oriented programming languages—different object variables can refer to the same object. This phenomenon, called aliasing, causes serious 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 $\text{o1.a} \doteq 1$ 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 classical 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 (see Sect. 3.2 for a discussion and a comparison of updates with assignments and substitutions).

The KeY system uses special simplification rules to compute the result of applying an update to terms and formulae that do not contain programs ($\Rightarrow$ Sect. 3.9). 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:

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

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*. 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 $\{\texttt{i := i+1}\}\langle \texttt{x = i;}\rangle\phi$ and then $\{\texttt{i := i+1}\}\{\texttt{x := i}\}\langle\rangle\phi$.

3.6.2 Rules for Handling General Assignments

There are four classes of rules in the JAVA CARD DL calculus for treating general assignment expressions (that may have side-effects). These classes—corresponding to steps in the evaluation of an assignment—are induced by the evaluation order rules of JAVA:

1. Unfolding the left-hand side of the assignment.
2. Saving the location.
3. Unfolding the right-hand side of the assignment.
4. Generating an update.

Of particular importance is the fact that though the right-hand side of an assignment can change the variables appearing in the left hand side, it cannot change the location scheduled for assignment, which is saved before the right-hand side is evaluated.

Step 1: Unfolding the Left-Hand Side

In this first step, the left-hand side of an assignment is unfolded if it is a non-simple expression, i.e., if its evaluation may have side-effects. One of the following rules is applied depending on the form of the left-hand side expression. In general, these rules work by introducing a new local variable v_0, to which the value of a sub-expression is assigned.

If the left-hand side of the assignment is a non-atomic field access—which is to say it has the form *nse*.*a*, where *nse* is a non-simple expression—then the following rule is used:

$$\text{assignmentUnfoldLeft} \ \frac{\Longrightarrow \langle \pi \ \ T_{nse} \ \ v_0 \texttt{=} nse \texttt{;} \ \ v_0 \texttt{.} a \texttt{=} e \texttt{;} \ \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ \ nse \texttt{.} a \texttt{=} e \texttt{;} \ \ \omega \rangle \phi}$$

Applying this rule yields an equivalent but simpler program, in the sense that the two new assignments have simpler left-hand sides, namely a local variable resp. an atomic field access.

Unsurprisingly, in the case of arrays, two rules are needed, since both the array reference and the index have to be treated. First, the array reference is analysed:

assignmentUnfoldLeftArrayReference

$$\frac{\Longrightarrow \langle \pi \ \ T_{nse} \ \ v_0 \ \texttt{=} \ nse \texttt{;} \ \ v_0 \texttt{[} e \texttt{]=} e_0 \texttt{;} \ \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ \ nse \texttt{[} e \texttt{]=} e_0 \texttt{;} \ \ \omega \rangle \phi}$$

Then, the rule for analysing the array index can be applied:

assignmentUnfoldLeftArrayIndex

$$\frac{\Longrightarrow \langle \pi \ \ T_v \ \ v_a \ \texttt{=} \ v \texttt{;} \ \ T_{nse} \ \ v_0 \ \texttt{=} \ nse \texttt{;} \ \ v_a \texttt{[} v_0 \texttt{]=} e \texttt{;} \ \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ \ v \texttt{[} nse \texttt{]=} e \texttt{;} \ \ \omega \rangle \phi}$$

Step 2: Saving the Location

After the left-hand side has been unfolded completely (i.e., has the form v, $v \texttt{.} a$ or $v \texttt{[} se \texttt{]}$), the right-hand side has to be analysed. But before doing this, we have to memorise the location designated by the left-hand side. The reason is that the location affected by the assignment remains fixed even if evaluating the right-hand side of the assignment has a side effect changing the location to which the left-hand side points. For example, if $i \doteq 0$, then `a[i] = ++i;` has to update the location `a[0]` even though evaluating the right-hand side of the assignment changes the value of `i` to 1.

Since there is no universal "location" or "address-of" operator in JAVA, this memorising looks different for different kinds of expressions appearing on the left. The choice here is between field resp. array accesses. For local variables, the memorising step is not necessary, since the "location value" of a variable is syntactically defined and cannot be changed by evaluating the right-hand side.

We will start with the rule variant where a field access is on the left. It takes the following form; the components of the premiss are explained in Table 3.2:

assignmentSaveLocation

$$\frac{\Longrightarrow \langle \pi \ \ check \texttt{;} \ \ memorise \texttt{;} \ \ unfoldr \texttt{;} \ \ update \texttt{;} \ \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ \ v \texttt{.} a \texttt{=} nse \texttt{;} \ \ \omega \rangle \phi}$$

There is a very similar rule for the case where the left-hand side is an array access, i.e., the assignment has the form $v \texttt{[} se \texttt{]=} nse$. The components of the premiss for that case are shown in Table 3.3.

Table 3.2. Components of rule assignmentSaveLocation for field accesses $v.a$=nse

check	`if (`v`==null) throw new`	abort if v is `null`
	`NullPointerException();`	
memorise	T_v v_0 = v;	
unfoldr	T_{nse} v_1 = nse;	set up Step 3
update	$v_0.a$ = v_1;	set up Step 4

Table 3.3. Components of rule assignmentSaveLocation for array accesses v[se]=nse

check	`if (`se`>=`v`.length || `se`<0) throw new`	abort if index
	`ArrayIndexOutOfBoundsException();`	out of bounds[a]
memorise	T_v v_0 = v; T_{se} v_1 = se;	
unfoldr	T_{nse} v_2 = nse;	set up Step 3
update	v_0[v_1] = v_2;	set up Step 4

[a] This includes an implicit test that v is not `null` when v.`length` is analysed.

Step 3: Unfolding the Right-Hand Side

In the next step, after the location that is changed by the assignment has been memorised, we can analyse and unfold the right hand side of the expression. There are several rules for this, depending on the form of the right-hand side. As an example, we give the rule for the case where the right-hand side is a field access $nse.a$ with a non-simple object reference nse:

$$\text{assignmentUnfoldRight} \quad \frac{\Longrightarrow \langle \pi \ T_{nse} \ v_0 \ = \ nse; \ v \ = \ v_0.a; \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ v \ = \ nse.a; \ \omega \rangle \phi}$$

The case when the right-hand side is a method call is discussed in the section on method calls ($\Rightarrow$ Sect. 3.6.5).

Step 4: Generate an Update

The fourth and final step of treating assignments is to turn them into an update. If both the left- and the right-hand side of the assignment are simple expressions, the basic assignment rule applies:

$$\text{assignment} \quad \frac{\Longrightarrow \{lhs := se^*\}\langle \pi \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ lhs \ = \ se; \ \omega \rangle \phi}$$

The value se^* appearing in the update is not identical to the se in the program because creating the update requires replacing any JAVA operators in the

program expression *se* by their JAVA CARD DL counterparts in order to obtain a proper logical term. For example, the JAVA division operator / has to be replaced by the function symbol *jdiv* (which is different from the standard mathematical division /, as explained in Chap. 12). The KeY system performs this conversion automatically to construct se^* from *se*. The complete list of predefined JAVA CARD DL operators is given in App. A.

If there is an atomic field access $v.a$ either on the left or on the right of the assignment, no further unfolding can be done and the possibility has to be considered here that the object reference may be null, which would result in a NullPointerException. Depending on whether the field access is on the left or on the right of the assignment one of the following rules applies:

assignment
$$\frac{v \,\dot{!=}\, null \Longrightarrow \{v_0 := v.a@Class\}\langle \pi \ \omega \rangle \phi \qquad v = null \Longrightarrow \langle \pi \text{ throw new NullPointerException(); } \omega \rangle \phi}{\Longrightarrow \langle \pi \ v_0 = v.a; \ \omega \rangle \phi}$$

assignment
$$\frac{v \,\dot{!=}\, null \Longrightarrow \{v.a@Class := se^*\}\langle \pi \ \omega \rangle \phi \qquad v = null \Longrightarrow \langle \pi \text{ throw new NullPointerException(); } \omega \rangle \phi}{\Longrightarrow \langle \pi \ v.a = se; \ \omega \rangle \phi}$$

A further complication is caused by field hiding. Hiding occurs when derived classes declare fields with the same name as in the superclass. The exact field reference has to be inferred from the static type of the target expression and the program context, in which the reference appears. Since logical terms do not have a program context, hidden fields have to be disambiguated. This can be achieved by an up-front renaming (as proposed in Def. 3.10), or (as done in the KeY system) with on-the-fly disambiguation in the assignment rule. It adds a qualifier @*Class* to the name of the field as the field migrates from the program into the update. The pretty-printer does not display the qualifier if there is no hiding.

For array access, we have to consider the possibility of an ArrayIndexOutOfBoundsException in addition to that of a NullPointerException. Thus, the rule for array access on the right of the assignment takes the following form (there is a slightly more complicated rule for array access on the left):

assignment
$$\frac{\begin{array}{l} v \,\dot{!=}\, null, se^* >= 0, se^* < v.length \Longrightarrow \{v_0 := v[se^*]\}\langle \pi \ \omega \rangle \phi \\ v = null \Longrightarrow \langle \pi \text{ throw new NullPointerException(); } \omega \rangle \phi \\ v \,\dot{!=}\, null, (se^* < 0 \mid se^* >= v.length) \Longrightarrow \\ \qquad \langle \pi \text{ throw new ArrayIndexOutOfBoundsException(); } \omega \rangle \phi \end{array}}{\Longrightarrow \langle \pi \ v_0 = v[se]; \ \omega \rangle \phi}$$

3.6.3 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

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

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:

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

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:

ifElse
$$\frac{\Longrightarrow \text{if}(se \doteq \text{TRUE}) \text{ then } \langle \pi \ p \ \omega\rangle\phi \text{ else } \langle \pi \ q \ \omega\rangle\phi}{\Longrightarrow \langle \pi \text{ if } (se) \ p \text{ else } q \ \omega\rangle\phi}$$

Note that the if-then-else in the premiss of the rule is a logical and not a program language construct ($\Rightarrow$ Def. 3.14).

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

$$(se \doteq \text{TRUE}) \longrightarrow \langle \pi \ p \ \omega\rangle\phi \quad \& \quad (se \mathrel{!\dot=} \text{TRUE}) \longrightarrow \langle \pi \ q \ \omega\rangle\phi$$

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.

3.6.4 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. And it is only practical if that number is small (as otherwise the proof gets too big).

If the number of loop iterations is not bound, the loop has to be verified using (a) induction ($\Rightarrow$ Chap. 11) or (b) an invariant rule ($\Rightarrow$ Sect. 3.7.1, 3.7.4). 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 standard case), the following simple version of the unwind rule can be applied:

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

Otherwise, in the general case where `break` and/or `continue` occur, the following more complex rule version has to be used:

$$\text{loopUnwind}$$
$$\frac{\Rightarrow \langle \pi \ \text{if} \ (e) \ l':\{ \ l'':\{ \ p' \ \} \ l_1:\ldots l_n:\text{while} \ (c) \ \{ \ p \ \} \ \} \ \omega \rangle \phi}{\Rightarrow \langle \pi \ l_1:\ldots l_n:\text{while} \ (e) \ \{ \ p \ \} \ \omega \rangle \phi}$$

where

- l' and l'' are new labels,
- p' is the result of (simultaneously) replacing in p
  - every "`break` l_i" (for $1 \le i \le n$) and every "`break`" (with no label) that has the `while` loop as its target by `break` l', and
  - every "`continue` l_i" (for $1 \le i \le n$) and every "`continue`" (with no label) that has the `while` loop as its target by `break` l''.

  (The target of a `break` or `continue` statement with no label is the loop that immediately encloses it.)

The label list $l_1:\ldots l_n:$ usually has only one element or is empty, but in general a loop can have more than one label.

In the "unwound" instance p' of the loop body p, the label l' is the new target for `break` statements and l'' is the new target for `continue` statements, which both had the `while` loop as target before. This results in the desired behaviour: `break` abruptly terminates the whole loop, while `continue` abruptly terminates the current instance of the loop body.

A `continue` (with or without label) is never handled directly by a Java Card DL rule, because it can only occur in loops, where it is always transformed into a `break` statement by the loop rules.

3.6.5 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. 3.8). To obtain an efficient calculus we have conservatively extended the programming language ($\Rightarrow$ Def. 3.12) 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.

Evaluation of Method Invocation Expressions

The process of evaluating a method invocation expression (method call) within our Java Card DL calculus consists of the following steps:

1. Identifying the appropriate method.
2. Computing the target reference.
3. Evaluating the arguments.
4. Locating the implementation (or throwing a `NullPointerException`).
5. Creating the method frame.
6. Handling the `return` statement.

Since method invocation expressions can take many different shapes, the calculus contains a number of slightly differing rules for every step. Also, not every step is necessary for every method invocation.

Step 1: Identify the Appropriate Method

The first step is to identify the appropriate method to invoke. This involves determining the right method signature and the class where the search for an implementation should begin. Usually, this process is performed by the compiler according to the (quite complicated) rules of the Java language specification and considering only static information such as type conformance and accessibility modifiers. These rules have to be considered as a background part of our logic, which we will not describe here though, but refer to the Java language specification instead. In the KeY system this process is performed internally (it does not require an application of a calculus rule), and

the implementation relies on the Recoder metaprogramming framework to achieve the desired effect (`recoder.sourceforge.net`).

For our purposes, we discern three different method invocation modes:

Instance or "virtual" mode. This is the most common mode. The target expression references an object (it may be an implicit `this` reference), and the method is not declared static or private. This invocation mode requires dynamic binding.

Static mode. In this case, no dynamic binding is required. The method to invoke is determined in accordance with the declared static type of the target expression and not the dynamic type of the object that this expression may point to. The static mode applies to all invocations of methods declared `static`. The target expression in this case can be either a class name or an object referencing expression (which is evaluated and then discarded). The static mode is also used for instance methods declared `private`.

Super mode. This mode is used to access the methods of the immediate superclass. The target expression in this case is the keyword `super`. The super mode bypasses any overriding declaration in the class that contains the method invocation.

Below, we present the rules for every step in a method invocation. We concentrate on the virtual invocation mode and discuss other modes only where significant differences occur.

Step 2: Computing the Target Reference

The following rule applies if the target expression of the method invocation is not a simple expression and may have side-effects. In this case, the method invocation gets unfolded so that the target expression can be evaluated first.

$$\mathsf{methodCallUnfoldTarget}$$
$$\frac{\Longrightarrow \langle \pi \ \ T_{nse} \ \ v_0 \ = \ nse \texttt{;} \ \ lhs \ = \ v_0 \texttt{.} mname \texttt{(} args \texttt{)} \texttt{;} \ \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ \ lhs \ = \ nse \texttt{.} mname \texttt{(} args \texttt{)} \texttt{;} \ \ \omega \rangle \phi}$$

This step is not performed if the target expression is the keyword `super` or a class name. For an invocation of a static method via a reference expression, this step *is* performed, but the result is discarded later on.

Step 3: Evaluating the Arguments

If a method invocation has arguments, they have to be completely evaluated before control is transferred to the method body. This is achieved by the following rule:

methodCallUnfoldArguments

$$\implies \langle \pi \ T_{e_{l_1}} \ v_1 = e_{l_1}\,;$$
$$\vdots$$
$$T_{e_{l_k}} \ v_k = e_{l_k}\,;$$
$$lhs \ = \ se.mname(a_1,\ldots,a_n)\,;$$
$$\underline{\omega\rangle\phi}$$
$$\implies \langle \pi \ lhs \ = \ se.mname(e_1,\ldots,e_n)\,; \ \omega\rangle\phi$$

The rule unfolds the arguments using fresh variables in the usual manner. However, only those argument expressions e_i get unfolded that are non-simple. We refer to the non-simple argument expressions as $e_{l_1} \ldots e_{l_k}$. The rule only applies if $k > 0$, i.e., there is at least one non-simple argument expression. The expressions a_i used in the premiss of the rule are then defined by:

$$a_i = \begin{cases} e_i & \text{if } e_i \text{ is a simple expression} \\ v_i & \text{if } e_i \text{ is a non-simple expression} \end{cases}$$

In the *instance* invocation mode, the target expression *se* must be simple (otherwise the rules from Step 2 apply). Furthermore, argument evaluation has to happen even if the target reference is `null`, which is not checked until the next step.

Step 4: Locating the Implementation

This step has two purposes in our calculus: to bind the argument values to the formal parameters and to simulate dynamic binding (for *instance* invocations). Both are achieved with the following rule:

methodCall

$$se \doteq \texttt{null} \implies \langle \pi \ \texttt{throw new NullPointerException();} \ \omega\rangle\phi$$
$$\underline{se \not\doteq \texttt{null} \implies \langle \pi \ T_{lhs} \ v_0\,; \ paramDecl\,; \ ifCascade\,; \ lhs = v_0\,; \ \omega\rangle\phi}$$
$$\implies \langle \pi \ lhs \ = \ se.mname(se_1,\ldots,se_n)\,; \ \omega\rangle\phi$$

The code piece *paramDecl* introduces and initialises new local variables that later replace the formal parameters of the method. That is, *paramDecl* abbreviates

$$T_{se_1} \ p_1 \ = \ se_1\,; \ \ldots \ T_{se_n} \ p_n \ = \ se_n\,;$$

The code schema *ifCascade* simulates dynamic binding. Using the signature of *mname*, we extract the set of classes that implement this particular method from the given JAVA program. Due to the possibility of method overriding, there can be more than one class implementing a particular method. At runtime, an implementation is picked based on the dynamic type of the target object—a process known as dynamic binding. In our calculus, we have to do a case distinction as the dynamic type is in general not known. We employ a

sequence of nested `if` statements that discriminate on the type of the target object and refer to the distinct method implementations via `method-body` statements ($\Rightarrow$ Def. 3.12). Altogether, *ifCascade* abbreviates:

```
if (se instanceof C₁)
    v₀=se.mname(p₁,...,pₙ)@C₁;
else if (se instanceof C₂)
    v₀=se.mname(p₁,...,pₙ)@C₂;
⋮
else if (se instanceof Cₖ₋₁)
    v₀=se.mname(p₁,...,pₙ)@Cₖ₋₁;
else v₀=se.mname(p₁,...,pₙ)@Cₖ;
```

The order of the if statements is a bottom-up latitudinal search over all classes $C_1, \ldots, C_k$ of the class inheritance tree that implement $mname(\ldots)$. In other words, the more specialised classes appear closer to the top of the cascade. Formally, if $i < j$ then $C_j \not\sqsubseteq C_i$.

If the invocation mode is *static* or *super* no *ifCascade* is created. The single appropriate method body statement takes its place. Furthermore, the check whether *se* is `null` is omitted in these modes, though not for private methods.

Step 5: Creating the Method Frame

In this step, the method-body statement v_0=`se.mname(...)`@*Class* is replaced by the implementation of *mname* from the class *Class* and the implementation is enclosed in a method frame:

$$
\frac{\begin{array}{l} \mathsf{methodBodyExpand} \\ \qquad\Longrightarrow \langle\pi \ \texttt{method-frame(result->}lhs\texttt{,} \\ \qquad\qquad\qquad\qquad\quad \texttt{source=}Class\texttt{,} \\ \qquad\qquad\qquad\qquad\quad \texttt{this=}se \\ \qquad\qquad\qquad\qquad\quad \texttt{) : \{ } body \texttt{ \} } \omega\rangle\phi \end{array}}{\langle\pi \ \ lhs\texttt{=}se\texttt{.}mname\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{)@}Class\texttt{; } \omega\rangle\phi \Longrightarrow}
$$

in the implementation *body* the formal parameters of *mname* are syntactically replaced by $v_1, \ldots, v_n$.

Step 6: Handling the `return` Statement

The final stage of handling a method invocation, after the method body has been symbolically executed, involves committing the return value (if any) and transferring control back to the caller. We postpone the description of treating method termination resulting from an exception (as well as the intricate interaction between a `return` statement and a `finally` block) until the following section on abrupt termination.

The basic rule for the `return` statement is:

methodCallReturn
$$\frac{\implies \langle \pi \ \texttt{method-frame(...):\{ } v\texttt{=}se\texttt{; \} } \omega \rangle \phi}{\implies \langle \pi \ \texttt{method-frame(result->}v\texttt{, ...) : \{ return } se\texttt{; } p \texttt{ \} } \omega \rangle \phi}$$

We assume that the return value has already undergone the usual unfolding analysis, and is now a simple expression se. Now, we need to assign it to the right variable v within the invoking code. This variable is specified in the head of the method frame. A corresponding assignment is created and v disappears from the method frame. Any trailing code p is also discarded.

After the assignment of the return value is symbolically executed, we are left with an empty method frame, which can now be removed altogether. This is achieved with the rule

methodCallEmpty
$$\frac{\implies \langle \pi \quad \omega \rangle \phi}{\implies \langle \pi \ \texttt{method-frame(...) : \{ \} } \omega \rangle \phi}$$

In case the method is void or if the invoking code simply does not assign the value of the method invocation to any variable, this fact is reflected by the variable v missing from the method frame. Then, slightly simpler versions of the return rule are used, which do not create an assignment.

Example for Handling a Method Invocation

Consider the example program from Fig. 3.3. The method `nextId()` returns for a given integer value `id` some next available value. In the `Base` class this method is implemented to return `id+1`. The class `SubA` inherits and retains this implementation. The class `SubB` overrides the method to return `id+2`, which is done by increasing the result of the implementation in `Base` by one.

We now show step by step how the following code, which invokes the method `nextId()` on an object of type `SubB`, is symbolically executed:

```Java
Base o = new SubB();
res = o.nextId(i);
```

First, the instance creation is handled, after which we are left with the actual method call. The effect of the instance creation is reflected in the updates attached to the formula, which we do not show here. Since the target reference o is already *simple* at this point, we skip Step 2. The same applies to the arguments of the method call and Step 3. We proceed with Step 4, applying the rule methodCall. This gives us two branches. One corresponds to the case where o is `null`, which can be discharged using the knowledge that o points to a freshly created object. The other branch assumes that o is not `null` and contains a formula with the following Java code (in the following, program part A is transformed into A′, B into B′ etc.):

```
public class Base {
  public int nextId(int i) {
    return ++i;
  }
}




public class SubA extends Base {
}




public class SubB extends Base {
  public int nextId(int i) {
    return super.nextId(i)+1;
  }
}
```
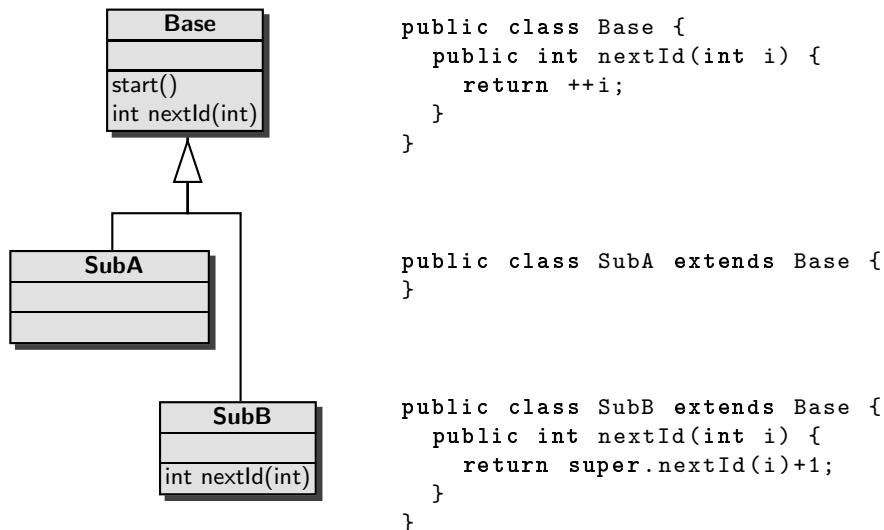
Fig. 3.3. An example program with method overriding

―― JAVA ――
```
int j; {
    int i_1 = i;
    if (o instanceof SubB)
        j=o.nextId(i_1)@SubB;        (A)
    else
        j=o.nextId(i_1)@Base;
}
 res=j;
```
―――――― JAVA ――

After dealing with the variable declarations, we reach the if-cascade simulating dynamic binding. In this case we happen to know the dynamic type of the object referenced by o. This eliminates the choice and leaves us with a method body statement pointing to the implementation from SubB:

―― JAVA ――
```
j=o.nextId(i_1)@SubB;        (A')
 res=j;
```
―――――― JAVA ――

Now it is time for Step 5, unfolding the method body statement and creating a method frame. This is achieved by the rule methodBodyExpand:

```
─── JAVA ────────────────────────────────────────
method-frame(result->j, source=SubB, this=o) : {
      return super.nextId(i_1)+1;   (B)                (A'')
}
 res=j;
```
─── JAVA ───

The method implementation has been inlined above. We start to execute it symbolically, unfolding the expression in the **return** statement in the usual manner, which gives us after some steps:

```
─── JAVA ────────────────────────────────────────
method-frame(result->j, source=SubB, this=o) : {
      int j_2 = super.nextId(i_1);   (C)
      j_1=j_2+1;                              (B')
      return   j_1;
}
res=j;
```
─── JAVA ───

The active statement is now again a method invocation, this time with the **super** keyword. The method invocation process starts again from scratch. Steps 2 and 3 can be omitted for the same reasons as above. Step 4 gives us the following code. Note that there is no if-cascade, since no dynamic binding needs to be performed.

```
─── JAVA ────────────────────────────────────────
method-frame(result->j, source=SubB, this=o) : {
      int j_3; {
          int i_2 = i_1;
          j_3=o.nextId(i_2)@Base;   (C')
      }
      j_2=j_3;
      j_1=j_2+1;
      return   j_1;
}
res=j;
```
─── JAVA ───

Now it is necessary to remove the declarations and perform the assignments to reach the method body statement **j_3=o.nextId(i_2)@Base;**. Then, this statement can be unpacked (Step 5), and we obtain two nested method frames. The second method frame retains the value of **this**, while the implementation source is now taken from the superclass:

─── JAVA ───

```
method-frame(result->j, source=SubB, this=o) : {
    method-frame(result->j_3, source=Base, this=o) : {
        return ++i_2;      (D)
    }
    j_2=j_3;                                                (C'')
    j_1=j_2+1;
    return j_1;
}
res=j;
```

─── JAVA ───

The return expression is unfolded until we arrive at a simple expression. The
actual return value is recorded in the updates attached to the formula. The
code in the formula then is:

─── JAVA ───

```
method-frame(result->j, source=SubB, this=o) : {
    method-frame(result->j_3, source=Base, this=o) : {
        return j_4;        (D')
    }                                                      (E)
    j_2=j_3;
    j_1=j_2+1;
    return j_1;
}
res=j;
```

─── JAVA ───

Now we can perform Step 6 (rule methodCallReturn), which replaces the
return statement of the inner method frame with the assignment to the
variable j_3. We know that j_3 is the receiver of the return value, since it
was identified as such by the method frame (this information is removed with
the rule application).

─── JAVA ───

```
method-frame(result->j, source=SubB, this=o) : {
    method-frame(source=Base, this=o) : {
        j_3=j_4;                           (E')
    }
    j_2=j_3;
    j_1=j_2+1;
    return j_1;
}
res=j;
```

─── JAVA ───

The assignment `j_3=j_4;` can be executed as usual, generating an update, and we obtain an empty method frame.

─── Java ───

```
method-frame(result->j, source=SubB, this=o) : {
    ┌─────────────────────────────────┐
    │ method-frame(source=Base, this=o) : │  ⓔ″
    │ }                               │
    └─────────────────────────────────┘
    j_2=j_3;
    j_1=j_2+1;
    return j_1;
}
res=j;
```

─── Java ───

The empty frame can be removed with the rule methodCallEmpty, completing Step 6. The invocation depth has now decreased again. We obtain the program:

─── Java ───

```
method-frame(result->j, source=SubB, this=o) : {
    j_2=j_3;
    j_1=j_2+1;
    return j_1;
}
res=j;
```

─── Java ───

From here, the execution continues in an analogous manner. The outer method frame is eventually removed as well.

3.6.6 Instance Creation and Initialisation

In this section we cover the process of instance creation and initialisation. We do not go into details of array creation and initialisation, since it is sufficiently similar.

Instance Creation and the Constant Domain Assumption

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 introduce *object repositories* with access functions and *implicit fields* that allow to change and query the program-visible instance state (created, initialised, etc.). These implicit fields behave as the usual class or instance attributes, except that they are not declared by the user but by the logic designer. To distinguish them from

normal (user declared) attributes, their names are enclosed in angled brackets. According to their use we distinguish object state and repository fields. An overview of the used implicit fields is given in Table 3.4.

Definition 3.52. *Given a non-abstract class type C, the object repository Rep_C is the set of all domain elements e of dynamic type C:*

$$Rep_C := \{e \in \mathcal{D}_0 \mid \delta(e) = C\}$$

Note, that Rep_C does not contain the objects of type D even if D is a subtype of C.

Allocating a new object requires to access the corresponding object repository. Therefore, we introduce access functions for the object repositories.

Definition 3.53. *For each non-abstract class type C there is a predefined rigid function symbol*

$$C\text{::}get : \texttt{integer} \rightarrow C$$

called repository access function.

Restricted to the set of non-negative integers, C::get is interpreted as a bijective mapping onto the object repository Rep_C of type C. For negative integers, C::get is also defined, but its values are unknown.

Given a JAVA CARD DL Kripke structure, the index *of an object o is the non-negative integer i for which the equation $\mathcal{I}(C\text{::}get)(i) = o$ holds.*

Example 3.54. Since the dynamic type function $\delta(\cdot)$ is only defined for a model, it cannot be used within the logic. We must take another way to express with a formula that a term ("expression") evaluates ("refers") to a domain element ("object") of a given dynamic type. The repository access functions allow us to do it concisely. For example the formula

$$\exists i : \texttt{integer}.(o \doteq C\text{::}get(i))$$

holds iff the term o evaluates to a domain element of dynamic type C (excluding, among other, elements of any type D, which might be a subtype of C).

To model instance allocation appropriately, we must ensure that the new object is not already in use. Therefore, we declare an implicit static integer field `<nextToCreate>` for each non-abstract class type C.

We call an object *created*, if its index is greater or equal to zero and less than the value of `<nextToCreate>`. When an instance of dynamic type T is allocated by a JAVA program, the instance with T.`<nextToCreate>` as object index is used and `<nextToCreate>` is incremented by one. In all states that are reachable by a JAVA program ($\Rightarrow$ Sect. 3.3.5), the value of `<nextToCreate>` is non-negative.

Further, there is the implicit boolean instance field `<created>` declared in `java.lang.Object`, which is supported mainly for convenience. This field is set for an object during the instance creation process.

Table 3.4. Implicit object repository and status fields

Modifier	Implicit field	Declared in	Explanation
private static	**int** `<nextToCreate>`	T	the index of the object to be taken the next time when a **new** $T(\ldots)$ expression is evaluated
protected boolean `<created>`		`Object`	indicates whether the object has been created
protected boolean `<initialised>`		`Object`	indicates whether the object has been initialised

Example 3.55. Consider an instance invariant of class `A` (a property that must hold for every object of class `A` or any class derived from `A`, in each observable state) that states that the field `head` declared in `A` must always be non-null.

With `<created>` this can be formalised concisely as:

$$\forall a : A.(a.\texttt{<created>} \doteq \text{TRUE} \rightarrow (a.\texttt{head} \mathrel{!\dot{=}} \texttt{null}))$$

Using `<nextToCreate>` and the repository access functions, in contrast, would yield a complicated formula, and even require enumerating all subtypes of `A` in it.

The Java Instance Initialisation Process

We use an approach to handle instance creation and initialisation that is based on program transformation. The transformation reduces a Java program p to a program p' such that the behaviour of p (with initialisation) is the same as that of p' when initialisation is disregarded. This is done by inserting code into p that explicitly executes the initialisation.

The transformation inserts code for explicitly executing all initialisation processes. To a large extent, the inserted code works by invoking implicit class or instance methods (similar to implicit fields), which do the actual work. An overview of all implicit methods introduced is given in Table 3.5.

The transformation covers all details of initialisation in Java, except that we only consider non-concurrent programs and no reflection facilities (in particular no instances of `java.lang.Class`). Initialisation of classes and interfaces (also known as static initialisation) is fully supported for the single threaded case. KeY passes the static initialisation challenge stated by Jacobs et al. [2003]. We omit the treatment of this topic here for space reasons though.

In the following we use the schematic class form that is stated in Figure 3.4.

Table 3.5. Implicit methods for object creations and initialisation declared in every non-abstract type T (syntactic conventions from Figure 3.4)

Static methods	
`public static <createObject>()`	T main method for instance creation and initialisation
`private static` T `<allocate>()`	allocation of an unused object from the object repository

Instance methods	
`protected void <prepare>()`	assignment of default values to all instance fields
mods T `<init>`(*params*)	execution of instance initialisers and the invoked constructor

```
mods₀ class  T {
   mods₁  T₁  a₁  =  initExpression₁ ;

   ⋮

   modsₘ  Tₘ  aₘ  =  initExpressionₘ ;

   {
      initStatementₘ₊₁ ;

      ⋮

      initStatementₗ ;
   }

   mods  T(params) {
      st₁ ;

      ⋮

      stₙ ;
   }
   ...
}
```

Fig. 3.4. Initialisation part in a schematic class

Example 3.56. Figure 3.5 shows a class `Person` and its mapping to the schematic class declaration of Figure 3.4. There is only one initialiser statement in class `Person`, namely "`id = 0`", which is induced by the corresponding field declaration of `id`.

```
class Person {
  private int id = 0;

  public Person(int persID) {
    id = persID;
  }
}
```

$$
\begin{aligned}
mods_0 &\mapsto - \\
T &\mapsto \texttt{Person} \\
mods_1 &\mapsto \texttt{private} \\
T_1 &\mapsto \texttt{int} \\
a_1 &\mapsto \texttt{id} \\
initExpression_1 &\mapsto \texttt{0} \\
mods &\mapsto \texttt{public} \\
params &\mapsto \texttt{int persID} \\
st_1 &\mapsto \texttt{id = persID}
\end{aligned}
$$

Fig. 3.5. Example for the mapping of a class declaration to the schema of Fig. 3.4

To achieve a uniform presentation we also stipulate that:

1. The default constructor `public` T`()` exists in T in case no explicit constructor has been declared.
2. Unless $T = \texttt{Object}$, the statement st_1 must be a constructor invocation. If this is not the case in the original program, "`super();`" is added explicitly as the first statement.

Both of these conditions reflect the actual semantics of JAVA.

The Rule for Instance Creation and Initialisation

The instance creation rule

$$
\text{instanceCreation} \quad \frac{\begin{aligned} \Longrightarrow \langle \pi \;\; T \;\; v_0 \;\; \texttt{=} \;\; T.\texttt{<createObject>();} \\ v_0.\texttt{<init>}(args)\texttt{;} \\ v_0.\texttt{<initialised>} \;\texttt{=}\; \texttt{true;} \\ v \;\texttt{=}\; v_0\texttt{;} \\ \omega \rangle \phi \end{aligned}}{\Longrightarrow \langle \pi \;\; v \;\texttt{=}\; \texttt{new} \;\; T(args)\texttt{;} \;\; \omega \rangle \phi}
$$

replaces an instance creation expression "$v = \texttt{new}\ T(args)$" by a sequence of statements. These can be divided into three phases, which we examine in detail below:

1. obtain the next available object from the repository (as explained above, it is not really "created") and assign it to a fresh temporary variable v_0
2. prepare the object by assigning all fields their default values
3. initialise the object of v_0 and subsequently mark it as initialised. Finally, assign v_0 to v

The reason for assigning to v in the last step is to ensure correct behaviour in case initialisation terminates abruptly due to an exception.[5]

Phase 1: Instance Creation

The implicit static method `<createObject>()` ($\Rightarrow$ Fig. 3.6) declared in each non-abstract class T returns the next available object from the object repository of type T after setting its fields to default values.

```
public static  T <createObject>() {
        // Get an unused instance from the object repository
    T newObject = T.<allocate>();
    newObject.<transient> = 0;
    newObject.<initialized> = false;
        // Invoke the preparation method to assign default values to
        // instance fields
    newObject.<prepare>();
        // Return the newly created object in order to initialise it:
    return newObject;
}
```

Fig. 3.6. Implicit method `<createObject>()`

`<createObject>()` delegates the actual interaction with the object repository to yet another helper, an implicit method called `<allocate>()`. The `<allocate>()` method has no JAVA implementation, its semantics is given by the following rule instead:

allocateInstance

$$\frac{\Rightarrow \{lhs := T :: get(T.\texttt{<nextToCreate>})\} \\ \quad \{lhs.\texttt{<created>} := true\} \\ \qquad \{T.\texttt{<nextToCreate>} := T.\texttt{<nextToCreate>} + 1\}\langle \pi\ \omega\rangle\phi}{\Rightarrow \langle \pi\ lhs\ =\ T.\texttt{<allocate>}();\ \omega\rangle\phi}$$

The rule ensures that after termination of `<allocate>()`:

- The object that has index `<nextToCreate>` (in the pre-state) is allocated and returned.
- Its `<created>` field has been set to true.
- The field `<nextToCreate>` has been increased by one.

---

[5] Nonetheless, JAVA does not prevent creating and accessing partly initialised objects. This can be done, for example, by assigning the object reference to a static field during initialisation. This behaviour is modelled faithfully in the calculus. In such cases the preparation phase guarantees that all fields have a definite value.

Note that the mathematical arithmetic addition is used to specify the increment of field `<nextToCreate>`. This is the reason for using a calculus rule to define `<allocate>()` instead of Java code. An unbounded number of objects could not be modelled with bounded integer data types of Java.

Phase 2: Preparation

The next phase during the execution of `<createObject>()` is the preparation phase. All fields, including the ones declared in the superclasses, are assigned their default values.[6] Up to this point no user code is involved, which ensures that all field accesses by the user observe a definite value. This value is given by the function *defaultValue* that maps each type to its default value (e.g., `int` to 0). The concrete default values are specified in the Java language specification [Gosling et al., 2000, § 4.5.5]. The method `<prepare>()` used for preparation is shown in Figure 3.7.

```
protected void <prepare>() {
        // Prepare the fields declared in the superclass...
    super.<prepare>();                 // unless T = Object
        // Then assign each field a_i of type T_i declared in T
        // to its default value:
    a_1 =  defaultValue(T_1);
    ...
    a_m =  defaultValue(T_m);
}
```

Fig. 3.7. Implicit method `<prepare>()`

Note 3.57. In the KeY system, `<createObject>()` does not call `<prepare>()` on the new object directly. Instead it invokes another implicitly declared method called `<prepareEnter>()`, which has private access and whose body is identical to the one of `<prepare>()`. The reason is that due to the `super` call in `<prepare>()`'s body, its visibility must be at least `protected` such that a direct call would trigger dynamic method dispatching, which is unnecessary and would lead to a larger proof.

Phase 3: Initialisation

After the preparation of the new object, the user-defined initialisation code can be processed. Such code can occur

---

[6] Since class declarations are given beforehand this is possible with a simple enumeration. In case of arrays, a quantified update is used to achieve the same effect, even when the actual array size is not known.

- as a field initialiser expression "T $attr$ = val;" (e.g., (*) in Figure 3.8); the corresponding initialiser statement is $attr$ = val;
- as an instance initialiser block (similar to (**) in Figure 3.8); such a block is also an initialiser statement;
- within a constructor body (like (***) in Figure 3.8).

```
class A {
  (*)     private int a = 3;          private <init>() {
  (**)    {a++;}                          super.<init>();
          public int b;                   a = 3;
                                          {a++;}
  (***)   private A() {                   a = a + 2;
             a = a + 2;                }
          }
  (***)   public A(int i) {           public <init>(int i) {
             this();                     this.<init>();
             a = a + i;                  a = a + i;
          }                           }
  ...                               }
```

Fig. 3.8. Example for constructor normal forms

For each constructor $mods$ $T(params)$ of T we provide a constructor normal form $mods$ T <init>($params$), which includes (1) the initialisation of the superclass, (2) the execution of all initialiser statements in source code order, and finally (3) the actual constructor body. In the initialisation phase the arguments of the instance creation expression are evaluated and passed on to this constructor normal form. An example of the normal form is given in Figure 3.8.

The exact blueprint for building a constructor normal form is shown in Figure 3.9, using the conventions of Figure 3.4. Due to the uniform class form assumed above, the first statement st_1 of every original constructor is either an alternate constructor invocation or a superclass constructor invocation (with the notable exception of $T = $ Object). Depending on this first statement, the normal form of the constructor is built to do one of two things:

1. $st_1 = $ super($args$): Recursive re-start of the initialisation phase for the superclass of T. If $T = $ Object stop. Afterwards, initialiser statements are executed in source code order. Finally, the original constructor body is executed.
2. $st_1 = $ this($args$): Recursive re-start of the initialisation phase with the alternate constructor. Afterwards, the original constructor body is executed.

If one of the above steps fails, the initialisation terminates abruptly throwing an exception.

```
mods  T  <init>(params) {            mods  T  <init>(params) {
        // invoke constructor
        // normal form of superclass         // constructor normal form
        // (only if T ≠ Object)               // instead of this(args)
    super.<init>(args);               this.<init>(args);
                                             // no initialiser statements
        // add the initialiser               // if st₁ is an explicit
        // statements:                       // this() invocation
    initStatement₁;

    ...

    initStatementₗ;
        // append constructor body           // append constructor body
    stₛ; ... stₙ;                     st₂; ... stₙ;
        // if T = Object then s = 1           // starting with its second
        // otherwise s = 2                    // statement
}                                    }
```

$$(a) \ st_1 = \texttt{super}(args) \qquad\qquad (b) \ st_1 = \texttt{this}(args)$$

(a) $st_1 = \texttt{super}(args)$
    in the original
    constructor

(b) $st_1 = \texttt{this}(args)$
    in the original
    constructor

Fig. 3.9. Building the constructor normal form

3.6.7 Handling Abrupt Termination

Abrupt Termination in Java Card DL

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 the 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 an exception is caught), or the whole program terminates abruptly (if an exception is not caught).

Evaluation of Arguments

If the argument of a throw or a return statement is a non-simple expression, the statement has to be unfolded first such that the argument can be (symbolically) evaluated:

$$\text{throwEvaluate} \quad \frac{\Longrightarrow \langle \pi \; T_{nse} \; v_0 \; = \; nse; \; \texttt{throw} \; v_0; \; \omega \rangle \phi}{\Longrightarrow \langle \pi \; \texttt{throw} \; nse; \; \omega \rangle \phi}$$

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. 3.18). For that case rules such as the following are provided in the JAVA CARD DL calculus for all abruptly terminating statements:

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

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 a modal operator but only changes the control flow is more difficult to handle and requires more rules. The basic idea for handling this case in our JAVA CARD DL calculus are rules that *symbolically* execute the change in control flow 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`). Below, rules for all combinations are discussed—with the following exceptions:

- The rule for the combination method frame/`return` is part of handling method invocations (Step 6 in Sect. 3.6.5).
- Due to restrictions of the JAVA language specification, the combination method frame/`break` does not occur.
- Since the `continue` statement can only occur within loops, all occurrences of `continue` are handled by the loop rules (Sect. 3.7).

Moreover, `switch` statements, which may contain a `break`, are not considered here; they are transformed into a sequence of `if` statements.

Rule for Method Frame and `throw`

In this case, the method is terminated, but no return value is assigned. The
`throw` statement remains unchanged (i.e., the exception is handed up to the
invoking code):

$$\mathsf{methodCallThrow}\ \dfrac{\Longrightarrow \langle \pi\ \mathtt{throw}\ se\mathtt{;}\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \mathtt{method\text{-}frame}(\ldots)\ \mathtt{:}\ \mathtt{\{throw}\ se\mathtt{;}\ p\ \mathtt{\}}\ \omega\rangle\phi}$$

Rules for `try` and `throw`

The rule in Figure 3.10 allows to handle `try-catch-finally` blocks and the
`throw` statement. The schema variable *cs* represents a (possibly empty) se-
quence of catch clauses. The rule covers three cases corresponding to the
three cases in the premiss:

1. The argument of the `throw` statement is the null pointer (which, of
   course, in practice should not happen). In that case everything remains
   unchanged except that a `NullPointerException` is thrown instead of
   null.
2. The first catch clause catches the exception. Then, after binding the ex-
   ception to v, the code p from the catch clause is executed.
3. The first catch clause does *not* catch the exception. In that case the first
   clause gets eliminated. The same rule can then be applied again to check
   further clauses.

Note, that in all three cases the code p after the `throw` statement gets elim-
inated.

$$\mathsf{tryCatchThrow}\ \dfrac{\begin{array}{l}\Longrightarrow \langle \pi\ \mathtt{if\ (}se\ \mathtt{==\ null)\ \{}\\ \qquad \mathtt{try\ \{\ throw\ NullPointerException\ ();\ \}}\\ \qquad \mathtt{catch\ (}T\ v\mathtt{)\ \{}\ q\ \mathtt{\}}\ cs\ \mathtt{finally\ \{}\ r\ \mathtt{\}}\\ \mathtt{\}\ else\ if\ (}se\ \mathtt{instanceof}\ T\mathtt{)\ \{}\\ \qquad \mathtt{try\ \{}\ T\ v\mathtt{;}\ v\ \mathtt{=}\ se\mathtt{;}\ q\ \mathtt{\}\ finally\ \{}\ r\ \mathtt{\}}\\ \mathtt{\}\ else\ \{}\\ \qquad \mathtt{try\ \{\ throw}\ se\mathtt{;\ \}}\ cs\ \mathtt{finally\ \{}\ r\ \mathtt{\}}\\ \mathtt{\}}\\ \omega\rangle\phi\end{array}}{\begin{array}{l}\Longrightarrow \langle \pi\ \mathtt{try\ \{\ throw}\ se\mathtt{;}\ p\mathtt{\}}\\ \qquad \mathtt{catch\ (}\ T\ v\ \mathtt{)\ \{}\ q\ \mathtt{\}}\ cs\ \mathtt{finally\ \{}\ r\ \mathtt{\}}\\ \omega\rangle\phi\end{array}}$$

Fig. 3.10. The rule for `try-catch-finally` and `throw`

When all catch clauses have been checked and the exception has still not
been caught, the following rule applies:

$$\text{tryFinallyThrow} \quad \frac{\Longrightarrow \langle \pi \ T_{se} \ v_{se} \ \texttt{=} \ se \texttt{;} \ r \ \texttt{throw} \ v_{se} \texttt{;} \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ \texttt{try \{ throw} \ se \texttt{; } p \ \texttt{\} finally \{ } r \ \texttt{\}} \rangle \phi}$$

This rule moves the code r from the finally block to the front. The `try`-block gets eliminated so that the thrown exception now may be caught by other `try` blocks in π (or remain uncaught). The value of se has to be saved in v_{se} before the code r is executed as r might change se.

There is also a rule for `try` blocks that have been symbolically executed without throwing an exception and that are now empty and terminate normally (similar rules exist for empty blocks and empty method frames). Again, cs represents a finite (possibly empty) sequence of catch clauses:

$$\text{tryEmpty} \quad \frac{\Longrightarrow \langle \pi \ r \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ \texttt{try\{ \}} \ cs \ \texttt{\{ } q \ \texttt{\} finally \{ } r \ \texttt{\}} \ \omega \rangle \phi}$$

Rules for `try/break` and `try/return`

A `return` or a `break` statement within a `try-catch-finally` statement causes the immediate execution of the `finally` block. Afterwards the `try` statement terminates abnormally with the `break` resp. the `return` statement (a different abruptly terminating statement that may occur in the `finally` block takes precedence). This behaviour is simulated by the following two rules (here, also, cs is a finite, possibly empty sequence of catch clauses):

tryBreak
$$\frac{\Longrightarrow \langle \pi \ r \ \texttt{break} \ l \texttt{;} \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ \texttt{try\{ break} \ l \texttt{; } p \ \texttt{\}} \ cs \ \texttt{\{ } q \ \texttt{\} finally\{ } r \ \texttt{\}} \ \omega \rangle \phi}$$

tryReturn
$$\frac{\Longrightarrow \langle \pi \ T_{v_r} \ v_0 \ \texttt{=} \ v_r \texttt{;} \ r \ \texttt{return} \ v_0 \texttt{;} \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ \texttt{try\{ return} \ v_r \texttt{; } p \ \texttt{\}} \ cs \ \texttt{\{ } q \ \texttt{\} finally\{ } r \ \texttt{\}} \ \omega \rangle \phi}$$

Rules for `block/break`, `block/return`, and `block/throw`

The following two rules apply to blocks being terminated by a `break` statement that does not have a label resp. by a break statement with a label l identical to one of the labels $l_1, \ldots, l_k$ of the block ($k \geq 1$).

$$\text{blockBreakNoLabel} \quad \frac{\Longrightarrow \langle \pi \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ l_1 \texttt{:} \ldots l_k \texttt{:\{ break; } p \ \texttt{\}} \ \omega \rangle \phi}$$

$$\text{blockBreakLabel} \quad \frac{\Longrightarrow \langle \pi \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ l_1 \texttt{:} \ldots l_i \texttt{:} \ldots l_k \texttt{:\{ break} \ l_i \texttt{; } p \ \texttt{\}} \ \omega \rangle \phi}$$

To blocks (labelled or unlabelled) that are abruptly terminated by a `break` statement with a label l not matching any of the labels of the block, the following rule applies:

$$\text{blockBreakNomatch} \quad \frac{\Longrightarrow \langle \pi \ \texttt{break} \ l; \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ l_1 : \ldots l_k : \{ \ \texttt{break} \ l; \ p \} \ \omega \rangle \phi}$$

Similar rules exist for blocks that are terminated by a `return` or `throw` statement:

$$\text{blockReturn} \quad \frac{\Longrightarrow \langle \pi \ \texttt{return} \ v; \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ l_1 : \ldots l_k : \{ \ \texttt{return} \ v; \ p \} \ \omega \rangle \phi}$$

$$\text{blockThrow} \quad \frac{\Longrightarrow \langle \pi \ \texttt{throw} \ v; \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ l_1 : \ldots l_k : \{ \ \texttt{throw} \ v; \ p \} \ \omega \rangle \phi}$$

3.7 Calculus Component 3: Invariant Rules for Loops

There are two techniques for handling loops in KeY: induction and using an invariant rule. In the following we describe the use of invariant rules. A separate chapter is dedicated to handling loops by induction (Chapter 11).

3.7.1 The Classical Invariant Rule

Before we discuss the problems that arise when setting up an invariant rule for a complex language like JAVA CARD, we first recall the classical invariant rule for a simple deterministic while-language with assignments, if-then-else, and while-loops. In particular, we assume that there is no abrupt termination and expressions do not have side-effects.

For such a simple while-language the invariant rule looks as follows:

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

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 $! \, se$,

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.

It is crucial to the soundness of Rule invRuleClassical that expressions are free of side-effects and that there is no concept of abrupt termination like, for example, in JAVA CARD. In the following we discuss the problems arising from side-effects of expressions and abrupt termination concerning the invariant rule.

3.7.2 Loop Invariants and Abrupt Termination in JAVA CARD DL

JAVA CARD DL does not distinguish non-termination and abrupt termination. For example, the formulae

$$[\text{while (true) ;}]\phi$$

and

$$[\text{i = i / (j - j);}]\phi$$

are equivalent (both evaluate to true). However, the program (fragment) in the first formula does not terminate while the program in the second formula terminates abruptly with an `ArithmeticException` (due to division by zero).

Thus, setting up a sound invariant rule for JAVA CARD DL requires a more fine-grained semantics concerning termination behaviour of programs. There are (at least) the following two approaches to distinguish between non-termination and abrupt termination.

Firstly, the logic JAVA CARD DL could be enriched with additional labelled modalities $[\ ]_R$ and $\langle\ \rangle_R$ with $R \subseteq \{break, exception, continue, return\}$ referring to the reason R of a possible abrupt termination. The semantics of a formula $[p]_R\phi$ is that, if the program p terminates abruptly with reason R, then the formula ϕ has to hold in the final state, whereas $\langle p \rangle_R\phi$ expresses that p terminates abruptly with reason R and in the final state ϕ holds.

The second possibility for distinguishing non-termination and abrupt termination is to perform a program transformation such that the resulting program catches all top-level exceptions and thus always terminates normally. Abrupt termination due to exceptions can, e.g., be handled by enclosing the original program with a `try-catch` block. For example, the following (valid) formula expresses that if the program from above terminates abruptly with an exception then formula ϕ has to hold:

```
[Throwable thrown = null;
try {
    i = i / (j - j);
} catch (Exception e) {
    thrown = e;
}
](thrown != null -> φ)
```

Using the additional modalities the same could be expressed more concisely as

$$[\text{i = i / (j - j);}]_{exception}\phi \ .$$

Handling the other reasons for abrupt termination by program transformation is more involved and is not explained here. The advantage of using dedicated modalities is that termination properties can be addressed on a syntactic level (suited for reasoning with a calculus), whereas the program transformation approach relies on the semantics of JAVA CARD to encode abrupt termination and its reason. In order to describe the invariant rule for JAVA CARD DL in the following (and also the improved rule in Section 3.7.4), we pursue the approach of introducing additional modalities. Note however, that the actual rule available in the KeY system is based on the program transformation approach. The reason for that is that introducing indexed modalities would result in a multitude of rules to be added to the calculus.

Since the general rule covering all reasons for abrupt termination and side-effects of the loop condition is very complex, we start with some simpler rules dealing with special cases excluding certain difficulties (e.g., abrupt termination).

Normal Termination and Condition Without Side-Effects

Under the assumptions that (a) the loop does not terminate abruptly (i.e., terminates normally or does not terminate at all) and that (b) the loop condition does not have side-effects, the invariant rule essentially corresponds to the classical rule (the only difference are the prefix π and the postfix ω, which are not present in the classical rule. As a reminder: π consists of opening braces, labels, and try-statement but no executable statements and ω contains the rest of the program (including closing braces and catch-blocks).

$$\text{invRuleSimple} \ \ \frac{\begin{array}{c} \Gamma \Longrightarrow \mathcal{U} Inv, \ \Delta \\ Inv \ \& \ se \Longrightarrow [p]Inv \\ Inv \ \& \ !\, se \Longrightarrow [\pi \ \omega]\phi \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi \ \text{while } (se) \ \{ \ p \ \} \ \omega]\phi, \ \Delta} \ \ (*)$$

Abrupt Termination and Condition Without Side-Effects

When a `continue` statement without label or with a label referring to the currently investigated loop is encountered in the loop body, the execution of the body is stopped and the loop condition is evaluated again, i.e., the loop moves on to the next iteration. Thus, the invariant Inv has to hold both when the body terminates normally and when a continue statement occurs (second premiss of rule invRuleAt).

If during the execution of the loop body

- a `continue` statement with a label referring to an enclosing loop,
- an exception is thrown that is not caught within the body,
- a `break` statement occurs without label or with a label that refers to a position in front of the loop, or
- a `return` statement occurs

then the whole loop statement terminates abruptly. In the rule, the reasons leading to abrupt termination of the whole loop statement are contained in the set $AT = \{break, exception, return\}$. Note that a `continue` with a label referring to an enclosing loop can be simulated by a corresponding `break` statement and, thus, we also use the label *break* to identify this case. In contrast, we use the label *continue* if the control flow is to be transferred to the beginning of the loop containing the `continue` statement.

The consequence of abrupt termination of a loop statement is that the execution of the loop is stopped immediately and the control flow is changed according to the reason for the abrupt termination. Thus, since the whole loop statement terminates, it is then not necessary to show that the invariant holds. Rather it must be shown that the postcondition ϕ holds after the rest of the program following the while loop has been executed (provided it terminates). This is expressed by the third premiss of rule invRuleAt, where the formula $\langle p \rangle_{AT}$ true holds iff p terminates abruptly (*se* is a simple expression and its evaluation to terminate normally). If no abrupt termination occurs in the loop body, rule invRuleAt reduces to rule invRuleSimple).

$$
\text{invRuleAt} \quad \frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}\,Inv,\ \Delta \\ Inv\,\&\,se \Longrightarrow [p]Inv\,\&\,[p]_{continue}Inv \\ Inv\,\&\,se \Longrightarrow \langle p \rangle_{AT}\ \text{true} \rightarrow [\pi\ \ p\ \ \omega]\phi \\ Inv\,\&\,!\,se \Longrightarrow [\pi\ \omega]\phi \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \texttt{while (}se\texttt{) \{ }p\texttt{ \} }\omega]\phi,\ \Delta}\ (*)
$$

Normal Termination and Condition with Side-Effects

Now we assume that the loop body terminates normally but the loop condition may have side-effects (including abrupt termination which leads to abrupt termination of the loop statement). A loop condition *nse* that may have side-effects is not a logical term and therefore has to be evaluated first. The result is then assigned to a new variable v of type `boolean`. The only reason for abrupt termination of a while statement during evaluation of the loop condition can be an exception. Thus, $AT = \{exception\}$.

The first premiss is identical to the one in the previous rules. The second and third premiss correspond to premisses two and three of rule invRuleSimple, but take possible side-effects (except for abrupt termination) of evaluating the loop condition e into account.

Abrupt termination of the loop condition caused by an exception is handled in premiss four, where the postcondition ϕ has to be established since

the whole loop statement terminates abruptly. If the evaluation of the loop condition does not throw an exception this premiss trivially holds since then $\langle \texttt{v=}e\texttt{;}\rangle_{AT}$ true evaluates to false.

In case that the evaluation of *nse* does not terminate at all, all premisses except for the first one are trivially valid. Note that this would not be true if modality $[\cdot]_{AT}$ had been used in the fourth premiss instead of $\langle\cdot\rangle_{AT}$.

$$
\text{invRuleNse}\ \frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}\mathit{Inv},\ \Delta \\ \mathit{Inv} \Longrightarrow [\texttt{boolean}\ v\texttt{=}\mathit{nse}\texttt{;}](v \doteq \text{TRUE} \mathrel{-\!\!>} [p]\mathit{Inv}) \\ \mathit{Inv} \Longrightarrow [\texttt{boolean}\ v\texttt{=}\mathit{nse}\texttt{;}](v \doteq \text{FALSE} \mathrel{-\!\!>} [\pi\ \omega]\phi) \\ \mathit{Inv},\ \langle\texttt{boolean}\ v\texttt{=}\mathit{nse}\texttt{;}\rangle_{AT}\,\text{true} \Longrightarrow [\pi\ \texttt{boolean}\ v\texttt{=}\mathit{nse};\ \omega]\phi \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \texttt{while}\ (\mathit{nse})\ \{\ p\ \}\ \omega]\phi,\ \Delta}\ (*)
$$

Abrupt Termination and Condition with Side-Effects

The following most general rule covers all possible cases of side-effects and abrupt termination.

Again, the sets $AT = \{\mathit{break}, \mathit{exception}, \mathit{return}\}$ and $AT' = \{\mathit{exception}\}$ contain the reasons leading to abrupt termination of the whole loop statement.

invRule
$$
\frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}\mathit{Inv},\ \Delta \\ \mathit{Inv} \Longrightarrow [\texttt{boolean}\ v\texttt{=}\mathit{nse}\texttt{;}](v \doteq \text{TRUE} \mathrel{-\!\!>} ([p]\mathit{Inv}\ \&\ [p]_{continue}\mathit{Inv})) \\ \mathit{Inv},\ \langle\texttt{boolean}\ v\texttt{=}\mathit{nse}\texttt{;}\rangle_{AT'}\,\text{true} \Longrightarrow [\pi\ \texttt{boolean}\ v\texttt{=}\mathit{nse};\ \omega]\phi \\ \mathit{Inv},\ \langle\texttt{boolean}\ v\texttt{=}\mathit{nse}\texttt{;}\rangle(v \doteq \text{TRUE}\ \&\ \langle p\rangle_{AT}\,\text{true}) \Longrightarrow \\ \hspace{4cm} [\pi\ \texttt{boolean}\ v\texttt{=}\mathit{nse};p\ \omega]\phi \\ \mathit{Inv} \Longrightarrow [\texttt{boolean}\ v\texttt{=}\mathit{nse}\texttt{;}](v \doteq \text{FALSE} \mathrel{-\!\!>} [\pi\ \omega]\phi) \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \texttt{while}\ (\mathit{nse})\ \{\ p\ \}\ \omega]\phi,\ \Delta}\ (*)
$$

The first premiss is identical to the one in previous rules and states that the invariant has to hold in the beginning.

The second premiss covers the case that executing the loop body once preserves the invariant both if the execution terminates normally and if a **continue** statement occurred. Note that it is important that the execution of p is started in a state reflecting the side-effects of evaluating the loop condition *nse*. Therefore, writing

$$
\mathit{Inv}, [\texttt{boolean}\ v\texttt{=}\mathit{nse}\texttt{;}](v \doteq \text{TRUE}) \Longrightarrow [p]\mathit{Inv}\ \&\ [p]_{continue}\mathit{Inv}
$$

instead would not be correct.

Premiss three (which is the same as premiss four of rule invRuleNse) states that if the invariant holds and the evaluation of the loop condition *nse* terminates abruptly (the only reason can be an exception), then the postcondition ϕ has to hold after the rest ω of the program has been executed. Since

the whole loop terminates abruptly if the loop condition terminates abruptly, the loop is omitted in the formula on the right side of the sequent.

Also in the fourth premiss, the postcondition ϕ has to be established if the invariant *Inv* holds and the evaluation of the loop condition *nse* terminates normally but the execution of p terminates abruptly due to one of the reasons in *AT*. Again, in the formula on the right side of the sequent the loop statement is replaced by a statement evaluating the loop condition and the body of the loop.

The last premiss applies to the case that the loop terminates because the loop condition evaluates to false. Then, assuming the invariant *Inv* holds before executing the rest ω of the program, the postcondition ϕ has to hold.

3.7.3 Implementation of Invariant Rules

The invariant rules, from invRuleSimple to invRule, are different from other rules of the JAVA CARD DL calculus, since in some premisses the context (denoted by Γ, Δ and the update $\mathcal{U}$) is deleted, which is why rule schemata with ($*$) are used ($\Rightarrow$ Def. 3.49). If the context would not be deleted, the rules would not be sound as the following example shows.

Example 3.58. Consider the following JAVA CARD program:

```
────── JAVA ──────
while ( i<10 ) {
  i=i+1;
}
```
────────────────────────────────── JAVA ──────

The sequent

$$i \doteq 0 \Longrightarrow [\texttt{int i=0; while (i<10) \{ i=i+1; \}}]i \doteq 0$$

is obviously *not* valid. In our example program no abrupt termination can occur and the loop condition has no side-effects. We thus can apply the rule invRuleSimple. Let us see what happens, if we use the following (unsound) variant where the context is not deleted from the second and the third premiss (schema version without ($*$)):

$$\text{unsoundRule} \; \frac{\begin{array}{c} \Longrightarrow Inv \\ Inv \,\&\, se \Longrightarrow [p]\,Inv \\ Inv \,\&\, !\,se \Longrightarrow [\pi\ \omega]\phi \end{array}}{\Longrightarrow [\pi\ \texttt{while }(se)\ \{\ p\ \}\ \omega]\phi}$$

Instantiating that rule yields (with $\Gamma = (i \doteq 0)$, $\Delta, \mathcal{U}$ empty, and using the formula true as invariant):

$$\text{unsoundInstance } \frac{\begin{array}{l} \mathtt{i} \doteq 0 \Longrightarrow \text{true} \\ \mathtt{i} \doteq 0,\ \text{true } \&\ \mathtt{i} < 10 \Longrightarrow [\mathtt{i=i+1;}]\text{true} \\ \mathtt{i} \doteq 0,\ \text{true } \&\ !(\mathtt{i} < 10) \Longrightarrow []\mathtt{i} \doteq 0 \end{array}}{\mathtt{i} \doteq 0 \Longrightarrow [\texttt{int i=0; while (i<10) \{ i=i+1; \}}]\mathtt{i} \doteq 0}$$

As one can easily see, the three premisses of this instance are valid but the conclusion is not. The reason for this unsoundness is that the context describes the initial state of the loop execution; however, for correctness, in the second premiss the loop body would have to be executed in an arbitrary state (that is described merely by the invariant) and in the third premiss the invariant and the negated loop condition must entail the postcondition in the final state of the loop execution.

If the invariant rule is to be implemented using the taclet language presented in Chapter 4, there is the problem that taclets do not allow to omit context formulae since they act locally on the formula or term in focus. This is a deliberate design decision and not a flaw of the taclet language, which in most cases is very useful. However, in the case of the invariant rule, it requires to use some additional mechanism. The implementation of the invariant rule using taclets is based on the idea that a special kind of updates can be used to achieve a similar effect as with omitting the context. These special updates are called *anonymising* updates and their intuitive semantics is that they assign arbitrary unknown values to all locations, thus "destroying" the information contained in the context.

Definition 3.59 (Anonymising Update). *Let a JAVA CARD DL signature* $(\text{VSym}, \text{FSym}_r, \text{FSym}_{nr}, \text{PSym}_r, \text{PSym}_{nr}, \alpha)$ *for a type hierarchy, a normalised JAVA CARD program* $P \in \Pi$, *and a sequent* $\Gamma \Longrightarrow \Delta$ *be given.*

For every $f_i : A_1, \dots, A_{n_i} \to A \in \text{FSym}_{nr}$ $(0 \leq i \leq n)$ *occurring in* P *or in* $\Gamma \cup \Delta$ *let* $f_i' : A_1, \dots, A_{n_i} \to A \in \text{FSym}_r$ *be a fresh (w.r.t.* P *and* $\Gamma \cup \Delta$) *rigid function symbol (i.e.,* f_i' *does neither occur in* P *nor in* $\Gamma \cup \Delta$). *Then, the update*

$$u_1 \,\|\, u_2 \,\|\, \cdots \,\|\, u_n$$

with

$$u_i \;=\; \mathtt{for}\ x_1^i;\ \text{true};\ \cdots \mathtt{for}\ x_{n_i}^i;\ \text{true};\ f_i(x_1^i, \dots, x_{n_i}^i) := f_i'(x_1^i, \dots, x_{n_i}^i)$$

is called an anonymising update *for the sequent* $\Gamma \Longrightarrow \Delta$. *In the following we abbreviate an anonymising update with* $\mathcal{V}$.

In the KeY system, the syntax for anonymising updates is $\{* := *n\}$, where $n \in \mathbb{N}$.

Using anonymising updates we can set-up invariant rules for JAVA CARD DL that can be implemented using the taclet language. Here, we only present the variant of the general rule invRule, covering abrupt termination and

side-effects of the loop condition (variants of the rules invRuleSimple and invRuleNse with anonymising updates are obtained analogously).

invRuleAnonymisingUpdate

$\Longrightarrow Inv$
$\mathcal{V}Inv \Longrightarrow \mathcal{V}[\texttt{boolean } v\texttt{=}nse\texttt{;}](v \doteq \text{TRUE} \to ([p]Inv \,\&\, [p]_{continue}Inv))$
$\mathcal{V}Inv, \mathcal{V}\langle\texttt{boolean } v\texttt{=}nse\texttt{;}\rangle_{AT'} \text{true} \Longrightarrow \mathcal{V}[\pi \ v\texttt{=}nse\texttt{; } \omega]\phi$
$\mathcal{V}Inv, \mathcal{V}\langle\texttt{boolean } v\texttt{=}nse\texttt{;}\rangle(v \doteq \text{TRUE} \,\&\, \langle p\rangle_{AT}\text{true}) \Longrightarrow \mathcal{V}[\pi \ v\texttt{=}nse\texttt{;} p \ \omega]\phi$
$\mathcal{V}Inv \Longrightarrow \mathcal{V}[\texttt{boolean } v\texttt{=}nse\texttt{;}](v \doteq \text{FALSE} \to [\pi \ \omega]\phi)$
$$\Longrightarrow [\pi \ \texttt{while (}nse\texttt{) \{ } p \texttt{ \} } \omega]\phi$$

As can be seen, the context remains unchanged in all premisses, but formulae whose evaluation must not be affected by the context are prefixed with an anonymising update $\mathcal{V}$.

Note 3.60. Please note that the above rule looks differently in the KeY system since in the implementation we follow the approach based on a program transformation to deal with abrupt termination of the loop instead of introducing additional modalities $\langle\cdot\rangle_{AT}$ and $[\cdot]_{AT}$ (see the discussion in Sect. 3.7.2).

3.7.4 An Improved Loop Invariant Rule

Performance and usability of program verification systems can be greatly enhanced if specifications of programs and program parts not only consist of the usual pre-/postcondition pairs and invariants but also include additional information, such as knowledge about which memory locations are changed by executing a piece of code. More precisely, we associate with a (sequence of) statement(s) p a set Mod_p of expressions, called the modifier set (for p), with the understanding that Mod_p is part of the specification of p. Its semantics is that those parts of a program state that are *not* referenced by an expression in Mod_p are never changed by executing p [Beckert and Schmitt, 2003].

Usually, modifier sets are used for method specifications ($\Rightarrow$ Chap. 5, 8). In this chapter we extend the idea of modifier sets to loops. Similar as with method specifications, modifier sets for loops allow to

- separate the aspects of (a) which locations change and (b) how they change,
- state the change information in a compact way,
- make the proof process more efficient.

To achieve the latter point, we define a new JAVA CARD DL proof rule for while loops that makes use of the information contained in a modifier set for the loop. The main idea is to throw away only those parts of the context Γ, Δ and $\mathcal{U}$ (i.e., of the descriptions of the initial state) that may be changed by the loop. Anything that remains unchanged is kept and can be used to establish the invariant (second premiss of rule invRuleAnonymisingUpdate) and

the postcondition (premisses 3–5 of rule invRuleAnonymisingUpdate). That is, we do not use an anonymising update $\mathcal{V}$ as in rule invRuleAnonymisingUpdate which assigns unknown values to *all* location but a more restricted update that only assigns anonymous values to *critical* locations.

An important advantage of using modifier sets is that usually a loop only changes few locations, and that only these locations need to be put in a modifier set. On the other hand, using the traditional rule, all locations that are *not* changed and whose value is of relevance have to be included in the invariant and, typically, the number of relevant locations that are not changed by a loop is much bigger than the number of locations that are changed. Of course, in general, not everything that remains unchanged is needed to establish the postcondition in the third premiss. But when applying the invariant rule it is often not obvious what information must be preserved, in particular if the loop is followed by a non-trivial program. That can lead to repeated failed attempts to construct the right invariant. Whereas, to figure out the locations that are (possibly) changed by the loop, it is usually enough to look at the small piece of code in the loop condition and the loop body.

As a motivating example, consider the following JAVA CARD program fragment $p_{\min}$ that computes the minimum of an array a of integers:

─── JAVA (3.1) ───

```
m = a[0]; i = 1;
while (i < a.length) {
  if (a[i] < m) then
    m = a[i];
  i++;
}
```

─── JAVA ───

A postcondition (in KeY syntax) for this program is

$$\phi_{\min} = \forall x.(0 <= x \ \& \ x < \texttt{a.length} \rightarrow \texttt{m} <= \texttt{a}[x]) \ \&$$
$$\exists x.(0 <= x \ \& \ x < \texttt{a.length} \ \& \ \texttt{m} \doteq \texttt{a}[x]) \ ,$$

stating that, after running $p_{\min}$, the variable m indeed contains the minimum of a. However, a specification that just consists of $\phi_{\min}$ is rather weak. The problem is that $\phi_{\min}$ can also be established using, for example, a program that sets m as well all elements of a to 0, which of course is not the *intended* behaviour. To exclude such programs, the specification must also state what the program does modify (the variables i and m) and does not modify (the array a and its elements). One way of doing this is to extend the postcondition with an additional part

$$\phi_{\text{inv}} = \forall x.(0 <= x \ \& \ x < \texttt{a.length} \rightarrow \texttt{a}[x] \doteq \texttt{a\_old}[x])$$

where a_old is a new array variable (not allowed to occur in the program) that is supposed to contain the "old" values of the array elements. To make

sure a_old has the same elements as a, the formula ϕ_{inv} must also be used as a precondition and, thus, be turned into an invariant. In JAVA CARD DL, this specification of $p_{\min}$ is written as $\phi_{\mathrm{inv}} \rightarrow [p_{\min}](\phi_{\min} \ \& \ \phi_{\mathrm{inv}})$.

In order to prove the correctness of the program $p_{\min}$ using the classical invariant rule invRule or the variant with anonymising updates (rule invRuleAnonymisingUpdate), it is crucial to add the formula ϕ_{inv} also to the loop invariant that is used. Otherwise, the loop invariant is not strong enough to entail the postcondition ϕ (the third premiss of the loop rule does not hold). The reason is that all premisses of the invariant rule except for the first one omit the context formulae Γ, Δ and the sequence $\mathcal{U}$ of updates, i.e., all information about the state reached before running the while loop is lost (one can construct similar examples where the second premiss of the rule does not hold). The only way to keep this information—as long as no modifier sets are used—is to add it to the invariant. In general, loop invariants are "polluted" with formulae stating what the loop does *not* do. All relevant properties of the pre-state that need to be preserved have to be encoded into the invariant, even if they are in no way affected by the loop. Thus, two aspects are intermingled:

- Information about what intended effects the loop *does* have.
- Information about what non-intended effects the loop *does not* have.

This problem can be avoided by encoding the second aspect (i.e., the change information) with a modifier set instead of adding it to the invariant. Then the correctness of the program and the correctness of the modifier set can be shown in independent proofs and, thus, the two aspects are separated on verification level as well.

Modifier Sets

We shall now formally define the notion of modifier sets which has been motivated above. Intuitively, a modifier set enumerates the set of locations that a code piece p may change—it is thus part of the specification of p. The question how correctness of a modifier set with respect to a program can be proved is addressed in Sect. 8.3.3.

In general programs and in particular loops can—and in practice often do—change a finite but *unknown* number of locations (though in our simple motivating example $p_{\min}$ the number of changed locations is known to be two). A loop may, for example, change all elements in a list whose length is not known at proof time but only at run time. Therefore, to handle loops, we define modifier sets that can describe location sets of unknown size. Of course, such modifier sets can no longer be represented as simple enumerations of ground terms. Rather, we use guard formulae to define the set of ground terms that may change (this is similar to the use of guard formulae in quantified updates).

Definition 3.61 (Syntax of Modifier Sets). *Let* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr},$ $\mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *be a signature for a type hierarchy.*

A modifier set *Mod is a set of pairs* $\langle \phi, f(t_1, \ldots, t_n)\rangle$ *with* $\phi \in$ *Formulae and* $f(t_1, \ldots, t_n) \in$ *Terms with* $f \in \mathrm{FSym}_{nr}$.

Given a sequent $\Gamma \Longrightarrow \Delta$, *let* $F \subseteq \mathrm{FSym}_{nr}$ *be the set of non-rigid function symbols* $f \in \mathrm{FSym}_{nr}$ *occurring in* $\Gamma \cup \Delta$. *Then,* $\{*\}$ *is the modifier set*

$$\bigcup_{f \in F} \{\langle \mathrm{true}, f(x_1, \ldots, x_n)\rangle\}$$

(specifying that any location in $\Gamma \Longrightarrow \Delta$ *may change).*

The intuitive meaning of a modifier set is that some location $(f, (d_1, \ldots, d_n))$ may be changed by a program p when started in a state S, if the modifier set for p contains an element $\langle \phi, f(t_1, \ldots, t_n)\rangle$ and there is variable assignment β such that the following conditions hold:

1. $\mathrm{val}_{S,\beta}(t_i) = d_i$ for $1 \leq i \leq n$, i.e., β assigns the free logical variables occurring in t_i values such that t_i coincides with d_i.
2. $S, \beta \models \phi$, i.e., the guard formula ϕ holds for the variable assignment β.

For our example program $p_{\min}$, an appropriate modifier set is

$$Mod_{\min} = \{\langle \mathrm{true}, \mathtt{i}\rangle, \langle \mathrm{true}, \mathtt{m}\rangle\} \ .$$

It states in a very compact and simple way that $p_{\min}$ only changes $\mathtt{i}$ and $\mathtt{m}$ and, in particular, does *not* change the array $\mathtt{a}$.

A modifier set *Mod* is said to be correct for a program p if p (at most) changes the value of locations mentioned in *Mod*.

Definition 3.62 (Semantics of Modifier Sets). *Given a signature for a type hierarchy, let* $\mathcal{K}_{\preceq} = (\mathcal{M}, \mathcal{S}, \rho)$ *be a KeY JAVA CARD DL Kripke structure, and let* β *be a variable assignment.*

A pair $(S_1, S_2) = ((\mathcal{D}, \delta, \mathcal{I}_1), (\mathcal{D}, \delta, \mathcal{I}_2)) \in \mathcal{S} \times \mathcal{S}$ *of states satisfies a modifier set Mod, denoted by*

$$(S_1, S_2) \models Mod \ ,$$

iff, for

(a) all $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}_{nr}$,
(b) all $(d_1, \ldots, d_n) \in \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n}$

the following holds:

$$\mathcal{I}_1(f)(d_1, \ldots, d_n) \neq \mathcal{I}_2(f)(d_1, \ldots, d_n)$$

implies that there is a pair $\langle \phi, f(t_1, \ldots, t_n)\rangle \in Mod$ *and a variable assignment* β *such that*

$$d_i = \mathrm{val}_{S_1,\beta}(t_i) \qquad (1 \leq i \leq n)$$

and

$$S_1, \beta \models \phi \ .$$

The modifier set Mod is correct *for a program p, if*

$$(S_1, S_2) \models Mod$$

for all state pairs $(S_1, p, S_2) \in \rho$.

The definition states that, if after running a program p there is some location $(f, (d_1, \ldots, d_n))$ which is assigned a value different from its value in the pre-state, then the modifier set must contain a corresponding entry, i.e., a pair $\langle \phi, f(t_1, \ldots, t_n) \rangle$ such that for some variable assignment β the formula ϕ holds and the t_i evaluate to d_i ($1 \leq i \leq n$). Note that ϕ and the t_i are evaluated in the pre-state.

Example 3.63. Consider the following JAVA CARD method, which has one parameter of type `int[]`.

```
——— JAVA ———————————————————————————
public multiplyByTwo(int[] a) {
  int i = 0;
  int j = 0;
  while (i < a.length) {
    a[i] = a[i] * 2;
    i++;
  }
}
                                          ——— JAVA ———
```

Since `a` is a parameter of the method, the value of `a.length` is unknown. Thus, for giving a correct modifier set, it is not possible to enumerate the locations `a[0], a[1], ..., a[a.length-1]`.

However, a correct modifier set for the above program can be written as

$$\{\langle 0 <= x \ \& \ x < \texttt{a.length}, \texttt{a}[x] \rangle, \langle \text{true}, \texttt{i} \rangle\} \ .$$

Another correct modifier set illustrating that modifier sets are not necessarily minimal is

$$\{\langle 0 <= x \ \& \ x < \texttt{a.length}, \texttt{a}[x] \rangle, \langle \text{true}, \texttt{i} \rangle, \langle \text{true}, \texttt{j} \rangle\} \ .$$

In general, a correct modifier set describes a superset of the locations that actually change.

The modifier set $\{\langle 0 <= x \ \& \ x < \texttt{a.length}, \texttt{a}[x] \rangle\}$ is not correct for the above program, since `i` is changed by the program but not contained in the modifier set.

Based on a modifier set *Mod*, we define the notion of an *anonymising update with respect to Mod*, which is an update that (only) assigns unknown values to those locations that are contained in the modifier set *Mod*.

Definition 3.64 (Anonymising Update w.r.t. a Modifier Set).   *Let a signature* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *for a type hierarchy, a modifier set Mod, and a sequent* $\Gamma \implies \Delta$ *be given. For every* $\langle \phi_i, f_i(t_1^i, \ldots, t_{n_i}^i) \rangle \in Mod$ *with* $f_i : A_1, \ldots, A_{n_i} \to A$, *let* $f_i' \in \mathrm{FSym}_r$ *be a fresh (w.r.t.* $\Gamma \cup \Delta$) *rigid function symbol with the same type as* f_i, *i.e.,* f_i' *does not occur in* $\Gamma \cup \Delta$.

Then the update $\mathcal{V}(Mod) =$

$$
\begin{cases}
\mathcal{V} & \textit{if } Mod = \{*\} \\
u_1 \,||\, \cdots \,||\, u_k & \textit{if } Mod = \{\langle \phi_1, f_1(t_1^1, \ldots, t_{n_1}^1) \rangle, \ldots, \langle \phi_k, f_k(t_1^k, \ldots, t_{n_k}^k) \rangle\}
\end{cases}
$$

with

$\mathcal{V}$ *being an anonymising update for the sequent* $\Gamma \implies \Delta$ *(Def. 3.59) and*
$u_i = \texttt{for } x_1^i; \text{ true}; \cdots \texttt{for } x_{l_i}; \phi_i; f_i(t_1^i, \ldots, t_{n_i}^i) := f_i'(t_1^i, \ldots, t_{n_i}^i)$

where

$$
\{x_1^i, \ldots, x_{l_i}^i\} = fv(\phi_i) \cup fv(t_1^i) \cup \cdots \cup fv(t_{n_i}^i)
$$

is called an anonymising update with respect to *Mod*.

Properties of Anonymising Updates w.r.t. *inReachableState*

An anonymising update $\mathcal{V}(Mod)$ assigns terms an unknown but fixed value. As a consequence, the state it describes is not necessarily reachable by a JAVA CARD program. The idea of an anonymising update however is that it approximates all possible state changes of some program and, thus, we require that anonymising updates preserve *inReachableState* ($\Rightarrow$ Sect. 3.3.5), i.e., the formula

$$
inReachableState \;-\!\!> \{\mathcal{V}(Mod)\}\; inReachableState
$$

is logically valid for any $\mathcal{V}(Mod)$.

Improved Invariant Rule for JAVA CARD DL

We now present an invariant rule that makes use of the information contained in a correct modifier set (if available). This rule is an improvement over rule invRuleAnonymisingUpdate since it keeps as much of the context as possible, i.e., only locations described by the modifier set are assigned unknown values [Beckert et al., 2005b]. In contrast, rule invRuleAnonymisingUpdate assigns

unknown values to *all* locations, no matter whether they can be modified by the loop or not.

The rule loopInvariantRule is identical to rule invRuleAnonymisingUpdate except that instead of the anonymising update $\mathcal{V}$, the update $\mathcal{V}(Mod)$ is used that is anonymising with respect to a modifier set *Mod* being correct for the set

```
{ ;
  if (nse) { p }
  if (nse) { p } if (nse) { p }
  if (nse) { p } if (nse) { p } if (nse) { p }

  ⋮

}
```

of programs. That is, the required modifier must be correct not only for the loop body p and the loop condition *nse* but also for an arbitrary number of iterations which, in classical dynamic logic, is denoted by $(\texttt{if } (nse) \texttt{ \{ } p \texttt{ \})}^*$ using the iteration operator $*$.

---

loopInvariantRule

$\Longrightarrow Inv$
$\mathcal{U}'\,Inv \Longrightarrow \mathcal{U}'[\texttt{boolean } v\texttt{=}nse\texttt{;}](v \doteq \text{TRUE} \mathbin{-\!\!>} ([p]\,Inv \mathbin{\&} [p]_{continue}\,Inv))$
$\mathcal{U}'\,Inv,\ \mathcal{U}'\langle\texttt{boolean } v\texttt{=}nse\texttt{;}\rangle_{AT'} \text{true} \Longrightarrow \mathcal{U}'[\pi\ v\texttt{=}nse\texttt{; } \omega]\phi$
$\mathcal{U}'\,Inv,\ \mathcal{U}'\langle\texttt{boolean } v\texttt{=}nse\texttt{;}\rangle(v \doteq \text{TRUE} \mathbin{\&} \langle p\rangle_{AT} \text{true}) \Longrightarrow$
$\hspace{7cm}\mathcal{U}'[\pi\ v\texttt{=}nse\texttt{;}p\ \omega]\phi$
$\mathcal{U}'\,Inv \Longrightarrow \mathcal{U}'[\texttt{boolean } v\texttt{=}nse\texttt{;}](v \doteq \text{FALSE} \mathbin{-\!\!>} [\pi\ \omega]\phi)$
$\rule{7cm}{0.4pt}$
$\hspace{2cm}\Longrightarrow [\pi\ \texttt{while } (nse) \texttt{ \{ } p \texttt{ \} } \omega]\phi$

---

where $\mathcal{U}' = \mathcal{V}(Mod)$ and *Mod* is a correct modifier set for $(\texttt{if } (nse) \texttt{ \{ } p \texttt{ \})}^*$.

Example 3.65. The following example shows that a "normal" modifier set that is correct for the loop condition and loop body is not sufficient for the rule to be sound.

Consider the program

— JAVA —
```
while ( i<10 ) {
  if ( i>0 ) {
    a = 5;
  }
  i=i+1;
}
```
— JAVA —

which we abbreviate with p in the following. A correct modifier set for the loop body and loop condition would be $Mod = \{\langle\text{true}, \texttt{i}\rangle, \langle\texttt{i} > 0, \texttt{a}\rangle\}$ since

i is modified in any case and a is modified if i is greater than zero. The anonymising update with respect to *Mod* is in this case

$$\mathcal{V}(Mod) = \texttt{for } y_1; \texttt{ true; i} := c \,||\, \texttt{for } y_2; \texttt{ i} > 0; \texttt{ a} := d \ .$$

Setting $\mathcal{U}' = \mathcal{V}(Mod)$ we try to prove the (invalid) formula

$$\{\texttt{i} := 0 \,||\, \texttt{a} := 0\} \,[\pi \ p \ \omega]\texttt{a} \doteq 0 \ .$$

Applying the loopInvariantRule with $Inv = (\texttt{a} \doteq 0)$ yields as a fifth premiss (after some simplification)

$$\{\texttt{a} := 0 \,||\, \texttt{i} := c\} \, \texttt{a} \doteq 0 \Longrightarrow$$
$$\{\texttt{a} := 0 \,||\, \texttt{i} := c\} \,[\texttt{v=i<10;}](\texttt{v} \doteq \text{FALSE} \mathrel{->} [\pi \ \omega]\texttt{a} \doteq 0)$$

which is a valid sequent. We do not show the other four premisses here which are also valid, i.e., the rule is not sound with $\mathcal{U}' = \mathcal{V}(Mod)$.

The reason for the unsoundness here is that *Mod* is a correct modifier set for the loop body and loop condition if executed only once. However, in a loop the body can be executed several times. In our example the modifier set $Mod = \{\langle\text{true}, \texttt{i}\rangle, \langle\texttt{i} > 0, \texttt{a}\rangle\}$ is correct for the program

```
if (i>0) { a=5; } i=i+1;
```

but not for

```
if (i>0) { a=5; } i=i+1; if (i>0) { a=5; } i=i+1; .
```

That is, the anonymising update $\mathcal{V}(Mod)$ only anonymises the locations that are modified in the first iteration of the loop. For the rule to be sound we however need an anonymising update that affects all locations that are changed by any execution of the body.

Usually, a modifier set describes the changes that a given, fixed (part of a) program can perform. In contrast, the modifier set that is required for the rule loopInvariantRule must describe the changes of an aribtrary number of iterations of a given program. This has two serious drawbacks: Firstly, it is unintuitive for the user since he is used to give modifier sets for *one* given program and, secondly, the proof obligation for the correctness of modifier sets ($\Rightarrow$ Sect. 8.3.3) cannot be used offhand for an unknown number if iterations of a program. In the following section we therefore present a method how a (correct) modifier set for the iteration p^* of a program p can be generated automatically from a (correct) modifier set for p.

Generating Modifier Sets for Iterated Programs

The following theorem states how a correct modifier set Mod_p^* for p^* can be obtained if a correct modifier set for p is given.

Theorem 3.66. *Let*

$$Mod_p = \{\langle \phi_1, f_1(s_1^1, \ldots, s_{n_1}^1) \rangle, \ldots, \langle \phi_m, f_m(s_1^m, \ldots, s_{n_m}^m) \rangle\}$$

be a correct modifier for the program p such that the ϕ_i ($1 \leq i \leq m$) are first-order formulae. Then the modifier set Mod_p^* that is the least set satisfying the conditions

- $Mod_p \subseteq Mod_p^*$.
- *If $\langle \psi, g(t_1, \ldots, t_n) \rangle \in Mod_p$, then $\langle \psi', g(t_1', \ldots, t_n') \rangle \in Mod_p^*$ if there is a substitution $\sigma = [x_1/l_1(r_1^1, \ldots, r_{o_1}^1), \ldots, x_k/l_k(r_1^k, \ldots, r_{o_k}^k)]$ such that*
  - *the variables x_i are fresh and of the same type as $l_i(r_1^i, \ldots, r_{o_i}^i)$,*
  - *for each $l_i(r_1^i, \ldots, r_{o_i}^i)$ there is some $\langle \phi, f(s_1, \ldots, s_k) \rangle \in Mod_p$ such that $f = l$ and $k = o_i$, and*
  - *$\sigma(\psi') = \psi$ and $\sigma(g(t_1', \ldots, t_n')) = g(t_1, \ldots, t_n)$.*

is correct for the iteration p^* of p.

Note that the modifier set Mod_p^* is not necessary minimal, even if Mod_p is minimal for p.

In the KeY system we make use of the above theorem, i.e., when applying the rule loopInvariantRule the user is asked to provide a modifier set that is correct for the loop body p and loop condition *nse*. The system then automatically generates a modifier set for the iterated loop body.

Example 3.67. We revisit Example 3.65 and apply Theorem 3.66 in order to obtain a correct modifier set for the iterated loop body.

For the loop in Example 3.65 a correct modifier set for the loop body and the loop condition is

$$Mod = \{\langle \mathtt{true}, \mathtt{i} \rangle, \langle \mathtt{i} > 0, \mathtt{a} \rangle\} .$$

Then, following Theorem 3.66, a correct modifier set for the iterated loop body is

$$Mod^* = \{\langle \mathtt{true}, \mathtt{i} \rangle, \langle \mathtt{i} > 0, \mathtt{a} \rangle, \langle x > 0, \mathtt{a} \rangle\}$$

with the corresponding substitution $\sigma = [x/\mathtt{i}]$.

For another example consider the program

```java
while ( i<10 ) {
  ar[i]=0;
  i=i+1;
}
```

A correct modifier set for the loop body and loop condition is

$$Mod = \{\langle \text{true}, \texttt{i} \rangle, \langle \text{true}, \texttt{ar[i]} \rangle\}$$

and, following Theorem 3.66, a correct modifier set for the iterated loop body is

$$Mod = \{\langle \text{true}, \texttt{i} \rangle, \langle \text{true}, \texttt{ar[i]} \rangle, \langle \text{true}, \texttt{ar[x]} \rangle\}$$

with the corresponding substitution $\sigma = [x/\texttt{i}]$.

3.8 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 ($\Rightarrow$ Sect. 3.6.5) or using the specification (which then, of course, has to be verified). The latter approach is discussed in this section.

Exploiting the specifications is indispensable in order for program verification to scale up. This way, each method only needs to be verified (i.e., executed symbolically) once. In contrast, inlined methods may have to be symbolically executed multiple times, and the size of the proofs would grow more than linearly in the size of the program code to be executed symbolically. Moreover, the source code of a (library) method may not be available. Then, the only way to deal with the invocation of the method is to use its specification.

The specification of a method is called *method contract* and is defined as follows.

Definition 3.68 (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:

- *$Pre \in Formulae$ 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.*
- *$Post \in Formulae$ is the postcondition of the form*

$$(exc \doteq \texttt{null} \rightarrow \phi) \,\&\, (exc\,! \doteq \texttt{null} \rightarrow \psi)$$

  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 formulae 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 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 formulae *Pre* and *Post* are JAVA CARD DL formulae. 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.

In this section, we assume that the method contract to be considered is correct, i.e., the method satisfies its contract. This is a prerequisite for the method contract rule to be correct. The question how to establish correctness of a method contract is addressed in Sect. 8.2.4.

The rule for using a contract for a method invocation in a diamond modality looks as follows:

methodContractTotal

$$\Longrightarrow \{self := se_{target} \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n\} \, Pre$$
$$\{\mathcal{V}(Mod)\} \; exc \doteq \texttt{null} \Longrightarrow$$
$$\quad \{\mathcal{V}(Mod) \,||\, self := se_{target} \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n \,||\, lhs := result\}$$
$$\quad (Post \dashrightarrow \langle \pi \; \omega \rangle \phi)$$
$$\{\mathcal{V}(Mod)\} \; exc \mathrel{!\dot{=}} \texttt{null} \Longrightarrow$$
$$\quad \{\mathcal{V}(Mod) \,||\, self := se_{target} \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n\}$$
$$\quad (Post \dashrightarrow \langle \pi \; \texttt{throw} \; exc; \; \omega \rangle \phi)$$

$$\rule{9cm}{0.4pt}$$
$$\Longrightarrow \langle \pi \; \texttt{lhs=}se_{target}\texttt{.}mname\texttt{(}se_1, \ldots, se_n\texttt{)@}C; \; \omega \rangle \phi$$

where $\mathcal{V}(Mod)$ is an anonymising update w.r.t. the modifier set *Mod* of the method contract.

The above rule is applicable to a method-body-statement

$$lhs\texttt{=}se\texttt{.}mname\texttt{(}t_1, \ldots, t_n\texttt{)@}C;$$

if a contract (*Pre, Post, Mod, total*) for the method $mname(T_1, \ldots, T_n)$ declared in class C is given. Note, that the rule cannot be applied if a contract with the termination marker set to *partial* is given since then termination of the method to be invoked is not guaranteed.

In the first premise we have to show that the precondition *Pre* holds in the state in which the method is invoked after updating the program variables *self* and p_i with the receiver object *se* and with the parameters se_i. This guarantees that the method contract's precondition is fulfilled and we can use the postcondition *Post* to describe the effect of the method invocation, where two cases must be distinguished.

In the first case (second premiss) we assume that the invoked method terminates normally, i.e., the program variable *exc* is `null`. If the method is non-void the return value *return* is assigned to the variable *lhs*. The second case deals with the situation that the method terminates abruptly (third premiss). Note, that in both cases the postcondition *Post* holds and the locations that the method possibly modifies are updated with the anonymising update $\mathcal{V}(Mod)$ with respect to *Mod*. As in the first premiss, the variables for the receiver object and the parameters are updated with the corresponding terms. In case of abrupt termination there is no result but an exception *exc* that must be thrown explicitly in the fourth premiss to make sure that the control flow of the program is correctly reflected.

The rule for the method invocations in the box modality is similar. It can be applied independently of the value of the termination marker.

$$
\begin{aligned}
&\mathsf{method ContractPartial} \\
&\Rightarrow \{self := se \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n\} \; Pre \\
&exc \doteq \texttt{null} \Rightarrow \\
&\quad \{\mathcal{V}(Mod) \,||\, self := se \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n \,||\, lhs := result\} \\
&\quad\quad (Post \rightarrow [\pi\,\omega]\phi) \\
&exc \mathrel{!\dot{=}} \texttt{null} \Rightarrow \\
&\quad \{\mathcal{V}(Mod) \,||\, self := se \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := t_n\} \\
&\quad\quad (Post \rightarrow [\pi\; \texttt{throw}\; exc;\; \omega]\phi) \\
\hline
&\quad\quad\quad \Rightarrow [\pi\;\; lhs\texttt{=}se.\,mname(se_1,\ldots,se_n)\texttt{@}C;\;\; \omega]\phi
\end{aligned}
$$

where $\mathcal{V}(Mod)$ is an anonymising update w.r.t. the modifier set *Mod* of the method contract.

Example 3.69. The following JAVA CARD class `MCDemo` contains the two methods `inc` and `init` that are annotated with JML specifications. The contract of `inc` states that:

- The method terminates normally.
- The result is equal to the sum of the parameter `x` and the literal 1.
- The method is pure, i.e., does not modify any location.

The contract of `init` expresses that:

- The method terminates normally.
- When the method terminates, the result is equal to the sum of the parameter `u` and the literal 1 and the attribute `attr` has the value 100

———— JAVA ————

```java
public class MCDemo {

    int attr;
```

```
/*@ public normal_behavior
  @ assignable \nothing;
  @ ensures \result == x+1;
  @ */
public int inc(int x) {
    return ++x;
}


/*@ public normal_behavior
  @    ensures \result == u+1 && attr == 100;
  @ */
public int init(int u) {
    attr = 100;
    return inc(u);
}

}
```
─────────────────────────────────────── JAVA ───

In this example, we want to prove the (total) correctness of the method `init`. We feed this annotated JAVA CARD class into the JML front-end of the KeY system and select the corresponding proof obligation for total correctness. When we arrive at the point in the proof where method `inc` is invoked we apply the rule methodContractTotal. This is possible even if we have not explicitly set-up a method contract according to Def. 3.68. Rather the KeY system automatically translates the JML specification of method `inc` into the method contract $(Pre, Post, Mod, term)$ with

$$Pre = \text{true}$$
$$Post = result \doteq p_1 + 1$$
$$Mod = \{\}$$
$$term = total$$

Since we have specified that the method `inc` terminates normally, the KeY system generates only the following (slightly simplified) two premises instead of the three in the rule scheme (the one dealing with abrupt termination is omitted):

1. In the first sequent we have to establish the precondition of method `inc` in the state where `inc` is invoked. This is trivial here since the JML specification does not contain a `requires` clause and, thus, the precondition is true.

```
——— KeY ————————————————————————————————————————
==>
 {u_old:=u_lv,
   self:=self_lv,
   x:=u_lv,
   self_lv.attr:=100} true
————————————————————————————————————————— KeY ———
```

2. In the second sequent we make use of the postcondition of method inc.
   In lines 4 and 5 the variables for the receiver object self and the pa-
   rameter x, respectively, are updated with the corresponding parameter
   terms. In the JML specification we have an `assignable nothing;` state-
   ment saying that the method inc does not modify any location. As a
   consequence, the anonymising update $\mathcal{V}(Mod)$ in the rule scheme here is
   empty and, thus, omitted.

```
——— KeY ————————————————————————————————————————
    ==>
    {u_old:=u_lv,
3    self_lv.attr:=100
    self:=self_lv,
    x:=u_lv}
6    (    j = x + 1
      -> \<{method-frame(result->result,
                        source=MCDemo,
9                       this=self): {
              return  j;
           }
12        }\> (result = u_old + 1  &  self.attr = 100))
————————————————————————————————————————— KeY ———
```

The validity of the two sequents shown above can be established automati-
cally by the KeY prover.

Note that the program would also be correct if we omit from the speci-
fication `assignable nothing;`. Then, however, we could not prove the cor-
rectness of the program using the rule methodContractTotal since for the
rule being sound it must be assumed that anything, i.e., in particular the
attribute attr, can be changed by the corresponding method. As a conse-
quence, the validity of the equation `attr = 100` in the formula following
the diamond modality in the above sequent cannot be established. If instead
of the method contract rule the rule for inlining the method body is used,
the correctness of the program can be shown even if the assignable clause is
missing since the implementation of method inc in fact does not modify the
attribute attr.

3.9 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 involves an update and a term, a formula, or another update that it is applied to. Note that in the KeY system both normalisation and application of updates is done automatically; there are no interactive rules for that purpose.

3.9.1 General Simplification Laws

We first define an equivalence relation on the set of JAVA CARD DL updates, which holds if and only if two updates always have the same effect, i.e., represent the same state transition.

Definition 3.70. *Let u_1, u_2 be JAVA CARD DL updates. The relation*

$$\equiv\ \subseteq\ \textit{Updates} \times \textit{Updates}$$

is defined by

$$u_1 \equiv u_2 \quad \textit{iff} \quad \text{val}_{S,\beta}(u_1) = \text{val}_{S,\beta}(u_2)$$

for all variable assignments β and JAVA CARD DL states S.

The first update simplification law expressed in the following lemma is that the sequential and parallel update operators are associative.

Lemma 3.71. *For all $u_1, u_2, u_3 \in$ Updates the following holds:*

- $u_1 \,||\, (u_2 \,||\, u_3) \equiv (u_1 \,||\, u_2) \,||\, u_3$,
- $u_1 \,;\, (u_2 \,;\, u_3) \equiv (u_1 \,;\, u_2) \,;\, u_3$.

This justifies that in the sequel we omit parentheses when writing lists of sequential or parallel updates. However, neither of the operators $||$ and $;$ is commutative, as the following example demonstrates.

Example 3.72. The sequential and parallel update operators are not commutative:

$$
\begin{array}{ccccccc}
i := 0 \,;\, i := 1 & \equiv & i := 1 & \not\equiv & i := 0 & \equiv & i := 1 \,;\, i := 0 \\
i := 0 \,||\, i := 1 & \equiv & i := 1 & \not\equiv & i := 0 & \equiv & i := 1 \,||\, i := 0
\end{array}
$$

Another simple law is that quantification in an update has no effect if the quantified variable does not occur in the scope.

Lemma 3.73. *Let x be a variable, ϕ a JAVA CARD DL formula, and u an update. If ϕ is logically valid and $x \notin fv(\phi) \cup fv(u)$ then*

$$(\texttt{for } x;\ \phi;\ u) \quad \equiv \quad u \ .$$

3.9.2 Update Normalisation

In the following we present a normal form for updates and explain how arbitrary updates are transformed into this normal form. We use "for $\bar{x}$; ϕ; u" and "for $(x_1, \ldots, x_n)$; ϕ; u" to abbreviate "for x_1; true; $\cdots$ for x_n; ϕ; u".

The normal form for updates is a sequence of quantified updates (with function updates as sub-updates) executed in parallel.

Definition 3.74 (Update Normal Form). *An update u is in* update normal form *if it has the form*

$$\text{for } \bar{x}_1;\ \phi_1;\ u_1 \,\|\, \text{for } \bar{x}_2;\ \phi_2;\ u_2 \,\|\, \cdots \,\|\, \text{for } \bar{x}_n;\ \phi_n;\ u_n$$

where the u_i are function updates ($\Rightarrow$ Def. 3.8).

It is crucial for this normal form that the well-ordering of the domain (of a JAVA CARD DL Kripke structure with ordered domain) is expressible in the object logic. For that purpose JAVA CARD DL contains the binary predicate *quanUpdateLeq* . It is used for resolving clashes in quantified updates on a syntactic level. This requires to express that there is an element x satisfying some property ϕ and that it is the smallest such element: $\exists x.(\phi\ \&\ \forall y.([y/x]\phi \rightarrow quanUpdateLeq(x, y)))$.

We now present simplification laws that allow arbitrary updates to be turned into normal form.

Function Updates

A function update $f(t_1, \ldots, t_n) := s$ can easily be transformed into normal form by applying Lemma 3.73:

$$f(t_1, \ldots, t_n) := s \quad \equiv \quad \text{for } x;\ \text{true};\ f(t_1, \ldots, t_n) := s$$

where $x \notin fv(f(t_1, \ldots, t_n) := s)$.

Sequential Update

Sequential updates $u_1 \,;\, u_2$ can be transformed into normal form by applying the following law which introduces an update application $\{u_1\}\ u_2$ ($\Rightarrow$ Sect. 3.9.3).

Lemma 3.75. *For all $u_1, u_2 \in$ Updates:*

$$u_1 \,;\, u_2 \quad \equiv \quad u_1 \,\|\, \{u_1\}u_2$$

Quantified Updates with Non-function Sub-updates

We consider a quantified update `for` x; ϕ; u where u is not a function update (otherwise the update would already be in normal form).

   If u is a sequential update, we apply the previous rule to transform u into a parallel update. The handling of parallel updates however is not that straightforward. For $u = u_1 \,\|\, u_2$, the quantification cannot be simply distributed over the parallel update operator as the following example shows.

Example 3.76. For simplicity, we assume that x ranges only over the nonnegative integers (which shall be ordered as usual). Then,

$$
\begin{aligned}
&\texttt{for } x;\ 0 <= x <= 2;\ (f(x+1) := x \,\|\, f(x) := x) \\
\equiv\quad &f(3) := 2 \,\|\, f(2) := 2 \,\|\, f(2) := 1 \,\|\, f(1) := 1 \,\|\, f(1) := 0 \,\|\, f(0) := 0 \\
\equiv\quad &f(3) := 2 \,\|\, f(2) := 1 \,\|\, f(1) := 0 \,\|\, f(0) := 0 \\
\not\equiv\quad &f(3) := 2 \,\|\, f(2) := 2 \,\|\, f(1) := 1 \,\|\, f(0) := 0 \\
\equiv\quad &f(3) := 2 \,\|\, f(2) := 1 \,\|\, f(1) := 0 \,\|\, f(2) := 2 \,\|\, f(1) := 1 \,\|\, f(0) := 0 \\
\equiv\quad &(\texttt{for } x;\ 0 <= x <= 2;\ f(x+1) := x) \,\| \\
&(\texttt{for } x;\ 0 <= x <= 2;\ f(x) := x)
\end{aligned}
$$

As the above example suggests, a quantified update `for` x; ϕ; u can be understood as a (possibly infinite) sequence $\ldots \,\|\, [x/t_2]u \,\|\, [x/t_1]u$ where instances of the sub-update u are put in parallel for all values satisfying the guard (syntactically represented by terms t_i). To preserve the clash semantics of quantified updates, the order of the updates $[x/t_i]u$ put in parallel is crucial. A term t_i must evaluate to a domain element d_i that is smaller than or equal to all the d_j that the terms t_j, $j > i$, evaluate to. Intuitively, in the sequence $\ldots \,\|\, [x/t_2]u \,\|\, [x/t_1]u$ a term t_i must be smaller than all the terms t_{i+n} occurring to its left to correctly represent the corresponding quantified update, since this guarantees that in case of a clash "the least element wins".

   Distributing a quantification over the parallel-composition operator corresponds to a permutation of the updates in the sequence $\ldots \,\|\, [x/t_2]u \,\|\, [x/t_1]u$, which in general alters the semantics (as the above example shows). Only in the case that no clashes occur, permutations preserve the semantics of parallel updates.

Example 3.77. We revisit the updates from Example 3.76 and visualise the permutation of sub-updates induced by distributing quantification over the parallel-composition operator. The arrows in Fig. 3.11 indicate the order of the updates $[x/t_i]f(x+1) := x$ and $[x/t_i]f(x) := x$ if the quantified updates from Example 3.76 are understood as a sequence of parallel updates.
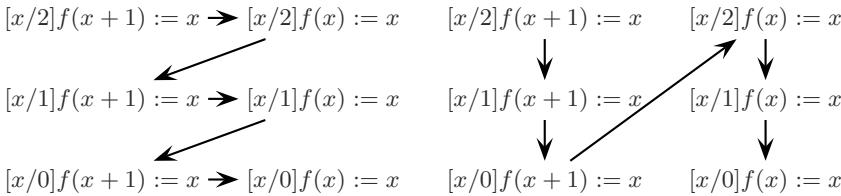
   The left part of Fig. 3.11 shows the order for the update

$$
\texttt{for } x;\ 0 \le x \le 2;\ (f(x+1) := x \,\|\, f(x) := x)
$$

and the right part the order for

$$\texttt{for } x;\ 0 \leq x \leq 2;\ f(x+1) := x \,||\, \texttt{for } x;\ 0 \leq x \leq 2;\ f(x) := x\ ,$$

i.e., after distributing the quantification over the parallel sub-update. The figure shows that the order of clashing parallel updates in the two cases differ. For example, the update $[x/1](f(x) := x)$ clashes with $[x/0](f(x+1) := x)$. In the left part of the figure, the latter update wins, while in the right part, the former update takes precedence.

$$[x/2]f(x+1) := x \rightarrow [x/2]f(x) := x \qquad [x/2]f(x+1) := x \qquad [x/2]f(x) := x$$

$$[x/1]f(x+1) := x \rightarrow [x/1]f(x) := x \qquad [x/1]f(x+1) := x \qquad [x/1]f(x) := x$$

$$[x/0]f(x+1) := x \rightarrow [x/0]f(x) := x \qquad [x/0]f(x+1) := x \qquad [x/0]f(x) := x$$

Fig. 3.11. Evaluation order of quantified updates

Fig. 3.11 does not only illustrate that naive distribution of quantification over parallel composition is not a correct update simplification law, but also gives a hint on how to "repair" the law based on the following two observations:

- If no clashes occur, the parallel update operator is commutative, i.e., in that case the order in a sequence of parallel updates is irrelevant.
- In case of a clash, the update that gets overridden by a later update can simply be omitted (since parallel updates do not influence each other).

The idea is now to distribute the quantification and to add a guard formula ψ to the quantified update to prevent wrong overriding. The formula ψ is constructed in such a way that it evaluates to false if the update would wrongly override another update. For example, in Fig. 3.11 that situation occurs if an update clashes with an other update that is located in a column to its left and in a row below.

That way, an update of the form

$$\texttt{for } x;\ \varphi;\ (u_1 \,||\, u_2)$$

can still be transformed into an update of the shape

$$\texttt{for } x;\ \varphi;\ u_1 \,||\, \texttt{for } x;\ \varphi\ \&\ \psi;\ u_2$$

where u_1 can be assumed to be in normal form

$$\texttt{for } \bar{y}_1;\ \varphi_1;\ v_1 \,||\, \texttt{for } \bar{y}_2;\ \varphi_2;\ v_2 \,||\, \cdots \,||\, \texttt{for } \bar{y}_n;\ \varphi_n;\ v_n$$

and u_2 is an update of the form

$$\texttt{for } (z_1, \ldots, z_o); \, \omega; \, f(t_1, \ldots, t_k) := s \ .$$

The formula ψ which adds additional constraints preventing the right update from overriding the left one in a wrong way looks like

$$\psi = \forall x'.((C_1 \mid \cdots \mid C_n) \rightarrow quanUpdateLeq(x, x'))$$

where x' is a fresh variable and C_i determines whether the i-th part

$$\texttt{for } (y_1, \ldots, y_l); \, \varphi_i; \, g(r_1, \ldots, r_m) := s_i$$

of u_1 might collide with u_2.

This is the case if and only if

- the left hand sides $f(t_1, \ldots, t_k)$ and $g(r_1, \ldots, r_m)$ of both updates syntactically match (i.e. same top-level function symbols $f = g$ and same arities $k = m$) and
- there are values $y_1, \ldots, y_l$ such that the guard φ_i evaluates to true for a value $x' < x$ (i.e. u_2 illicitly overrides u_1 "in a row below") and for that same value x' the arguments t_j and r_j pairwise evaluate to the same value (i.e. there is in fact a clash).

Thus, C_i is defined as

$$
C_i \;\;=\;\; \begin{cases} \exists y_1.\cdots \exists y_l.C_i' & \text{for } f = g, k = m \\ \text{false} & \text{otherwise} \end{cases}
$$

$$
C_i' \;\;=\;\; [x/x']\varphi_i \,\&\, t_1 \doteq [x/x']r_1 \,\&\, \cdots \,\&\, t_k \doteq [x/x']r_k
$$

Example 3.78. We apply the transformation described above to Example 3.76. Since we assume x to range only over non-negative integers with the usual ordering, we can write $y \Lleftarrow z$ instead of $quanUpdateLeq\,(y, z)$.

In a first step we transform the sub-updates of the parallel update $f(x+1) := x \,||\, f(x) := x$ into normal form:

$$
\begin{aligned}
& \texttt{for } x; \, 0 \Lleftarrow x \Lleftarrow 2; \, (f(x+1) := x \,||\, f(x) := x) \\
\equiv \;\; & \texttt{for } x; \, 0 \Lleftarrow x \Lleftarrow 2; \, (\texttt{for } y; \text{true}; \, f(x+1) := x \,|| \\
& \qquad\qquad\qquad\qquad\qquad\qquad \texttt{for } z; \text{true}; \, f(x) := x)
\end{aligned}
$$

Then, we distribute the quantification over the parallel update and add a formula ψ to guarantee the correctness of the transformation.

$$
\begin{aligned}
\equiv \;\; & (\texttt{for } x; \, 0 \Lleftarrow x \Lleftarrow 2; \, \texttt{for } y; \text{true}; \, f(x+1) := x) \,|| \\
& (\texttt{for } x; \, 0 \Lleftarrow x \Lleftarrow 2 \,\&\, \psi; \, \texttt{for } z; \text{true}; \, f(x) := x)
\end{aligned}
$$

where $\psi = \forall x'.(x \doteq x' + 1 \rightarrow x <= x')$. The formula $0 <= x <= 2 \,\&\, \psi$ can be simplified to $x \doteq 0$, and we obtain

$$\equiv \quad \begin{aligned}&(\texttt{for } x;\ 0 <= x <= 2;\ \texttt{for } y;\ \text{true};\ f(x+1) := x)\,||\\&(\texttt{for } x;\ x \doteq 0;\ \texttt{for } z;\ \text{true};\ f(x) := x)\end{aligned}$$

$$\equiv \quad \begin{aligned}&(\texttt{for } x;\ 0 <= x <= 2;\ f(x+1) := x)\,||\\&(\texttt{for } x;\ x \doteq 0;\ f(x) := x)\end{aligned}$$

$$\equiv \quad f(3) := 2\,||\,f(2) := 1\,||\,f(1) := 0\,||\,f(0) := 0$$

The last equivalence shows that the transformed formula is in fact equivalent to the original one (see Example 3.76).

Note 3.79. The normal form for updates consists of quantified updates put in parallel. In KeY we also allow function updates to appear instead of quantified updates, i.e., it is not necessary to transform a function update into a quantified update. The reason is that the majority of updates are function updates and the normal form becomes very clumsy and hard to read if these updates are transformed into quantified updates (see, e.g., Example 3.76).

In the KeY system, the parallel sub-updates of an update in normal form are ordered lexicographically. That makes it possible to close many proof goals without additional rules for permuting the parallel sub-updates.

3.9.3 Update Application

The second part of the update simplification process is the application of updates to other updates, terms, and formulae. Since updates are "semantical" substitutions, the application of an update cannot (always) be effected by a mere syntactical substitution but may require more complex syntactical manipulations.

Applying an Update to an Update

The application of an update to another update is based on the following simplification laws.

Lemma 3.80. *Let an arbitrary update* $u \in$ *Updates and function updates* $u_1, \ldots, u_n \in$ *Updates be given. Then,*

- $\{u\}\,(f(t_1, \ldots, t_n) := s) \quad \equiv \quad f(\{u\}\,t_1, \ldots, \{u\}\,t_n) := \{u\}\,s$
- *if none of the variables in the variable lists* $\bar{x}_i$ *occur in* u

$$\begin{aligned}&\{u\}\,(\texttt{for } \bar{x}_1;\ \phi_1;\ u_1\,||\,\cdots\,||\,\texttt{for } \bar{x}_n;\ \phi_n;\ u_n) \quad \equiv\\&\texttt{for } \bar{x}_1;\ \{u\}\,\phi_1;\ \{u\}\,u_1\,||\,\cdots\,||\,\texttt{for } \bar{x}_n;\ \{u\}\,\phi_n;\ \{u\}\,u_n\end{aligned}$$

In the above lemma only applications of updates to function updates and to updates in normal form are considered. That, however, is sufficient since all updates can be transformed into normal form using the rules from Sect. 3.9.2.

Applying an Update to a Term

In the following, we use the notation $t \equiv t'$ and $\phi \equiv \phi'$ to denote that the terms t, t' resp. the formulae ϕ, ϕ' have the same value in all states for all variable assignments, in which case one can safely be replaced by the other preserving the semantics of the term of formula.

Definition 3.81. *Given terms $t, t' \in Terms$ and formulae $\phi, \phi' \in Formulae$, we write*

- *$t \equiv t'$ if the formula $t \doteq t'$ is logically valid,*
- *$\phi \equiv \phi'$ if the formula $\phi <\!\!-\!\!> \phi'$ is logically valid.*

Lemma 3.82. *Let*

$$u = \texttt{for } \bar{y}_1;\ \phi_1;\ t_1 := s_1 \,\|\, \cdots \,\|\, \texttt{for } \bar{y}_m;\ \phi_m;\ t_m := s_m$$

be an update in normal form. Then,

- *for all rigid terms $t \in Terms$,*

$$\{u\}\, t \quad \equiv \quad t \ ,$$

- *for all terms $f(a_1, \ldots, a_n) \in Terms$,*

$$\{u\}\, f(a_1, \ldots, a_n) \quad \equiv$$
$$\text{if } C_m \text{ then } T_m \text{ else } \ldots \text{ if } C_1 \text{ then } T_1 \text{ else } f(\{u\}\, a_1, \ldots, \{u\}\, a_n)$$

where $C_1, \ldots, C_m$ are guard formulae expressing that the i-th sub-update of u affects the term $f(a_1, \ldots, a_n)$, and $T_1, \ldots, T_m$ are terms that describe the value of the expression in these cases.
C_i and T_i are defined as follows. Suppose that the i-th part of u is of the form

$$\texttt{for } (z_1, \ldots, z_l);\ \phi_i;\ g(b_1, \ldots, b_k) := s_i \ .$$

Then, the formula C_i is defined by

$$C_i \quad = \quad \begin{cases} \exists z_1. \cdots \exists z_l.\, C_i' & \text{if } f = g \text{ and } n = k \\ \text{false} & \text{otherwise} \end{cases}$$

$$C_i' \quad = \quad \phi_i \,\&\, (\{u\}\, a_1) \doteq b_1 \,\&\, \cdots \,\&\, (\{u\}\, a_k) \doteq b_k$$

and the terms T_i are constructed from the s_i by applying substitutions that instantiate the occurring variables with the smallest of clashing values (corresponding to the clash semantics of quantified updates):

$$\begin{aligned} \big[z_1 \,/\, (\text{ifExMin } z_1. \exists z_2. \cdots \exists z_l.\, C_i' \text{ then } z_1 \text{ else } z_1), \\ z_2 \,/\, (\text{ifExMin } z_2. \exists z_3. \cdots \exists z_l.\, C_i' \text{ then } z_2 \text{ else } z_2), \\ \cdots, \\ z_l \,/\, (\text{ifExMin } z_l. C_i' \text{ then } z_l \text{ else } z_l) \big] s_i \end{aligned}$$

- *for all $u_1 \in$ Updates and $t \in$ Terms,*

$$\{u\}\, (\{u_1\}\, t) \quad \equiv \quad \{u\,;u_1\}\, t \ .$$

The order of the C_i and T_i in the second equivalence in the above lemma is relevant. Due to the last-win semantics of parallel updates, the right-most sub-update `for` $\bar{y}_i;\ \phi_i;\ t_i := s_i$, $i = m$, must be checked first, and the left-most sub-update, $i = 1$, must be checked last such that the i-th update "wins" over the j-th update if $i > j$.

Example 3.83. As an example, we consider the term $\{a(o) := t\}\, a(p)$. Intuitively, the update $a(o) := t$ affects the term $a(p)$ iff o and p evaluate to the same domain element. In a first step, we transform the update into normal form:

$$a(o) := t \quad \equiv \quad \texttt{for } y;\ \text{true};\ a(o) := t$$

where y is a fresh variable. Now, we can apply the normalised update on the term $a(p)$ using Lemma 3.82:

$$\{\texttt{for } y;\ \text{true};\ a(o) := t\}\, a(p) \quad \equiv$$
$$\text{if } C \text{ then } T \text{ else } a(\{\texttt{for } y;\ \text{true};\ a(o) := t\}\, p)$$

where

$$C \;=\; \exists y.C'$$
$$C' \;=\; \text{true} \ \& \ (\{\texttt{for } y;\ \text{true};\ a(o) := t\}\, p) \doteq o$$
$$\equiv\; (\{\texttt{for } y;\ \text{true};\ a(o) := t\}\, p) \doteq o$$
$$T \;=\; [y\,/\,(\text{ifExMin } z.C' \text{ then } z \text{ else } y)]t$$
$$=\; t \qquad\qquad (\text{since } y \text{ does not occur in } t)$$

The simplification of $(\{\texttt{for } y;\ \text{true};\ a(o) := t\}\, p)$ yields p since it can be excluded syntactically that this update can affect the non-rigid constant p. Thus, we finally obtain

$$\{\texttt{for } y;\ \text{true};\ a(o) := t\}\, a(p) \quad \equiv$$
$$\text{if } p \doteq o \text{ then } t \text{ else } a(p)$$

which coincides with our intuition.

Applying an Update to a Formula

The following lemma contains simplification laws for applications of updates to formulae. 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.

Lemma 3.84. *Let $u \in$ Updates be an update:*

- $\{u\}\, p(t_1, \ldots, t_n) \;\equiv\; p(\{u\}\, t_1, \ldots, \{u\}\, t_n),$
- $\{u\}$ true $\equiv$ true *and* $\{u\}$ false $\equiv$ false,
- $\{u\}\,(!\,\phi) \;\equiv\; !\{u\}\, \phi,$
- $\{u\}\,(\phi \circ \psi) \;\equiv\; \{u\}\, \phi \circ \{u\}\, \psi$ *for* $\circ \in \{|, \&, -\!\!>\},$
- $\{u\}\,\forall x.\phi \;\equiv\; \forall x.\{u\}\, \phi$ *and* $\{u\}\,\exists x.\phi \;\equiv\; \exists x.\{x\}\, \phi$ *provided that* $x \notin fv(u),$
- $\{u\}\,(\{u_1\}\, \phi) \;\equiv\; \{u\,;u_1\}\, \phi.$

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.

3.10 Related Work

An object-oriented dynamic logic, called ODL, has been defined [Beckert and Platzer, 2006], which captures the essence of JAVA CARD DL, consolidating its foundational principles into a concise logic. The ODL programming language is a While language extended with an object type system, object creation, and non-rigid functions that can be used to represent object attributes. However, it does not include the many other language features, built-in operators, etc. of JAVA. Using such a minimal extension that is not cluttered with too many features makes theoretical investigations much easier. A case in point are paper-and-pencil soundness and completeness proofs for the ODL calculus, which are—though not trivial—still readable, understandable and, hence, accessible to investigation.

A version of dynamic logic is also used in the software verification systems KIV [Balser et al., 2000] and VSE [Stephan et al., 2005] for (artificial) imperative programming languages. More recently, the KIV system also supports a fragment of the JAVA language [Stenzel, 2005]. In both systems, DL was successfully applied to verify software systems of considerable size.

The LOOP tool [Jacobs and Poll, 2001, van den Berg and Jacobs, 2001] translates JAVA programs and specifications written in the Java Modeling Language (JML) into proof goals expressed in higher-order logic. LOOP serves as a front-end to a theorem prover (PVS or Isabelle), in which the actual verification of the program properties takes place, based on a semantics of sequential JAVA that is formalised using coalgebras.

The JIVE tool [Meyer and Poetzsch-Heffter, 2000] follows a similar approach, translating programs that are written in a core subset of JAVA together with their specification into higher-order proof goals. These proof goals can then be discharged using the interactive theorem prover Isabelle.

von Oheimb [2000, 2001b] defines a Hoare calculus for JAVA LIGHT, which is shown to be sound and (relatively) complete. JAVA LIGHT includes side-effects, recursion, dynamic dispatch, exception handling, static class initialisation, object creation, static fields and methods as well as static overloading. And von Oheimb and Nipkow [2002] describe a Hoare calculus for NANOJAVA, which is a further restricted subset of JAVA CARD. Both calculi have been defined in Isabelle/HOL and proven sound and complete relative to a semantics of JAVA LIGHT resp. NANOJAVA specified in Isabelle.

Nipkow [2003] and Klein and Nipkow [2006] define an (artificial) programming language called JINJA to capture the essentials of object-orientation (without giving a calculus). Big step and small step operational semantics for JINJA are shown equivalent, and type safety is proven.

Pierik and de Boer [2003] present a wp-calculus for a moderate abstraction of an object-oriented programming language with a fairly rich set of features (without exceptions) and a focus on method invocation, using an assertion language with quantification over sequences of objects.

Abadi and Leino [1997] define a logic for reasoning about a programming language with prototype-based object inheritance. Their logic resembles a formal type system enriched with pre- and postconditions.

Igarashi et al. [2001] define a λ-calculus for a functional version of JAVA (without assignments), called FEATHERWEIGHT JAVA and use it to investigate JAVA's type-safety as well as parametric type genericity.

4

Construction of Proofs

by
Philipp Rümmer

The primary means of reasoning in a logic are *calculi*, collections of purely syntactic operations that allow us to determine whether a given formula is valid. Two such calculi are defined in Chap. 2 and 3 for first-order predicate logic and for dynamic logic (DL). Having such calculi at hand enables us in principle to create proofs of arbitrarily complex conjectures, using pen and paper, but it is obvious that we need computer support for all realistic applications. Such a mechanised *proof assistant* primarily helps us in two respects: 1. The assistant ensures that rules are applied correctly, e.g., that rules can only be applied if their side-conditions are not violated, and 2. the assistant can provide guidance for selecting the right rules. Whereas the first point is a necessity for making calculi and proofs meaningful, the second item covers a whole spectrum from simple analyses to determine which rules are applicable in a certain situation to the complete automation that is possible for many first-order problems.

Creating a proof assistant requires formalising the rules that the implemented calculus consists of. In our setting—in particular looking at calculi for dynamic logic—such a formalisation is subject to a number of requirements:

- JAVA CARD DL has a complex syntax (subsuming the actual JAVA CARD language) and a large number of rules: first-order rules, rules for the reduction of programs and rules that belong to theories like integer arithmetic. Besides that, in many situations it is necessary to introduce derived rules (*lemmas*) that are more convenient or that are tailored to a particular complex proof. This motivates the need for a language in which new rules can easily be written, rather than hard-coding rules as it is done in high-performance automated provers (for first-order logic). It is also necessary to ensure the soundness of lemmas, i.e., we need a mechanised way to reason about the soundness of rules.
- Because complete automation is impossible for most aspects of program verification, the formalisation has to support interactive theorem proving. KeY provides a graphical user interface (GUI) that makes most rules

applicable only using mouse clicks and drag and drop. This puts a limit on the complexity that a single rule should have for keeping the required user interaction clear and simple, and it requires that rules also contain "pragmatic" information that describes how the rules are supposed to be applied. Accounts on the user interface in KeY are Chap. 10 and [Giese, 2004].

- The formalisation also has to enable the automation of as many proof tasks as possible. This covers the simplification of formulae and proof goals, the symbolic execution of programs (which usually does not require user interaction) as well as automated proof or decision procedures for simpler fragments of the logic and for theories. The approach followed in KeY is to have global *strategies* that give priorities to the different applicable rules and automatically apply the rule that is considered most suitable. This concept is powerful enough to implement complete proof procedures for first-order logic[1] and to handle theories like linear integer arithmetic or polynomial rings mostly automatically.

This chapter is devoted to the formalism called *taclets* that is used in KeY to meet these requirements. The concept of taclets provides a notation for rules of sequent calculi, which has an expressiveness comparable to the "textbook-notation" for rules that is used in Chap. 2 and 3, while being more formal. Compared to textbook-notation, taclets inherently limit the degrees of freedom (non-determinism) that a rule can have, which is important to clarify user interaction. Furthermore, an *application mechanism*—the semantics of taclets—is provided that describes when taclets can be applied and what the effect of an application is.

Historically, taclets have first been devised by Habermalz [2000b,a] under the name "Schematic Theory Specific Rules," with the main purpose of capturing the axioms of theories and algebraic specifications as rules. The language is general enough, however, to also cover all rules of a first-order sequent calculus and most rules of calculi for dynamic logic. The development of taclets as a way to build interactive provers was influenced to a large degree by the theorem prover InterACT [Geisler et al., 1996], but also has strong roots in more traditional methods like tactics and derived rules that are commonly used for higher-order logics (examples for such systems are Isabelle/HOL, see [Nipkow et al., 2002], Coq, see [Dowek et al., 1993], or PVS, see [Owre et al., 1996]). Compared to tactics, the expressiveness of taclets is very limited, for the reasons mentioned above. A further difference is that taclets do not (explicitly) build on a small and fixed set of primitive rules, as tactics do in (foundational) higher-order frameworks like Isabelle, but that a rather large number of taclets are considered as *axioms* that are simply assumed.

---

[1] KeY does not use backtracking, the implemented procedure rather follows the non-destructive approach of Giese [2001].

```
 ─── KeY ──────────────────────────────────────────────────
\functions {
  integer exp(integer, integer);
}
\schemaVariables {
  \term integer a, b;
}
\rules {
  expZero { \find(exp(a, 0)) \replacewith(1) };
  expSucc { \find(exp(a, b)) \sameUpdateLevel
                              \replacewith(a * exp(a, b-1));
                              \add(==> b > 0) };
}
 ──────────────────────────────────────────────── KeY ─────
```

Fig. 4.1. Taclets for an exponentiation function on integers

A recent conceptual introduction to taclets is given by Beckert et al. [2004]. The article lacks, however, many details of how taclets currently are used in KeY, because the taclet concept has constantly been extended over the last years in order to implement the rules of Java Card DL. This chapter gives a more comprehensive description of taclets as they now exist in KeY, and also includes features that were only added recently. At the same time, even in the scope of this chapter many details had to be left out. For an introduction of taclets from a user's point of view—how taclets are applied interactively and automatically when working with KeY—we refer to Chap. 10.

Purpose of this Chapter

In most cases it is not necessary for a user of the theorem prover KeY to define taclets, because KeY already comes with complete implementations of the calculi for first-order and dynamic logic. There are, nevertheless, situations when the introduction of new taclets can be valuable:

- The introduction of lemmas, i.e., of non-axiom taclets that can be derived from existing rules, can help to structure complex proofs. Such taclets can be written to an external file and be loaded on demand. When lemmas are loaded, proof obligations that ensure soundness ($\Rightarrow$ Sect. 4.5) are automatically created by KeY as new proof tasks and have to be proven using already existing taclets. This means that lemmas only can add convenience, but do not increase the set of derivable formulae. Instead of applying lemmas, one could as well apply more basic rules, but this usually leads to a longer and more intricate proof. Typical examples are lemmas about complex arithmetic transformations.
- Assumptions under which a conjecture is to be proven can be formulated as taclets. For verifying an algorithm, we might, for instance, want to

```
─── KeY ─────────────────────────────────────────────
\schemaVariables {
  \term integer a, b, c;
}
\rules {
  expSplit { \find(exp(a, b)) \sameUpdateLevel
                              \replacewith(exp(a, b-c) * exp(a, c));
                              \add(==> c >= 0);
                              \add(==> b >= c) };
}
─────────────────────────────────────────── KeY ───
```

Fig. 4.2. Lemma for the exponentiation function

introduce an exponentiation function *exp* through the clauses

$$exp(a, 0) \; = \; 1, \qquad exp(a, b) \; = \; a \cdot exp(a, b-1) \quad (b > 0)$$

While the two equations can, in principle, simply be added as (quantified) formulae to the conjecture in question, having a large number of such definitions would clutter proofs and would also be very tedious to apply. Fig. 4.1 shows how *exp* can instead be defined with two simple taclets that can be used in a proof exactly like the ordinary rules of a calculus. The keywords and clauses of the taclets are explained in detail later in this chapter (⇒ Sect. 4.4).

Note, that the two taclets of Fig. 4.1 are not lemmas but *axioms*: the normal rules of a calculus will not tell us anything about the function *exp* and will in particular not entail that *exp* actually describes exponentiation. For this reason, such axioms cannot be loaded on demand while proving but have to be defined as part of a problem file that can be loaded by KeY. Based on the axioms, in turn, lemmas can be defined, loaded at a later point, and then also proven correct. An example for such a lemma is the following identity (the corresponding taclet is shown in Fig. 4.2):

$$exp(a, b) \; = \; exp(a, b-c) \cdot exp(a, c) \quad (c \geq 0, \, b \geq c)$$

- More generally, new theories can be defined and axiomatised through appropriate taclets, which is the original intention of the taclet concept. This is described in more detail by Habermalz [2000b,a]. Typical examples would be algebraic datatypes like lists, finite sets or trees, the laws of which can naturally be captured with taclets.

The next pages give all information that is required to write taclets for these purposes. In the whole chapter, we assume that the reader already knows about *sequents* $\Gamma \Longrightarrow \Delta$ and about their meaning (see Sect. 2.5 and 3.4.1).

Organisation of this Chapter

We continue with introducing the concepts and keywords of taclets informally in Sect. 4.1; we look at a number of taclets that implement the rules that are given in Chap. 2 and 3. After that, Sect. 4.2 provides a complete account on schema variables. The two sections (Sect. 4.1 and 4.2) together with Sect. 4.3 about meta variables contain all practical information that is necessary for developing new taclets. An in-depth introduction of the taclet language and a discussion of the soundness aspect of taclets are given in the two remaining sections (Sect. 4.4 and 4.5).

4.1 Taclets by Example

The next pages give a tour through the taclet language and illustrate the most important features with examples. We organise the section along logics of increasing complexity that are defined in the first two chapters: 1. propositional logic, the fragment of first-order predicate logic that is obtained by removing quantifiers, variables and terms, 2. first-order predicate logic, and 3. dynamic logic for JAVA CARD (JAVA CARD DL). Accordingly, many of the taclets discussed here correspond to rules that are given in Chap. 2 and 3, in particular to the rules of Fig. 2.2 and 2.3. As a convention, in this chapter we use `typewriter` font both for schema variables (in order to distinguish them from normal variables x, y) and for taclet names (to distinguish them from rule names like allLeft as in Chap. 2 and 3).

Propositional Rules as Taclets

The first example is the taclet `close` ($\Rightarrow$ Fig. 4.3) representing an axiom that closes a branch of a proof (corresponding to rule close in Fig. 2.2). It can be applied whenever the sequent of a proof leaf contains the same formula both in the antecedent and the succedent.[2] The taclet makes use of two different keywords of the taclet language:

- `\assumes` imposes a condition on the applicability of the taclet and has a sequent as parameter. In the case of `close`, the `\assumes` clause states that the taclet must only be applied if an arbitrary formula `phi` appears both in the antecedent and the succedent of a sequent (the sequent may very well contain further formulae).
- `\closegoal` specifies that an application of the taclet closes a proof branch.

The expression in an `\assumes` clause (like all expressions that turn up in a taclet) may contain *schema variables* like the variable `phi`. A schema

---

[2] In a sequent $\Gamma \Longrightarrow \Delta$ we call the left part Γ the *antecedent* and the right part Δ the *succedent* ($\Rightarrow$ Chap. 3).

```
━━ KeY ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
\schemaVariables {
  \formula phi, psi;
}
\rules {
  close     { \assumes(phi ==> phi) \closegoal };
  impRight { \find(==> phi -> psi) \replacewith(phi ==> psi) };
  cut       { \add(phi ==>); \add(==> phi) };
  mpLeft    { \assumes(phi ==>) \find(phi -> psi ==>)
                                \replacewith(psi ==>) };
}
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ KeY ━━
```

Fig. 4.3. Examples of taclets implementing propositional rules

variable has a *kind* that defines which expressions the variable can stand for
(a precise definition is given in Sect. 4.2). In our example, `phi` represents
an arbitrary formula. More generally, the taclet language provides schema
variables that are necessary for all first-order logics, e.g., kinds for match-
ing variables, terms, and formulae. Further kinds are necessary for rules of
dynamic logic and enable variables representing program entities (like JAVA
statements or expressions).

Note 4.1. The keywords of the taclet language reflect the direction in which
sequent calculus proofs are constructed: we start with a formula that is sup-
posed to be proven and create a tree upwards by *analysing* the formula and
taking it apart. Taclets describe expansion steps (or, as a border case, closure
steps), and by the *application* of a taclet we mean the process of adding new
nodes to a leaf of a proof tree following this description.

In order to describe rules that modify formulae of a sequent, the taclet
language offers keywords for specifying which expression a taclet works
on (the *focus* of the taclet application) and in which way it is modified.
Taclet `impRight` (⇒ Fig. 4.3) corresponds to rule impRight in Fig. 2.2 and
contains clauses to this end:

- `\find` defines a pattern (in this taclet `phi -> psi`, where `phi`, `psi` are
  again schema variables) that must occur in the sequent to which the taclet
  is supposed to be applied. Accordingly, `impRight` can be applied whenever
  an implication turns up in the succedent of a proof leaf.
- `\replacewith` tells how the focus of the application will be altered, which
  for `impRight` means that an implication `phi -> psi` in the succedent
  will be removed, that the formula `phi` is added to the antecedent and
  that `psi` is added to the succedent. In general, when a taclet with a
  `\replacewith` clause is applied, a new proof goal is created from the

previous one by replacing the expression matched in the `\find` part with the expression in the `\replacewith` part (after substituting the correct concrete expressions for schema variables).

Besides rules that modify a term or a formula, there are also rules that add new formulae (but not terms) to a sequent. A typical example is the *cut rule*, which is a rule with two premisses that makes a case distinction on whether a formula `phi` is true or false ($\Rightarrow$ Sect. 3.5.2):

$$\text{cut } \frac{\Gamma, \phi \Longrightarrow \Delta \qquad \Gamma \Longrightarrow \phi, \Delta}{\Gamma \Longrightarrow \Delta}$$

Taclet `cut` ($\Rightarrow$ Fig. 4.3) shows how case distinctions like this can be realised in the taclet language and contains a keyword that has not turned up so far:

- `\add` specifies formulae that are added to a sequent when the taclet is applied. Similarly to `\replacewith`, the argument of `\add` is a sequent that gives a list of formulae to be added to the antecedent and a second list to be added to the succedent.

The taclet `cut` also shows how taclets can be written for rules that have more than one premiss and split a proof branch into two branches. The clauses that belong to different branches are in the taclet separated by semicolons. In case of `cut`, an application creates two new proof goals and adds `phi` to the antecedent in one of the goals and to the succedent in the other one.

The examples above exclusively contained either `\add` or `\replacewith` clauses. It is, however, legal to use both in a taclet, and often they are interchangeable. We could—without changing the meaning of the taclet—write taclet `impRight` also in the following way:

—— Taclet ——

```
impRightAdd { \find(==> phi -> psi) \replacewith(==> psi)
                                     \add(phi ==>) };
```

—————— Taclet ——

Similarly, `\assumes` and `\find` can be combined for specifying that a formula can be modified provided that certain other formulae occur in the sequent. An example is the rule known as *modus ponens*: if a formula `phi` and the implication `phi -> psi` hold, then also `psi` holds. Because the converse is true as well—if `phi` and `psi` hold, then also `phi -> psi`—we can safely eliminate the implication:

$$\text{mpLeft } \frac{\Gamma, \phi, \psi \Longrightarrow \Delta}{\Gamma, \phi, \phi \rightarrow \psi \Longrightarrow \Delta}$$

The taclet `mpLeft` ($\Rightarrow$ Fig. 4.3) implements this rule. If the formulae `phi` and `phi -> psi` both occur in the antecedent of a sequent, then the taclet is applicable and `phi -> psi` can be replaced with `psi`. The assumption `phi` will not be altered by the taclet application.

```
  ──── KeY ────────────────────────────────────────────────────
\sorts {
  \generic G;
}
\schemaVariables {
  \formula phi;            \variables G x;
  \skolemTerm G cnst;      \term G s;
}
\rules {
  allLeft   { \find (\forall x; phi ==>)
              \add ({\subst x; s}phi ==>) };
  allRight  { \find (==> \forall x; phi)
              \varcond (\new(cnst, \dependingOn(phi)))
              \replacewith (==> {\subst x; cnst}phi) };
}
  ──────────────────────────────────────────────────── KeY ────
```

Fig. 4.4. Examples of taclets implementing first-order rules

First-Order Rules as Taclets

Dealing with a calculus for first-order logic (as opposed to propositional logic) using the taclet approach requires handling terms and variables, in particular schema variables for variables and for terms are necessary. As an example, we consider the rule allLeft ($\Rightarrow$ Fig. 2.2) for universal quantifiers, which is implemented by the taclet `allLeft` ($\Rightarrow$ Fig. 4.4). Apart from a variable `phi` for formulae, we need the following schema variables:

- `x` represents logical variables of type `G` that can be bound by a quantifier.
- `s` represents an arbitrary (ground) term with static type `G` (or a subtype of `G`).

The `\find` clause of `allLeft` specifies that the rule is applied to universally quantified formulae of the antecedent. Upon application, the taclet adds an instance of the formula to the antecedent by substituting term `s` for the quantified variable `x`. Because `s` does not turn up in the `\find` clause of the taclet, it can essentially be chosen arbitrarily when applying the taclet, reflecting the nature of the allLeft rule.

The taclet `allLeft` demonstrates a further feature: the rule allLeft is supposed to be applicable for arbitrary static types A of the quantified variable and the substituted term. This is realised in taclets by introducing a type `G` that is marked as `\generic` and that can stand for arbitrary "concrete" types A, in the same way as a schema variable can represent concrete formulae or terms.

The analogue of `allLeft` is the taclet `allRight` ($\Rightarrow$ Fig. 4.4) that realises the rule with same name in Fig. 2.2. In contrast to `allLeft`, here the original

quantified formula can be *replaced* with an instance in which a fresh[3] constant is substituted for the bound variable. There is a particular kind of schema variable for introducing new constants that can be used here: variable `cnst` is defined as a variable of kind `\skolemTerm` and will always represent a fresh constant or function symbol that does not yet turn up in the proof in question. The strange line `\varcond(\new(cnst,\dependingOn(phi)))` becomes important in the presence of meta variables (see Sect. 4.2.1 and 4.3, where detailed explanations are given) and ensures that all meta variables that occur in `phi` also are arguments of the function symbol.

Rewriting Taclets

In all of the taclets that we have looked at so far, the parameter of `\find` clauses were sequents containing exactly one formula. For implementing many first-order rules it is, however, necessary also to modify expressions (formulae or terms) *inside* of formulae, leaving the surrounding formula or term unchanged. Examples are most of the equality rules in Fig. 2.3, where we can use an equation $s \doteq t$ for replacing the term s with t anywhere in a sequent. The taclets that we can use for making rules like this available are called *rewriting taclets*: now, the argument of `\find` is a single formula or term and does not contain the arrow `==>`.

A first and very simple rewriting taclet is `zeroRight` ($\Rightarrow$ Fig. 4.5). It states that 0 is the right identity of addition. In `zeroRight`, the `\find` expression is a term. As we obviously cannot replace terms with formulae, in order to make the taclet well-formed then also `\replacewith` expressions have to be terms (if the `\find` expression were a formula, also `\replacewith` expressions would have to be).

Using `zeroRight`, we can for instance conduct the following proof, where the taclet is used to turn $p(a + 0)$ into $p(a)$. Subsequently, the proof can be closed using `close`.

$$\frac{\overline{p(a) \Rightarrow p(a)}}{p(a + 0) \Rightarrow p(a)}$$

Both of the rules `eqLeft` and `eqRight` ($\Rightarrow$ Fig. 2.3) for applying equations are implemented by taclet `applyEq` ($\Rightarrow$ Fig. 4.5), because a rewriting taclet does not distinguish between a focus in the antecedent and in the succedent. The taclet again uses an `\assumes` clause for demanding the presence of certain formulae in a sequent, here of the appropriate equation in the antecedent, and a `\find` clause for specifying the terms that can be modified. `t1` and `t2` are schema variables for the left and the right side of the equation.

When examining the rules `eqLeft` and `eqRight` carefully, we see that both rules also have a side-condition $\sigma(t_2) \sqsubseteq \sigma(t_1)$ that demands that the type of t_2 is a subtype of the type of t_1. This condition is captured in the taclet `applyEq` by declaring the schema variable `t1` as `strict`:

---

[3] A symbol is called *fresh* for a given proof if it does not occur in the proof.

```
 ─── KeY ─────────────────────────────────────────────────────────
\sorts {
  \generic G;
}
\schemaVariables {
  \formula phi;          \variables G x;
  \term[strict] G t1;    \term G t2;        \term integer intTerm;
}
\rules {
  zeroRight { \find (intTerm + 0) \replacewith (intTerm) };
  removeAll { \find (\forall x; phi) \varcond (\notFreeIn(x, phi))
              \replacewith (phi) };
  applyEq   { \assumes (t1 = t2 ==>) \find(t1) \sameUpdateLevel
                                     \replacewith(t2) };
  applyEqAR { \find (t1 = t2 ==>)
              \addrules ( rewrWithEq { \find (t1) \sameUpdateLevel
                                       \replacewith (t2) } ) };
}
 ──────────────────────────────────────────────────────── KeY ────
```

Fig. 4.5. Examples of rewriting taclets

- The option `strict` demands that the term that is represented by `t1` *exactly* has type `A`, otherwise also subtypes would be allowed.

Because `t2` is non-`strict`, also subtypes are allowed and the condition $\sigma(t_2) \sqsubseteq \sigma(t_1)$ is met.

Implementing the rules for applying equations in a sound way—also for dynamic logic—requires a further feature of the taclet language:

- `\sameUpdateLevel` is a *state condition* and can only be added to rewriting taclets. This clause ensures that the focus of the taclet application (the term that is represented by `t1` in `\find`) does not occur in the scope of modal operators apart from updates. Updates are allowed above the focus, but in this case the equation `t1` $\doteq$ `t2`—or, more generally, all formulae referred to using `\assumes`, `\replacewith` and `\add`—have to be in the scope of the same[4] update.

This keyword is necessary for `applyEq`, because too liberal an application of equations is not sound in dynamic logic ($\Rightarrow$ Sect. 3.5.1). In order to illustrate the effect of `\sameUpdateLevel`, we consider two potential applications of `applyEq`:

---

[4] It is enough if the updates in front of the different constituents have the same effect. This can be determined more or less liberally, e.g., by checking for syntactic identity or by taking laws of updates ($\Rightarrow$ Sect. 3.9) into account.

Illegal: Legal:

$$\frac{x \doteq v+1 \Longrightarrow \{v := 2\}p(v+1)}{x \doteq v+1 \Longrightarrow \{v := 2\}p(x)} \qquad \frac{\{v := 2\}(x \doteq v+1) \Longrightarrow \{v := 2\}p(v+1)}{\{v := 2\}(x \doteq v+1) \Longrightarrow \{v := 2\}p(x)}$$

We have to rule out the left application (by adding the flag `\sameUpdateLevel`) because the equation $x \doteq v+1$ must not be used in the state that is created by the update $v := 2$. The right application is admissible, however, because here the equation is preceded by the same update and we know that it holds if v has value 2.

Compared with the rules of Fig. 2.3, `applyEq` differs in a further aspect: while the rule eqRight, for instance, only *adds* new formulae to a sequent, leaving the original formulae untouched, the taclet `applyEq` directly and destructively modifies formulae. Taclets cannot immediately capture the copy-behaviour of eqRight. We will show later in this section how the behaviour of eqRight could be simulated. In practice, however, the rewriting-behaviour of `applyEq` tends to match the intention of a user better than the copy-behaviour of the rule eqRight: equations are usually applied in order to simplify expressions, there is no reason to keep the original and more complicated expression.

Sometimes it is necessary to impose conditions on the variables that may turn up (or not turn up) in formulae or terms involved. For this purpose, the taclet language offers the keyword `\varcond` that is illustrated in the rewriting taclet `removeAll`. The taclet eliminates universal quantifiers, provided that the variable that is quantified over does not occur in the scope of the quantifier. Because the taclet is a rewriting taclet, it can also be applied in situations in which ordinary quantifier elimination (using rules like `allRight`) is not possible, namely if a quantifier is not top-level.

Nested Taclets

Taclets have restricted higher-order features: it is possible to write taclets that upon application introduce further taclets, i.e., make further taclets available for proof construction.

- `\addrules` has as argument a list of taclets that will be made available when the parent taclet is applied. `\addrules` is used similarly to `\add`, in particular it is possible to introduce different taclets in each branch that a taclet creates.

Consider the taclet `applyEqAR` ($\Rightarrow$ Fig. 4.5), which is an alternative taclet for handling equations and is essentially equivalent to `applyEq`. If the antecedent contains an equality that can be matched by `t1` $\doteq$ `t2`, then applying the taclet results in a new rewriting taclet that replaces a term matched by `t1` with a term matched by `t2`. For the equation $f(a) \doteq b$, for instance, we would obtain the following additional taclet (the whole truth is, however, more complicated and explained in Sect. 4.4.7):

---- Taclet --

```
rewrWithEq { \find (f(a)) \sameUpdateLevel \replacewith (b) };
```

--- Taclet -----

This means that the actual application of an equality is now performed by
two taclets. Due to the \addrules-clause, the set of available taclets is not
fixed but can grow dynamically during the course of a proof. Note that the
generated taclets are not sound in general: `rewrWithEq` above is only ren-
dered sound by the presence of the equation $f(a) \doteq b$ in the antecedent.[5]
Soundness of taclets is discussed in Sect. 4.5. The flag \sameUpdateLevel is
set in `rewrWithEq` for the same reason as for `applyEq`, but now entails that
`rewrWithEq` can only be applied in the same state (below the same updates)
as the taclet `applyEqAR` by which it was introduced.

   Using \addrules, we can also store formulae that might be needed again
later in a proof in the form of a taclet (before applying destructive modifica-
tions) or hide formulae:

---- Taclet --

```
saveLeft  { \find (phi ==>)
            \addrules( insert { \add (phi ==>) } ) };
hideLeft  { \find (phi ==>) \replacewith (==>)
            \addrules( insert { \add (phi ==>) } ) };
```

--- Taclet -----

Rules of Dynamic Logic as Taclets

So far, we have shown examples for taclets representing calculus rules for
propositional and first-order logic. However, taclets are not restricted to these
two logics but can also be used for formally capturing the rules of the dy-
namic logic of Chap. 3 (JAVA CARD DL). A taclet for handling the if-then-else
statement in JAVA CARD is `ifElseSplit` ($\Rightarrow$ Fig. 4.6), which captures the
rule ifElseSplit in Sect. 3.6.3. The basic idea of the taclet is to split the if-
then-else statement into two statements representing the possible branches
($\Rightarrow$ Sect. 3.6). As it was done for formulae, terms and variables, the taclet
makes use of schema variables within programs, in this case a schema vari-
able `#se` for side-effect free expressions and `#s0`, `#s1` representing program
statements.

   As we have seen on page 188, it is possible to apply a rewriting taclet
containing the keyword \sameUpdateLevel only if the application focus and
the formulae referred to using \assumes, \replacewith and \add are in the
scope of the same update. The same holds for taclets that are not rewriting

---

[5] It is also sound to apply `rewrWithEq` in the (direct or indirect) children of the
   sequent, as long as the rules applied in between are *locally sound*. Because sound
   taclets are also locally sound, this is always ensured in our setting.

─── KeY ───

```
\schemaVariables {
  \formula phi;
  \program SimpleExpression #se;        \program Statement #s0, #s1;
}
\rules {
  ifElseSplit { \find (==> \<{.. if(#se) #s0 else #s1 ...}\>phi)
     "if #se true":  \replacewith (==> \<{.. #s0 ...}\>phi)
                     \add (#se = TRUE ==>);
     "if #se false": \replacewith (==> \<{.. #s1 ...}\>phi)
                     \add (#se = FALSE ==>) };
}
```

─── KeY ───

Fig. 4.6. Example of a taclet implementing a rule of Java Card DL

taclets, i.e., where the \find pattern is a sequent or there is no \find clause, although it is not necessary to include the flag \sameUpdateLevel explicitly for such taclets. The following application of ifElseSplit is possible, in which the update $v := a$ occurs in front of all affected formulae:

$$\frac{a \geq 0 \Longleftrightarrow b \doteq \text{TRUE}, \{v := a\}\,(b \doteq \text{TRUE}) \Longrightarrow \{v := a\}\,\langle\ v\text{++};\ \rangle v > 0}{a \geq 0 \Longleftrightarrow b \doteq \text{TRUE} \Longrightarrow \{v := a\}\,\langle\ \text{if}\ (b)\ v\text{++};\ \text{else}\ v\,\text{=-}v;\ \rangle v > 0}$$

$$a \geq 0 \Longleftrightarrow b \doteq \text{TRUE}, \{v := a\}\,(b \doteq \text{FALSE}) \Longrightarrow \{v := a\}\,\langle\ v\,\text{=-}v;\ \rangle v > 0$$

Another feature of taclets used in ifElseSplit are the dots ../... surrounding the program. These dots can be considered a further kind of schema variable and stand for the context in which a statement (here the conditional statement that is eliminated by ifElseSplit) occurs, which can be certain blocks around the statement (like try-blocks) and arbitrary trailing code. In Chap. 3, this context is denoted with π and ω.

Finally, the taclet ifElseSplit shows how the different branches that a taclet creates can be given names for convenience reasons. In the KeY implementation, these strings (like "if #se true") are used as annotations in the proof tree and can make navigation easier. As can be seen here, branch names may also contain schema variables (#se) that are replaced with their concrete instantiation when the taclet is applied.

The remaining chapter is a more systematic and less tutorial-like account on taclets. When taking a step back, we see that the approach is based on two notions or principles that are mostly orthogonal to each other:

- *Schema variables*, the special kind of variables that is used as wildcards in expressions. Schema variables are a concept that also occurs unrelated to taclets, for instance in Chap. 2 and 3 for writing rules in textbook notation, and are a general means of describing *rule schemata* ($\Rightarrow$ Sect. 3.4.3).

As this chapter targets the actual implementation of rules within theorem provers, it is, however, necessary to develop the notion of schema variables in a more formal manner.

- A simple and high-level imperative language for modifying sequents. A program in this language—a *taclet*—describes conditions when a modification is possible, where it is possible, a list of modification statements for adding, removing and modifying formulae, and means of branching and closing proof goals. Speaking in terms of sequent calculi, taclets are suitable for describing and also practically executing almost arbitrary local rules.

Most parts of the chapter relate to these two concepts: Sect. 4.2 is an account on schema variables, whereas the actual taclet language is defined in Sect. 4.4.

4.2 Schema Variables

Despite the name *variable*, schema variables are in the context of KeY a very broad concept: they comprise a large number of different kinds of placeholders that can be used in taclets. All schema variables have in common that they are wildcards for syntactic entities, which can again be different kinds of variables (like logical variables or program variables), terms, formulae, programs or more abstract things like modal operators.

Schema variables are used to define taclets. When a taclet is applied, i.e., when a goal of a proof is modified by carrying out the steps described by the taclet, the contained schema variables will be replaced by *concrete* syntactic entities. This process is called *instantiation* and ensures that schema variables never occur in the proof itself. Instantiation is formally defined in Sect. 4.2.3. In order to ensure that no ill-formed expressions occur while instantiating schema variables with concrete expressions, e.g., that no formula is inserted at a place where only terms are allowed, the *kind* of a schema variable defines which entities the schema variable can represent and may be replaced with.

Example 4.2. In KeY syntax, we declare `phi` to be a schema variable representing formulae and `n` a variable for terms of type *integer*:

```
——— KeY ———————————————————————————————————
\schemaVariables {
  \formula phi;
  \term integer n;
}
                                                 ——— KeY ———
```

The kinds of schema variables that exist in the KeY system are given in Table 4.1. A more detailed explanation of each of the different categories is given

Table 4.1. Kinds of schema variables in the context of a type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$

`\variables` A	Logical variables of type $A \in \mathcal{T}$
`\term` A	Terms of type $B \sqsubseteq A$ (with $A \in \mathcal{T}$)
`\formula`	Formulae
`\skolemTerm` A	Skolem constants/functions of type $A \in \mathcal{T}$
`\program` t	Program entities of type t (from Table 4.2)
`\modalOperator` M	Modal operators that are elements of set M
`\programContext`	Program context

in Sect. 4.2.1. Table 4.1 has been found to be rather stable during the development of KeY in the last years and is not expected to be modified a lot in the future. The part that in our experience used to be altered most frequently (e.g., for adding support for further features of a programming language or completely new languages) are the different types of program entities that can be described with variables of kind `\program` t (see Table 4.2).

For the following definition, recall that a *type hierarchy* ($\Rightarrow$ Chap. 2) is a tuple $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ consisting of a set $\mathcal{T} = \mathcal{T}_d \,\dot\cup\, \mathcal{T}_a$ of types, which can be either non-abstract (dynamic) types ($\mathcal{T}_d$) or abstract types ($\mathcal{T}_a$), and a subtype relation $\sqsubseteq$.

Definition 4.3. *Let $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ by a type hierarchy. A set SV of schema variables over $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ is a set of symbols that are distinct from all other declared symbols, where each schema variable $\mathtt{sv} \in SV$ has exactly one of the kinds from Table 4.1 over $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$.*

4.2.1 The Kinds of Schema Variables in Detail

We can roughly distinguish two different categories of schema variables, those which belong to first-order logic (the upper part of Table 4.1) and the more special kinds that are used to write taclets for Java Card DL.

Variables: `\variables` A

Schema variables for variables can be instantiated with logical variables that have static type A. In contrast to schema variables for terms, logical variables of subtypes of A are not allowed, as such a behaviour has been found to make development of sound taclets difficult ($\Rightarrow$ Sect. 4.5). Schema variables can also occur bound in formulae—which actually is the most common use— and also bound occurrences will be replaced with concrete variables when instantiations are applied (this is illustrated in Example 4.11 below).

Terms: `\term` A

Schema variables for terms can be instantiated with arbitrary terms that have the static type A or a subtype of A. Subtypes are allowed because this

behaviour is most useful in practice: there are only very few rules for which the static type of involved terms *exactly* has to match some given type (in case the reader wants to implement a rule like this, the modifier `strict` ($\Rightarrow$ Sect. 4.2.5) can be used). In general, there are no conditions on the logical variables that may occur (free) in terms substituted for such schema variables. When a term schema variable is in the scope of a quantifier, logical variables can be "captured" when applying the instantiation, which needs to be considered when writing taclets (again, this is illustrated in Example 4.11 below).

Formulae: `\formula`

Schema variables for formulae can be instantiated with arbitrary formulae. As for schema variables for terms, the substituted concrete formulae may contain free variables, and during instantiation variable capture can occur.

Skolem Terms: `\skolemTerm` A

Schema variables for Skolem terms are supposed to be instantiated with terms of the form

$$f_{\mathrm{sk}}(X_1, \ldots, X_n)$$

with a fresh function symbol f_{sk} of type A and a list $X_1, \ldots, X_n$ of *meta variables* as arguments. Meta variables (or *free variables*) are a means of postponing the instantiation of schema variables for terms, which is essential for automated proving, and are described in Sect. 4.3. In most cases, namely if no meta variables are involved, the term degenerates to a fresh constant c_{sk} of type A.

The taclet application mechanism in KeY simply creates new function symbols when a taclet is applied that contains such schema variables. This ensures that the inserted symbols are fresh for a proof and hence can be used as Skolem symbols ($\Rightarrow$ Chap. 2). In order to determine the arguments $X_1, \ldots, X_n$ of a Skolem term, the variable conditions `\new(sk,\dependingOn(te))` have to be used ($\Rightarrow$ Sect. 4.2.6). Adding such conditions can be necessary for ensuring that taclets are sound. For more details see Sect. 4.3.

In practice, there are only few rules that use schema variables for Skolem terms, and most probably the reader will never make use of them in his or her own taclets.

Program Entities: `\program` t

There is a large number of different kinds of program entities that can be represented using program schema variables. Table 4.2 contains the most important ones. A complete list of the currently defined kinds schema variables

Table 4.2. A selection of the kinds of schema variables for program entities

	Expressions
`Expression`	Arbitrary JAVA expressions
`SimpleExpression`	Any expression whose evaluation, for syntactical reasons, cannot have side-effects. It is defined as one of the following: 1. a local variable, 2. an access to an instance attribute via the target expression `this` (or, equivalently, no target expression), 3. an access to a static attribute of the form $t.a$, where the target expression t is a type name or a simple expression, 4. a literal, 5. a compile-time constant, 6. an `instanceof` expression with a simple expression as the first argument, 7. a `this` reference
`NonSimpleExpression`	`Expression`, but not `SimpleExpression`
`Java*Expression`	The same as `SimpleExpression`, but in addition the type of an expression has to be `*`, which can be `Boolean`, `Byte`, `Char`, `Short`, `Int` or `Long`
`LeftHandSide`	A simple expression that can appear on the left-hand-side of an assignment. This amounts to the items 1–3 from the definition of simple expressions above
`Variable`	Local program variables
`StaticVariable`	An access to a static attribute of the form $t.a$, where the target expression t is a type name or a simple expression
	Statements
`Statement`	A single arbitrary JAVA statement
`Catch`	A `catch` clause of a `try` block
	Types
`Type`	Arbitrary JAVA type references
`NonPrimitiveType`	The same as `Type`, but not the primitive types of JAVA
	Miscellaneous
`Label`	A JAVA label

is given in App. B.3.2 on page B.3.2. Most of the kinds introduced here correspond to the schema variables that are used in Chap. 3 (see Table 3.1) in order to define the calculus for JAVA CARD DL. In KeY, the name of a program schema variable always has to start with a hash (like `#se`), mostly for the purpose of parsing schematic programs.

Modal Operators: `\modalOperator` M

When implementing rules for dynamic logic, very often the same rule should be applicable for different modal operators. In most cases the versions of rules for box and diamond operators, for instance, do not differ apart from the fact that different modalities are used. In this situation, having to define essentially the same rule multiple times would be very inconvenient. The

Table 4.3. Modal operators that exist in KeY

`diamond`, `box`	Standard operators ($\Rightarrow$ Chap. 3)
`throughout`	Throughout modality ($\Rightarrow$ Chap. 9)
`diamond_trc`, `box_trc`, `throughout_trc`, `diamond_tra`, `box_tra`, `throughout_tra`, `diamond_susp`, `box_susp`, `throughout_susp`	Chap. 9

problem gets worse with the introduction of further modal operators, as done in Chap. 9.

More concise definitions of rules for a variety of modal operators use schema variables that represent groups of modal operators. An overview of the modalities that exist in KeY is given in Table 4.3, and the syntax for such schema variables is illustrated in the following example:

Example 4.4. We implement the most basic assignment rule for JAVA CARD DL (for assignments with side-effect free left- and right-hand side, see Sect. 3.6):

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

This rule is shown here for the diamond modal operator, but it can be formulated in the same way for other modalities. In taclets, this can be realised by introducing a variable `#normalMod` that represents exactly the admissible operators. The syntax for using such schema variables in taclets is

`\modality{<variable>}{ <program> }\endmodality(<postcondition>)`

The complete declaration of the taclet is given in Fig. 4.7.

Program Context: `\programContext`

Context schema variables cannot be declared explicitly. Instead, there is always at most one variable of this kind that is hidden behind the π and ω in a formula:

$$\langle \pi\ p\ \omega\rangle\phi$$

In KeY syntax, π and ω are simply written as dots:

`\<.. p ...\> phi`

Chap. 3 explains how program contexts are used in rules of JAVA CARD DL. We use the notation $\pi/\omega \in SV$ to talk about the context schema variable itself. An instantiation of the context schema variable π/ω is a pair (α, β) of two program fragments, where the "left half" α only consists of opening braces, opening try blocks and similar "inactive" parts of JAVA, and the "right half" β is a continuation that closes all blocks that were opened in α.

```
── KeY ──────────────────────────────────────────────────────
\schemaVariables{
  \formula phi; \program Variable #loc; \program SimpleExpression #se;
  \modalOperator { diamond, box, diamond_trc,
                   box_trc, throughout_trc } #normalMod;
}
\rules{
  assign { \find (        ==> \modality{#normalMod}
                                {.. #loc = #se; ...}
                              \endmodality(phi) )
            \replacewith ( ==> {#loc:= #se}
                               \modality{#normalMod}
                                 {.. ...}
                               \endmodality(phi) ) };
}
──────────────────────────────────────────────────── KeY ──
```

Fig. 4.7. Assignment taclet from Example 4.4

Example 4.5. We can use context schema variables to enclose a program statement of interest in a context of blocks and following statements. A possible choice for the program fragments α and β is shown here:

$$\underbrace{\text{try\{}}_{\alpha}\ \text{x = y;}\ \underbrace{\text{f(13);\} finally\{ x = 0;\}}}_{\beta}$$

4.2.2 Schematic Expressions

For all families of expressions that are introduced in Chap. 2 and 3, like terms and formulae of first-order logic or of JAVA CARD DL, programs, sequents, etc. we can introduce corresponding *schematic* versions in which appropriate schema variables can be used as surrogates for concrete sub-expressions. For obvious reasons, however, we do not want to repeat all definitions given so far. We only give, as an example, a simplified definition of schematic terms and formulae, based on the more complete languages described in Chap. 3. The augmentation with further connectives is obvious. What is also left out in this chapter is the definition of *schematic JAVA programs*, which are defined in the same spirit as the other schematic expressions.

Definition 4.6 (Basic schematic terms and formulae). *Suppose that* $(\text{VSym}, \text{FSym}, \text{PSym}, \alpha)$ *is a signature for the type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *and SV a set of schema variables over* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$.

The system of sets $\{SchemaTerms_A\}_{A \in \mathcal{T}}$ *of schematic terms of static type A is inductively defined as the least system of sets such that:*

- sv $\in$ *SchemaTerms$_A$ for any schema variable* sv $\in SV$ *that is of kind* \variables *A,* \term *A or* \skolemTerm *A;*
- $f(t_1, \ldots, t_n) \in$ *SchemaTerms$_A$ for any function symbol*

$$f : A_1, \ldots, A_n \to A \in \text{FSym}$$

  and terms $t_i \in$ *SchemaTerms$_{A_i'}$ with $A_i' \sqsubseteq A_i$ for $i = 1, \ldots, n$;*
- $\{\backslash subst$ va$;\ s\}(t) \in$ *SchemaTerms$_A$ for any schema variable* va $\in SV$ *of kind* \variables *B and terms* $s \in$ *SchemaTerms$_{B'}$, $t \in$ SchemaTerms$_A$ with $B' \sqsubseteq B$.*

The set SchemaFormulae of JAVA CARD DL *formulae is inductively defined as the least set such that:*

- phi $\in$ *SchemaFormulae for any schema variable* phi $\in SV$ *of kind* \formula*;*
- $p(t_1, \ldots, t_n) \in$ *SchemaFormulae for any predicate symbol* $p : A_1, \ldots, A_n \in$ PSym *and terms* $t_i \in$ *SchemaTerms$_{A_i'}$ with $A_i' \sqsubseteq A_i$ for $i = 1, \ldots, n$;*
- true, false, $!\phi$, $(\phi \mid \psi)$, $(\phi\ \&\ \psi)$, $(\phi \rightarrow \psi) \in$ *SchemaFormulae for any* $\phi, \psi \in$ *SchemaFormulae;*
- $\forall$va. ϕ, $\exists$va. $\phi \in$ *SchemaFormulae for any* $\phi \in$ *SchemaFormulae and any schema variable* sv $\in SV$ *of kind* \variables*;*
- $\{\backslash subst$ va$;\ s\}(\phi) \in$ *SchemaFormulae for any schema variable* va $\in SV$ *of kind* \variables *B, term* $s \in$ *SchemaTerms$_{B'}$ with $B' \sqsubseteq B$ and formula* $\phi \in$ *SchemaFormulae.*

Note 4.7. According to this definition, schematic expressions never contain logical variables, the set VSym is not used. While this is not a strict necessity, it has been found that *not* considering the case where logical and schema variables simultaneously turn up in expressions significantly simplifies the following sections. Concrete (non-schema) variables are fortunately not necessary for writing taclets, as they can always be replaced with schema variables of kind \variables. The actual logical variables that a taclet operates on, are only determined when the taclet is applied, i.e., when schema variables are replaced with concrete expressions ($\Rightarrow$ Sect. 4.2.3).

Note 4.8. In contrast to Chap. 2, substitutions $\{\backslash subst$ va$;\ s\}$ are here introduced as *syntactic* constructs and not directly as operations on terms or formulae. This is necessary as substitutions are used in taclets and have to be given a formal syntax. Further, we also allow the case that s contains schema variables of kind \variables, which corresponds to substituting terms that contain free variables (non-ground substitutions, compare Sect. 2.5.2). Only considering ground substitutions would be an unreasonable restriction for taclets, but, as a downside, the application of non-ground substitutions is more involved. Sect. 4.2.3 and 4.2.4 describe how substitutions are eliminated when instantiating expressions.

4.2.3 Instantiation of Schema Variables and Expressions

Schema variables are replaced with concrete entities when a taclet is applied. This replacement can be considered as a generalisation of the notion of *ground substitutions* in Chap. 2, and like substitutions the replacement is carried out in a purely syntactic manner. A mapping from schema variables to concrete expressions is canonically extended to terms and formulae.

Definition 4.9 (Instantiation of Schema Variables). *Let the quadruple* $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$ *be a signature for a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *and SV a set of schema variables over* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$. *An* instantiation *of SV is a partial mapping*

$$\iota : SV \to \Big(Formulae \cup \bigcup_{A \in \mathcal{T}} Terms_A \cup Programs \Big)$$

that assigns concrete syntactic entities to schema variables in accordance with Tables 4.1 and 4.2. An instantiation is called complete *for SV if it is a total mapping on SV.*

For sake of brevity, we also talk about instantiations of (schematic) terms, formulae or programs ($\Rightarrow$ Def. 4.6), which really are instantiations of the set of schema variables that turn up in the expression. Given a complete instantiation of such an expression—which in general is more complex than only a single schema variable—we can turn the expression into a concrete one by replacing all schema variables $\mathtt{sv}$ with their concrete value $\iota(\mathtt{sv})$. The extension of ι to arbitrary schematic expressions makes use of a further prerequisite, *(possibly non-ground) substitutions* $[x/s](t)$, which is provided in Sect. 4.2.4 but follows the same idea as the ground substitutions in Sect. 2.5.2. Again, the corresponding definition for instantiation of schematic programs is left out.

Definition 4.10 (Instantiation of Terms and Formulae). *Let* ι *be a complete instantiation of SV. We extend* ι *to arbitrary schematic terms over SV:*

- $\iota(f(t_1, \ldots, t_n)) := f(\iota(t_1), \ldots, \iota(t_n))$
- $\iota(\{\backslash subst\ \mathtt{va};\ s\}(t)) := [\iota(\mathtt{va})/\iota(s)](\iota(t))$

  Likewise, ι *is extended to schematic formulae over SV:*

- $\iota(p(t_1, \ldots, t_n)) := p(\iota(t_1), \ldots, \iota(t_n))$
- $\iota(\mathrm{true}) := \mathrm{true}$ *and* $\iota(\mathrm{false}) := \mathrm{false}$,
- $\iota(!\,\phi) := !\,\iota(\phi)$,
- $\iota(\phi\ \&\ \psi) := \iota(\phi)\ \&\ \iota(\psi)$ *(and correspondingly for* $\phi \mid \psi$ *and* $\phi \mathrel{->} \psi$*),*
- $\iota(\forall \mathtt{va}.\,\phi) := \forall \iota(\mathtt{va}).\,\iota(\phi)$ *and* $\iota(\exists \mathtt{va}.\,\phi) := \exists \iota(\mathtt{va}).\,\iota(\phi)$,
- $\iota(\{\backslash subst\ \mathtt{va};\ s\}(\phi)) := [\iota(\mathtt{va})/\iota(s)](\iota(\phi))$.

Table 4.4. Examples of schematic expressions and their instantiation

Expression t	Instantiation ι	Instance $\iota(t)$
$f(\texttt{te})$	$\{\texttt{te} \mapsto g(a)\}$	$f(g(a))$
$f(\texttt{va})$	$\{\texttt{va} \mapsto x\}$	$f(x)$
$\forall \texttt{va}.\, p(\texttt{va})$	$\{\texttt{va} \mapsto x\}$	$\forall x.\, p(x)$
$\forall \texttt{va}.\, p(\texttt{te})$	$\{\texttt{va} \mapsto x, \texttt{te} \mapsto x\}$	$\forall x.\, p(x)$
$\forall \texttt{va}.\, \texttt{phi}$	$\{\texttt{va} \mapsto x,\ \texttt{phi} \mapsto p(x)\}$	$\forall x.\, p(x)$
$\texttt{phi}\ \&\ p(\texttt{te})$	$\{\texttt{phi} \mapsto q \mid r,\ \texttt{te} \mapsto f(a)\}$	$(q \mid r)\ \&\ p(f(a))$
$p(\texttt{sk}) \mathbin{\texttt{->}} \exists \texttt{va}.\, p(\texttt{va})$	$\{\texttt{sk} \mapsto c,\ \texttt{va} \mapsto x\}$	$p(c) \mathbin{\texttt{->}} \exists x.\, p(x)$
$\{\texttt{\textbackslash subst va; sk}\}(\texttt{phi})$ $\mathbin{\texttt{->}} \exists \texttt{va}.\, \texttt{phi}$	$\{\texttt{sk} \mapsto c,\ \texttt{va} \mapsto x,\ \texttt{phi} \mapsto p(x)\}$	$p(c) \mathbin{\texttt{->}} \exists x.\, p(x)$

Example 4.11. Table 4.4 illustrates the instantiation of the different kinds of schema variables for first-order logic. We use the following schema variables:

```
── KeY ───────────────────────────────────────────────

\schemaVariables {
  \variables A va;          \term A te;
  \formula phi;             \skolemTerm A sk;
}
                                          ─────── KeY ──
```

Further, we assume that $f, g : A \rightarrow A$ are function symbols, $a, c : A$ are constants, $p : A$ and q, r are predicates and $x{:}A$ is a logical variable. The most interesting instantiation takes place in the last line of Table 4.4, where first the schema variables are replaced with terms and variables and then the substitution is applied:

$$
\begin{aligned}
\iota(\{\texttt{\textbackslash subst va; sk}\}(\texttt{phi}) &\mathbin{\texttt{->}} \exists \texttt{va}.\, \texttt{phi}) \\
&= [x/c](p(x)) \mathbin{\texttt{->}} \exists x.\, p(x)) \\
&= p(c) \mathbin{\texttt{->}} \exists x.\, p(x)
\end{aligned}
$$

Note 4.12. Example 4.11 demonstrates the interrelation between schema variables of kind `\variables`, `\term`, and `\formula`. Instantiations of variables of the latter two kinds, like $\iota(\texttt{phi}) = p(x)$, may contain free logical variables that also schema variables `va` of kind `\variables` are instantiated with. Such occurrences can become bound when evaluating an expression like $\iota(\forall \texttt{va}.\, \texttt{phi})$, effectively turning `\term` and `\formula` variables into higher-order variables: the variable `phi` represents a predicate with the formal argument `va`. This feature is essential for taclets like `allLeft` ($\Rightarrow$ Fig. 4.4) or induction rules. A more thorough discussion is given in Sect. 4.4.3.

4.2.4 Substitutions Revisited

The sequent calculus for first-order logic of Chap. 2 makes use of *ground substitutions* (introduced in Sect. 2.5.2) in order to define rules for quantified

formulae and for handling equations. Substitutions are syntactic operations on terms or formulae that replace variables with terms and that, similarly to schema variables, are always eliminated when a rule is applied. Substitutions never turn up in the actual proofs.

In the context of a general rule language like taclets, the restriction to ground substitutions would often be a real limitation. We might, for instance, formulate a taclet that eliminates existential quantifiers if the only possible solution can directly be read off:

─── Taclet ───

```
uniqueEx { \find (\exists va; (va=t & phi))
           \varcond (\notFreeIn (va, t))
           \replacewith ({\subst va; t}phi) };
```

─────────────────────────────────── Taclet ───

The taclet is a rewriting taclet and can also be applied in the scope of quantifiers, for instance in

$$\frac{\Longrightarrow \forall x. \forall z. x \doteq z}{\Longrightarrow \forall x. \exists y. (y \doteq x \,\&\, \forall z. y \doteq z)}$$

where the expression $[y/x](\forall z. y \doteq z)$ has to be evaluated. Note that the substitution applied here is not ground.

Unfortunately, application of non-ground substitutions can raise problems that do not occur in the ground case. While both formulae given in the example above are obviously not valid, the following slight modification of the conclusion (to which `uniqueEx` is applied) shows that the taclet is not sound if the substitution is carried out naively. We rename the innermost bound variable to x, which does not alter the meaning of the lower formula:

$$\frac{\Longrightarrow \forall x. \forall x. x \doteq x}{\Longrightarrow \forall x. \exists y. (y \doteq x \,\&\, \forall x. y \doteq x)}$$

Surprisingly, the result of applying `uniqueEx` is a valid formula (the premiss above the bar). A rule that draws invalid conclusions from valid premisses, however, is not sound. The cause of unsoundness is the application of the substitution in $[y/x](\forall x. y \doteq x)$—following the definitions of Sect. 2.5.2—which turns a formula that is not valid into a valid one. This phenomenon is known as *variable capture* or *collision* and occurs whenever a term containing a (free) variable x is moved into the scope of a quantifier like $\forall x.$ (or any operator binding x). Because the transformation changes the place where x is bound, also the meaning of an expression will be altered drastically.

It is possible to circumvent variable capture by suitable *bound renaming*, i.e., by renaming quantified variables when the danger of captured variables arises. The concept of bound renaming often occurs when working with bound variables: giving variables names (or an identity) is only a means to determine

the place where a variable is bound. One variable can always be exchanged (or renamed) with another (unused) variable. This is also known as α-*conversion*. Such a renaming can as well be performed deeply within formulae. The KeY implementation performs bound renaming automatically whenever it is necessary. In order to ensure that we can always pick a variable that is fresh for some expression or proof, in this section we assume that a signature $(\text{VSym}, \text{FSym}_r, \text{FSym}_{nr}, \text{PSym}_r, \text{PSym}_{nr}, \alpha)$ always contains infinitely many variables for each type. The actual definition of substitutions mostly coincides with the definition of ground substitutions (Sect. 2.5.2, Def. 2.45):

Definition 4.13. *A substitution is a function τ that assigns (possibly non-ground) terms to some finite set of variable symbols* $\text{dom}(\tau) \subseteq \text{VSym}$, *the* domain *of the substitution, with the restriction that:*

If $v \in \text{dom}(\tau)$ and $v : B$, then $\tau(v) \in \text{Terms}_A$, for some A with $A \sqsubseteq B$.

We write $\tau = [u_1/t_1, \ldots, u_n/t_n]$ for the substitution with $\text{dom}(\tau) = \{u_1, \ldots, u_n\}$ and $\tau(u_i) := t_i$.
We denote by τ_x the result of removing a variable from the domain of τ, i.e. $\text{dom}(\tau_x) := \text{dom}(\tau) \setminus \{x\}$ and $\tau_x(v) := \tau(v)$ for all $v \in \text{dom}(\tau_x)$.

When extending substitutions to arbitrary terms or formulae, compared to Def. 2.48 the only interesting and new case are quantifiers, where it can be necessary to perform renaming.

Definition 4.14. *The application of a substitution τ is extended to non-variable terms by the following definitions:*

* $\tau(x) := x$ *for a variable $x \notin \text{dom}(\tau)$.*
* $\tau(f(t_1, \ldots, t_n)) := f(\tau(t_1), \ldots, \tau(t_n))$.

The application of a substitution τ to a formula is defined by:

* $\tau(p(t_1, \ldots, t_n)) := p(\tau(t_1), \ldots, \tau(t_n))$.
* $\tau(\text{true}) := \text{true}$ *and $\tau(\text{false}) := \text{false}$.*
* $\tau(!\,\phi) := !(\tau(\phi))$.
* $\tau(\phi \,\&\, \psi) := \tau(\phi) \,\&\, \tau(\psi)$, *and correspondingly for $\phi \mid \psi$ and $\phi \mathbin{->} \psi$.*
* *If there exists $y \in \text{dom}(\tau) \setminus \{x\}$ such that the variable x (of type A) occurs in $\tau(y)$, then $\tau(\forall x.\,\phi) := \forall z.\,\tau([x/z]\phi)$ for a fresh variable $z : A \in \text{VSym}$. Otherwise, $\tau(\forall x.\,\phi) := \forall x.\,\tau_x(\phi)$. (Correspondingly for $\exists x.\,\phi$.)*

With these definitions, we can repeat the previously unsound application of `uniqueEx` and now get a correct result (that we also would obtain when using KeY):

$$\frac{\Longrightarrow \forall x.\,\forall u.\,x \doteq u}{\Longrightarrow \forall x.\,\exists y.\,(y \doteq x \,\&\, \forall x.\,y \doteq x)}$$

where u is an arbitrary unused variable.

Expressions Modulo Bound Renaming

An elegant approach to bound renaming—that often occurs in the literature—is to always work with the equivalence classes of formulae modulo bound renaming. When following this notion, two formulae like $\forall x.\, p(x)$ and $\forall y.\, p(y)$ are different representatives of the same equivalence class and are considered as *the same* formula. While definitions usually get shorter and less technical this way because bound renaming does not have to be handled explicitly anymore, the differences between the approaches are negligible for an implementation.

Note 4.15. As discussed in Sect. 3.5.1, the substitution of non-rigid terms in dynamic logic (which could happen when applying rules like allLeft or exRight) in combination with modal operators can have unwanted effects and has to be restricted. The KeY implementation implicitly ensures that no erroneous substitutions are carried out when evaluating substitution expressions $\{\backslash \text{subst va};\ t\}(\phi)$ during the application of taclets. In problematic situations, the substitution is delayed and in the next step resolved by adding an equation to the antecedent that has the correct effect.

4.2.5 Schema Variable Modifiers

Some of the schema variable kinds come in more than one flavour: it is possible to change the set of concrete expressions that are represented by the schema variable using certain *modifiers*. Such modifiers can restrict the instantiations allowed for a variable further, or can also modify the meaning of a kind more drastically. The KeY prover currently implements the three modifiers that are given in Table 4.5.

Example 4.16. We define `phi` to represent exclusively *rigid* formulae (instead of arbitrary formulae, see Chap. 3), `te` to represent rigid terms that have *exactly* static type A (subtypes of A are not allowed), and `#slist` to represent a whole sequence of JAVA statements (separated by `;`).

--- KeY ---

```
\schemaVariables {
  \formula[rigid] phi;
  \term[rigid,strict] A te;
  \program[list] statement #slist;
}
```

--- KeY ---

Table 4.5. Modifiers for schema variables

Modifier	Applicable to	
`rigid`	`\term` A `\formula`	Terms or formulae that can syntactically be identified as rigid
`strict`	`\term` A	Terms of type A (and not of proper subtypes of A)
`list`	`\program` t	Sequences of program entities. The type t can be any of the types of Table 4.2 apart from `Label`, `Type` and `NonPrimitiveType`. Sequences of expressions can be used to represent arguments of method invocations.

4.2.6 Schema Variable Conditions

The simple notion of *kinds* of schema variables described in the previous sections is often not expressive enough for writing useful taclets. In many cases, one has to impose further restrictions on the instantiations of schema variables, e.g., state that certain logical variables must not occur free in certain terms. The taclet formalism is hence equipped with a simple language for expressing such conditions, *variable conditions*. To each taclet, a list of variable conditions can be attached ($\Rightarrow$ Sect. 4.4.1), which will be checked when the taclet is about to be applied.

Table 4.6 contains the most important variable conditions in KeY. Particularly useful is the `\notFreeIn` condition that is frequently needed for defining theories using taclets.

4.2.7 Generic Types

Schema variables for terms, logical variables and Skolem terms are typed and may only be instantiated with terms or variables of certain static types. While such schema variables are in principle sufficient for implementing all rules of a calculus for dynamic logic, this would not be particularly convenient: for certain rules, a number of taclets would be required, one for each existing type. Example for such rules are allLeft, allRight ($\Rightarrow$ Fig. 2.2).

To handle this situation in a better way, the taclet formalism provides the possibility of writing *generic* taclets, i.e., taclets in which the types of schema variables involved are flexible and are assigned only when the taclet is applied. The concept resembles schema variables, which represent concrete syntactic entities and are instantiated when applying the taclet. When writing taclets, we thus distinguish between *concrete* types, which are exactly the types defined in Chap. 2, and *generic* types that are mapped to concrete types when applying the taclet. Like schema variables, generic types can only be used for defining taclets and are not part of the actual signature of a logic.

Note 4.17. Generic types should not be confused with the *abstract* types of Sect. 2.1, the two notions are not related. For technical reasons, we will in

Table 4.6. Schema variable conditions

First-Order Conditions	
`\notFreeIn(va, te)` `\notFreeIn(va, fo)`	The logical variable that is the instantiation of `va` must not occur (free) in the instantiation of `te`/`fo`.
`\new(sk, \dependingOn(te))` `\new(sk, \dependingOn(fo))`	If `sk` is instantiated with $f_{\mathrm{sk}}(X_1, \ldots, X_n)$, then ensure that $X_1, \ldots, X_n$ contains all meta variables that occur in the instantiation of `te`/`fo`. There can be more than one such condition for `sk`. Also see Sect. 4.3.
Introducing Fresh Local Program Variables (KeY will create a new program variable as instantiation of `#x` when a taclet with such a condition is applied)	
`\new(#x, *)`	`#x` will have the JAVA type `*` (for instance `int[]`).
`\new(#x, \typeof(#y))`	`#x` will have the same JAVA type as the instantiation of `#y`.
`\new(#x, \elemTypeof(#y))`	The instantiation of `#y` has to be of a JAVA array type. Its component type will be the type of `#x`.

The schema variables used above are of the following kinds:

`va`	`\variables`
`te`	`\term`
`fo`	`\formula`
`sk`	`\skolemTerm`
`#x`, `#y`	`\program LeftHandSide` or `\program Variable`

fact consider generic types as dynamic (i.e., non-abstract), but when applying a taclet a generic type can represent both abstract and non-abstract concrete types.

For introducing generic types we extend the notion of a type hierarchy ($\Rightarrow$ Chap. 2). Because *generic type hierarchies* can still be seen as normal type hierarchies by simply ignoring the distinction between generic and concrete types, Def. 4.6 about schematic expressions does not have to be changed but also covers terms or formulae containing "generic parts."

Definition 4.18. *A* generic type hierarchy *is a tuple* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \mathcal{T}_g, \sqsubseteq, \mathcal{R}_g)$ *of*

- *a set of* static types $\mathcal{T}$,
- *a set of* dynamic types $\mathcal{T}_d$,
- *a set of* abstract types $\mathcal{T}_a$,
- *a set of* generic types $\mathcal{T}_g$,
- *a subtype relation* $\sqsubseteq$, *and*
- *a range relation* $\mathcal{R}_g$ *of the generic types*

such that:

- *Each generic type is a dynamic type, but is not universal:* $\mathcal{T}_g \subseteq \mathcal{T}_d \backslash \{\top\}$

```
\sorts {
  \generic GenA;
  \generic GenRefType \extends Object;
}
```
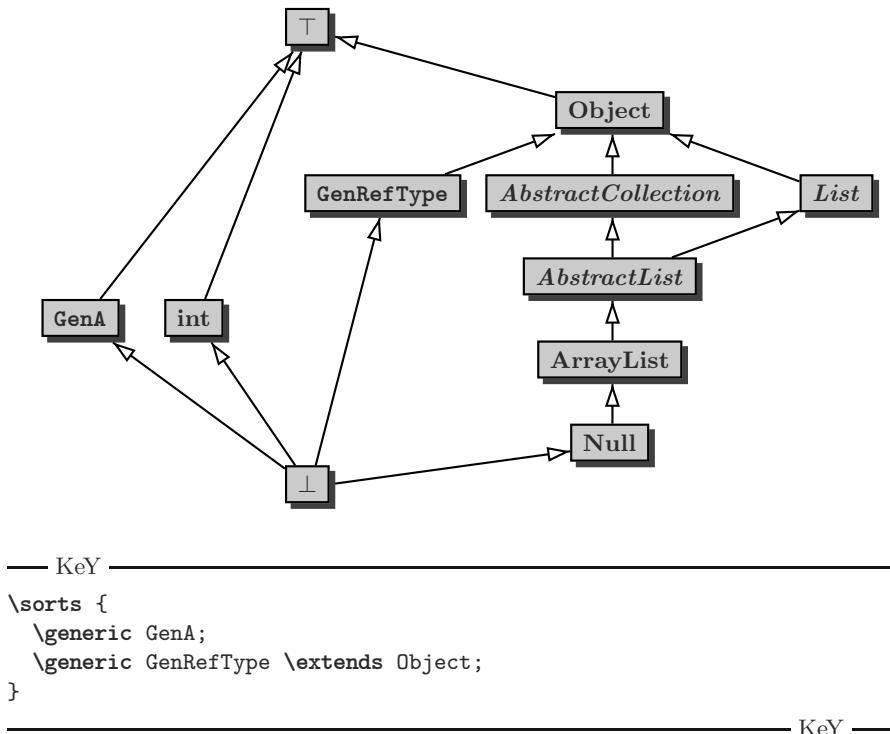
Fig. 4.8. An example type hierarchy

- $\mathcal{R}_g$ *is a relation between generic and concrete types:* $\mathcal{R}_g \subseteq \mathcal{T}_g \times (\mathcal{T} \backslash \mathcal{T}_g)$
- $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *is a type hierarchy* ($\Rightarrow$ *Def. 2.1*)
- *The concrete (non-generic) types also form a type hierarchy on their own:*

$$\big(\mathcal{T} \backslash \mathcal{T}_g, \ \mathcal{T}_d \backslash \mathcal{T}_g, \ \mathcal{T}_a, \ \sqsubseteq \cap \ \big((\mathcal{T} \backslash \mathcal{T}_g) \times (\mathcal{T} \backslash \mathcal{T}_g)\big)\big)$$

  is a type hierarchy
- *The subtypes* $A \sqsubseteq G$ *of generic types* $G \in \mathcal{T}_g$ *are either generic or empty:* $A \in \mathcal{T}_g \cup \{\bot\}$

Example 4.19. Fig. 4.8 adds two generic types to the type hierarchy shown in Example 2.6:

- `GenA`, which is subtype of $\top$ and can thus be used to denote *arbitrary* concrete types in a taclet, and
- `GenRefType`, which can only represent reference types (subtypes of class Object).

As range relation, we choose the full relation $\mathcal{R}_g = \mathcal{T}_g \times (\mathcal{T} \backslash \mathcal{T}_g)$, which means that $\mathcal{R}_g$ does not impose any restrictions on the instantiation of generic types. The figure also shows how to declare the two generic types in the concrete syntax of KeY (provided that type Object exists). After the declaration, we could use the types in taclets as illustrated in Sect. 4.1, for instance in order to implement the rule allLeft.

When a taclet containing generic types (i.e., containing schema variables with generic type) is applied, first an instantiation of these types with concrete types is chosen: all generic types that occur in the taclet are replaced with concrete types. It can then be checked whether instantiations of schema variables are allowed according to Table 4.1. Instantiations of generic types cannot be arbitrary, however, as the creation of ill-formed terms or formulae has to be prevented. Referring to the types of the previous example, a taclet could, for instance, contain the term $f(\mathtt{te})$, where $f : \mathrm{Object} \to \mathrm{Object}$ is a function symbol and $\mathtt{te}$ a schema variable of kind \term GenRefType (note that GenRefType is a subtype of Object). It is obvious that we would run the risk of ill-formed terms if GenRefType was allowed to be instantiated with types that themselves are *not* subtypes of Object, because then also $\mathtt{te}$ could be replaced with terms whose type is not a subtype of Object. This insight is generalised by demanding that type instantiations always are *monotonic* w.r.t. the subtype relation.

Definition 4.20. *Given a generic type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \mathcal{T}_g, \sqsubseteq, \mathcal{R}_g)$, *a type instantiation is a partial mapping* $\iota_t : \mathcal{T} \to \mathcal{T}$ *such that:*

- ι_t *is defined on concrete types* $A \in \mathcal{T} \backslash \mathcal{T}_g$, *which are fixed-points:* $\iota_t(A) = A$.
- *Generic types* $G \in \mathcal{T}_g$ *are mapped to concrete types:* $\iota_t(G) \notin \mathcal{T}_g$ *(provided that* $\iota_t(G)$ *is defined).*
- *The mapping is monotonic: if* $A \sqsubseteq B$ *then also* $\iota_t(A) \sqsubseteq \iota_t(B)$ *(provided that* $\iota_t(A)$ *and* $\iota_t(B)$ *are defined).*
- *The instantiation* $\iota_t(G)$ *of a generic type* $G \in \mathcal{T}_g$ *is within the range of* G: $(G, \iota_t(G)) \in \mathcal{R}_g$ *(provided that* $\iota_t(G)$ *is defined).*

Example 4.21. For the type hierarchy that is declared in Example 4.19, one possible type instantiation is given by

$$\iota_t(\mathtt{GenA}) = \top, \qquad \iota_t(\mathtt{GenRefType}) = AbstractList, \qquad \iota_t(A) = A \quad (A \notin \mathcal{T}_g)$$

An instantiation $\iota_t(\mathtt{GenRefType}) = int$ would not be allowed, because there is no monotonic extension of this mapping to the set $\mathcal{T}$ of all types that maps Object to itself.

Given the notion of a type instantiation, we can augment Def. 4.9 to also take schema variables of generic types into account. First, we can extend type instantiations ι_t to the kinds of schema variables:

$$
\begin{aligned}
\texttt{\textbackslash term } A &\;\mapsto\; \texttt{\textbackslash term } \iota_t(A) \\
\texttt{\textbackslash variables } A &\;\mapsto\; \texttt{\textbackslash variables } \iota_t(A) \\
\texttt{\textbackslash skolemTerm } A &\;\mapsto\; \texttt{\textbackslash skolemTerm } \iota_t(A) \\
k &\;\mapsto\; k \qquad \text{(all other kinds)}
\end{aligned}
$$

In the presence of generic types, instantiations are then described by a pair consisting of a schema variable instantiation and a type instantiation:

Definition 4.22. *Let* $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$ *be a signature for a generic type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \mathcal{T}_g, \sqsubseteq, \mathcal{R}_g)$ *and* SV *a set of schema variables over the same type system. An* instantiation under generic types *of a set* SV *of schema variables is a pair* (ι_t, ι), *where*

- ι_t *is a type instantiation that is defined for all types of variables in* SV, *and*
- ι *is a partial mapping (as in Def. 4.9)*

$$
\iota : SV \to \big(Formulae \cup \bigcup_{A \in \mathcal{T}} Terms_A \cup Programs \big)
$$

such that, for each schema variable $\texttt{sv} \in SV$ *of kind* k *with* $\iota(\texttt{sv}) \neq \bot$, $\iota(\texttt{sv})$ *is an admissible instantiation for a schema variable of kind* $\iota_t(k)$.

Example 4.23. The purpose of *range relations* $\mathcal{R}_g$ is to provide a more direct control over the concrete types that can be chosen for generic ones. If we, for instance, would like to write a taclet that only can be applied for abstract types, we could strengthen the declaration of Fig. 4.8:

— KeY —

```
\sorts {
  \generic GenA;
  \generic GenRefType \extends Object
          \oneof { AbstractCollection, AbstractList, List };
}
\schemaVariables {
  \term[strict] GenRefType te;
}
```
                                                              — KeY —

The schema variable `te` would then exclusively represent terms of the types `AbstractCollection`, `AbstractList`, `List`. Leaving out the keyword `strict`, `te` could also stand for terms of the subtypes `ArrayList` and `Null`.

Generic Types vs. Schema Variables

The role of generic types is comparable to that of schema variables, and an alternative way to define taclets that are parametric over types would indeed be to have a concept of type schema variables. The two approaches only represent different views on the same idea. While the differences are negligible for an implementation, we believe that including generic types in the normal type hierarchy enables an easier theory and presentation: in this approach, also schematic terms that occur in a taclet are well-typed.

4.2.8 Meta-operators

Certain operations or transformations, in particular, operations that are performed on Java programs, are difficult or tedious to capture using only taclets. Examples are arithmetic operations on literals, which would be rather inefficient when handled by taclets, or taclets that need to access the Java program that is verified, e.g., for inserting method bodies when method invocations are executed ($\Rightarrow$ Sect. 3.6). In order to handle such cases, a number of very specific operators have been introduced in KeY that are used in taclets implementing the DL calculus from Chap. 3. Theoretically, such *meta-operators* can be seen as an extension of Def. 4.10 about instantiation of terms and formulae. Practically, they allow the execution of arbitrary Java code in taclets. Because it is virtually impossible to properly describe their semantics, meta-operators should be avoided in user-written taclets. We only give one example of a meta-operator:

- `#method-call`: transforms its argument, which is a method reference, into an if-else-cascade simulating dynamic binding by case distinction on the runtime type of the target object.

4.3 Instantiations and Meta Variables

Before a taclet can be applied, generic types and schema variables need to be instantiated. Selecting the expression that a schema variable `sv` represents is comparatively easy if the variable turns up in the `\find` or `\assumes` clause of a taclet: in this case, the expression already has to be part of the sequent to which the taclet is applied and can be found by *matching* the formulae of the sequent with the schematic terms or formulae of `\find` and `\assumes`. This situation also allows for a simple automated application of taclets.

There can also be schema variables that only turn up in the goal templates of a taclet, i.e., only in `\replacewith` or `\add` clauses. The most well-known example for such a rule is the taclet `allLeft` ($\Rightarrow$ Sect. 4.1), in which the schema variable `s` for terms only occurs in `\add` (a further taclet containing

such schema variables is `expSplit` ($\Rightarrow$ Fig. 4.2)). This means that the term that is represented by `s` can be chosen arbitrarily when applying the taclet. While making this choice can already be a difficult task for the human user of a proof assistant, the automated application of the taclet is even more hampered, and it is necessary to put a large amount of "intelligence," heuristics and knowledge about the particular problem domain into an automatic search strategy for guessing the right terms. The problem is also made difficult by the fact that the terms that need to be inserted do in general not appear in the proof up to this point (although they often do in practice). It can be necessary to "invent" or synthesise completely new terms.

A general approach to overcome the problem, which has been developed in the area of automated theorem proving, are "free variables" or "meta variables." Meta variables are place-holders that can be introduced instead of actual expressions when applying rules. For an extensive account on meta variables in first-order logic see, for instance, [Fitting, 1996]. While meta variables can in principle be introduced for all kinds of expressions, most commonly (and also in KeY) they are only used as place-holders for terms: whenever a taclet is applied that contains schema variables for terms that only occur in goal templates, instead of immediately choosing the instantiation of the schema variables, meta variables can be introduced.

At some point after introducing a meta variable, it often becomes obvious which term the meta variable should stand for, or it becomes necessary to choose a certain term in order to apply a rule. This is achieved by applying a *substitution* ($\Rightarrow$ Sect. 4.2.4) to the whole proof tree that replaces the meta variable with the chosen term. Because such a replacement is a destructive operation (substituting the wrong term can make it necessary to start over with parts of the proof), KeY follows the non-destructive approach that is described in [Fitting, 1996, Giese, 2001] and actually never applies substitutions. Instead of substitutions, *constraints* are stored that describe substitution candidates. Constraints are generated whenever the application of substitutions becomes necessary in order to apply rules and are attached to formulae and to proof goals.

Not all terms can be substituted for meta variables. Because meta variables are considered as *rigid* symbols, in particular, it is not allowed to substitute non-rigid terms (like program variables) for meta variables. In practice, this seriously limits the usefulness of meta variables when doing proofs in dynamic logic and is an issue that belongs to the "Future Work" section.

An example for the usage of meta variables and constraints is given in Chap. 10.

Skolem Symbols and the "Occurs Check"

The concept of meta variables collides, to a certain degree, with rules that are supposed to introduce fresh symbols (like `allRight` in Sect. 4.1). The problematic situation is as follows: by applying substitutions to a proof tree,

the sequents that rules are operating on can be modified *after* actually applying the rules. This means that we can no longer be sure that a symbol that does not turn up in sequents actually is fresh, because it could also be inserted at a later point through a substitution. In order to illustrate this phenomenon, we try to prove the (non-valid) formula $\exists x. \forall y. x \doteq y$ using a meta variable X:

$$\frac{\dfrac{\implies \exists x. \forall y. x \doteq y, X \doteq c_{\mathrm{sk}}}{\implies \exists x. \forall y. .x \doteq y, \forall y. X \doteq y} \text{ allRight}}{\implies \exists x. \forall y. x \doteq y} \text{ exRight}$$

At this point, it becomes obvious that we would like to substitute c_{sk} for X:

$$\frac{\dfrac{\implies \exists x. \forall y. x \doteq y, c_{\mathrm{sk}} \doteq c_{\mathrm{sk}}}{\implies \exists x. \forall y. x \doteq y, \forall y. c_{\mathrm{sk}} \doteq y} \text{ allRight}}{\implies \exists x. \forall y. x \doteq y} \text{ exRight}$$

The proof can now be closed by applying eqClose to the formula $c_{\mathrm{sk}} \doteq c_{\mathrm{sk}}$. Searching for the mistake, we see that the application of allRight becomes illegal after applying the substitution, because the constant c_{sk} that is introduced is no longer fresh.

There is a simple and standard solution to this inconsistency: when introducing symbols that are supposed to be fresh, critical meta variables have to be listed *as arguments* of the fresh symbols. A correct version of our proof attempt is:

$$\frac{\dfrac{\implies \exists x. \forall y. x \doteq y, X \doteq c_{\mathrm{sk}}(X)}{\implies \exists x. \forall y. x \doteq y, \forall y. X \doteq y} \text{ allRight}}{\implies \exists x. \forall y. x \doteq y} \text{ exRight}$$

It is easy to see that no substitution can make the terms X and $c_{\mathrm{sk}}(X)$ syntactically equal, so that it is impossible to close the proof: the terms are *not unifiable*, because X occurs in the term that it is supposed to represent. This situation is known as a failing *occurs check*.

When writing taclets that introduce fresh symbols (using schema variables of kind \skolemTerm), it is currently necessary to specify the meta variables that have to turn up as arguments of the symbol by hand using the variable condition \new(...,\dependingOn(...)) ($\Rightarrow$ Sect. 4.2.6). An example is the taclet allRight ($\Rightarrow$ Sect. 4.1). It is likely, however, that these dependencies will be computed automatically by KeY in the future.

4.4 Systematic Introduction of Taclets

This section introduces the syntax[6] and semantics of the taclet language. The first pages are written in the style of a reference manual for the different taclet constructs and provide most of the information that is necessary for writing one's own taclets. Later, the meaning of taclets is defined in a more rigorous setting.

4.4.1 The Taclet Language

$\langle taclet \rangle$ ::=
  $\langle identifier \rangle$ {
    $\langle contextAssumptions \rangle$? $\langle findPattern \rangle$?
    $\langle stateCondition \rangle$? $\langle variableConditions \rangle$?
    ($\langle goalTemplateList \rangle$ | **\closegoal**)
    $\langle ruleSetMemberships \rangle$?
  }

Taclets describe elementary goal expansion steps. In short, a taclet contains information about 1. when and to which parts of a sequent the taclet can be applied, and 2. in which way the proof is expanded or a proof goal is closed. This information is given by the different parts that make up the body of a taclet. Fig. 4.9 shows the syntax of the taclet parts, which are explained in more detail on the following pages.

Context Assumptions: What Has to Be Present in a Sequent

$\langle contextAssumptions \rangle$ ::= **\assumes** ($\langle schematicSequent \rangle$)

Context assumptions are—together with the **\find** part of a taclet—the means of expressing that a goal modification can only be performed if certain formulae are present in the goal. If a taclet contains an **\assumes** clause, then the taclet may only be applied if the given sequent is part of the goal that is supposed to be modified. Formulae specified as assumptions are not consumed[7] by the application of the taclet, they are instead kept and will also be present in the modified goals.

Examples in Sect. 4.1: close, mpLeft ($\Rightarrow$ Fig. 4.3), applyEq ($\Rightarrow$ Fig. 4.5).

---

[6] Appendix B provides a complete grammar of taclets as they are implemented in KeY and contains more elements than we describe here. The present chapter concentrates on those taclet constructs that are of interest for a KeY user.

[7] It is possible, however, that a taclet is applied on one of its assumptions, i.e., that an assumption is also matched by the **\find** pattern of a taclet. In this situation a taclet application can modify or remove an assumption.

--- KeY Syntax --

⟨*taclet*⟩ ::=
  ⟨*identifier*⟩ {
      ⟨*contextAssumptions*⟩? ⟨*findPattern*⟩?
      ⟨*stateCondition*⟩? ⟨*variableConditions*⟩?
      (⟨*goalTemplateList*⟩ | **\closegoal**)
      ⟨*ruleSetMemberships*⟩?
  }

⟨*contextAssumptions*⟩ ::= **\assumes** (⟨*schematicSequent*⟩)

⟨*findPattern*⟩ ::= **\find** (⟨*schematicExpression*⟩)
⟨*schematicExpression*⟩ ::=
      ⟨*schematicSequent*⟩ | ⟨*schematicFormula*⟩ | ⟨*schematicTerm*⟩

⟨*stateCondition*⟩ ::= **\inSequentState** | **\sameUpdateLevel**

⟨*variableConditions*⟩ ::= **\varcond** (⟨*variableConditionList*⟩)
⟨*variableConditionList*⟩ ::= ⟨*variableCondition*⟩ (**,** ⟨*variableCondition*⟩)∗

⟨*goalTemplateList*⟩ ::= ⟨*goalTemplate*⟩ (**;** ⟨*goalTemplate*⟩)∗
⟨*goalTemplate*⟩ ::=
      ⟨*branchName*⟩?
      (**\replacewith** (⟨*schematicExpression*⟩))?
      (**\add** (⟨*schematicSequent*⟩))?
      (**\addrules** (⟨*taclet*⟩ (**,** ⟨*taclet*⟩)∗))?
⟨*branchName*⟩ ::= ⟨*string*⟩ :

⟨*ruleSetMemberships*⟩ ::= **\heuristics** (⟨*identifierList*⟩)
⟨*identifierList*⟩ ::= ⟨*identifier*⟩ (**,** ⟨*identifier*⟩)∗

-- KeY Syntax ----

Fig. 4.9. The taclet syntax

Find Pattern: To Which Expressions a Taclet Can Be Applied

⟨*findPattern*⟩ ::= **\find** (⟨*schematicExpression*⟩)
⟨*schematicExpression*⟩ ::=
      ⟨*schematicSequent*⟩ | ⟨*schematicFormula*⟩ | ⟨*schematicTerm*⟩

More specifically than just to a goal of a proof, taclets are usually applied to
an occurrence of either a formula or a term within this goal. This occurrence
is called the *focus* of the taclet application and is the only place in the goal
where the taclet can modify an already existing formula (apart from which,
only new formulae can be added to the goal). There are three different kinds
of patterns a taclet can match on:

- A schematic sequent that contains exactly one formula: this either speci-
  fies that the taclet can be applied if the given formula is an element of the
  antecedent, or if it is an element of the succedent, with the formula being
  the focus of the application. It is allowed, however, that the occurrence of
  the formula is preceded by updates (see the section on "State Conditions"
  and Sect. 3.4.3).
- A formula: the focus of the application can be an arbitrary occurrence of
  the given formula (also as subformula) within a goal.
- A term: the focus of the application can be any occurrence of the given
  term within a goal.

Taclets with the last two kinds of \find patterns are commonly referred to
as *rewriting taclets*.

Examples in Sect. 4.1: Virtually all taclets given there.

State Conditions: Where a Taclet Can Be Applied

⟨*stateCondition*⟩ ::= **\inSequentState** | **\sameUpdateLevel**

In a modal logic like JAVA CARD DL, a finer control over where the focus
of a taclet application may be located is needed than is provided by the
different kinds of \find patterns. For rewriting rules it is, for instance, often
necessary to forbid taclet applications within the scope of modal operators
(like program blocks) in order to ensure soundness. There are three different
"modes" that a taclet can have:

- \inSequentState: the most restrictive mode, in which the focus of a
  taclet application must not be located within the scope of *any* modal
  operator (like programs or updates).
- \sameUpdateLevel: this mode is only allowed for rewriting taclets (i.e.,
  there is a \find clause, and the pattern is not a sequent) and allows the
  application focus of a taclet to lie within the scope of updates, but not in
  the scope of other modal operators. The same updates have to occur in
  front of the application focus and the formulae referred to using \assumes,
  \replacewith and \add.

  Examples in Sect. 4.1: applyEq, applyEqAR (⇒ Fig. 4.5).

- Default: the most liberal mode. For rewriting taclets, this means that
  the focus can occur arbitrarily deeply nested and in the scope of any
  modal operator. If the \find pattern of the taclet is a sequent, then the
  application focus may occur below updates, but not in the scope of any
  other operator.

State conditions also affect the formulae that are required or added by
\assumes, \add or \replacewith clauses. Such formulae have to be pre-
ceded by the same updates as the focus of the taclet application (which also

Table 4.7. Matrix of the different taclet modes and the different `\find` patterns. For each combination, it is shown 1. where the focus of the taclet application can be located, and 2. which updates consequently have to occur above the formulae that are matched or added by `\assumes`, `\add` or `\replacewith`.

	`\find` pattern is sequent	`\find` pattern is term or formula	No `\find`
Operators that are allowed above focus			
`\inSequentState`	None	All non-modal operators	*Forbidden combination*
`\sameUpdateLevel`	*Forbidden combination*	All non-modal operators, updates	*Forbidden combination*
Default	Updates	All operators	—
Which updates occur above `\assumes` *and* `\add` *formulae*			
`\inSequentState`	None	None	*Forbidden combination*
`\sameUpdateLevel`	*Forbidden combination*	Same updates as above focus	*Forbidden combination*
Default	Same updates as above focus	None	None
Which updates occur above `\replacewith` *formulae*			
`\inSequentState`	None	None	*Forbidden combination*
`\sameUpdateLevel`	*Forbidden combination*	Same updates as above focus	*Forbidden combination*
Default	Same updates as above focus	Same updates as above focus	None

explains the keyword `\sameUpdateLevel`). The only exception are rewriting taclets in "default" mode, where formulae that are described by `\assumes` or `\add` must *not* be in the scope of updates, whereas there are no restrictions on the location of the focus. The relation between the positions of the different formulae is also shown in Table 4.7.

Variable Conditions: How Schema Variables May Be Instantiated

⟨*variableConditions*⟩ ::= **\varcond** (⟨*variableConditionList*⟩)
⟨*variableConditionList*⟩ ::= ⟨*variableCondition*⟩ (**,** ⟨*variableCondition*⟩)∗

Schema variable conditions have already been introduced in Sect. 4.2.6, together with the concrete syntax for such conditions that is used in KeY. A list of such conditions can be attached to each taclet to control how schema variables are allowed to be instantiated.

Examples in Sect. 4.1: `removeAll` (⇒ Fig. 4.5).

Goal Templates: The Effect of the Taclet Application

$\langle goalTemplateList \rangle$ ::= $\langle goalTemplate \rangle$ (; $\langle goalTemplate \rangle$)∗
$\langle goalTemplate \rangle$ ::=
    $\langle branchName \rangle$?
    (**\replacewith** ($\langle schematicExpression \rangle$))?
    (**\add** ($\langle schematicSequent \rangle$))?
    (**\addrules** ($\langle taclet \rangle$ (, $\langle taclet \rangle$)∗))?
$\langle branchName \rangle$ ::= $\langle string \rangle$:

If the application of a taclet on a certain goal and a certain focus is permitted and is carried out, the *goal templates* of the taclet describe in which way the goal is altered. Generally, the taclet application will first create a number of new proof goals (split the existing proof goal into a number of new goals) and then modify each of the goals according to one of the goal templates. A taclet without goal templates will close a proof goal. As shown above as well as in Sect. 4.1, in this case the keyword \closegoal is written instead of a list of goal templates in order to clarify this behaviour syntactically.

Goal templates are made up of three kinds of operations:

- \replacewith: if a taclet contains a \find clause, then the focus of the taclet application can be replaced with new formulae or terms. \replacewith has to be used in accordance with the kind of the \find pattern: if the pattern is a sequent, then also the argument of the keyword \replacewith has to be a sequent, etc. In contrast to \find patterns, there is no restriction concerning the number of formulae that may turn up in a sequent being argument of \replacewith. It is possible to remove a formula from a sequent by replacing it with an empty sequent, or to replace it with multiple new formulae.
- \add: independently of the kind of the \find pattern, the taclet application can add new formulae to a goal.
- \addrules: a taclet can also create new taclets when being applied. We ignore this feature for the time being and come back to it in Sect. 4.4.7.

Examples in Sect. 4.1: applyEqAR ($\Rightarrow$ Fig. 4.5), saveLeft, hideLeft.

Apart from that, each of the new goals (or branches) can be given a name in order to improve readability of proof trees.

Rule Sets: How Taclets Are Applied Automatically

$\langle ruleSetMemberships \rangle$ ::= **\heuristics** ($\langle identifierList \rangle$)
$\langle identifierList \rangle$ ::= $\langle identifier \rangle$ (, $\langle identifier \rangle$)∗

Each taclet can be declared to be element of a number of rule sets, which in turn are used by the *strategies* in KeY that are responsible for applying rules automatically. Rule sets are intended as an abstraction from the actual taclets and identify taclets that should be treated in the same way. Existing rule sets

in KeY are, amongst many others, `alpha`[8] (for non-splitting elimination of propositional connectives), `beta` (splitting elimination of connectives), and `simplify` (simplification of expressions).

4.4.2 Managing Rules: An Excursion to Taclet Options

It frequently happens that slightly differing versions of a calculus or a theory are to be defined. Examples for this in KeY are 1. the different ways of interpreting JAVA integers ($\Rightarrow$ Chap. 12), which in KeY are realised by different sets of rules for symbolic execution, 2. whether static initialisation of classes should be taken into account during symbolic execution ($\Rightarrow$ Chap. 3), 3. whether dereferencing during symbolic execution (accesses to attributes or array elements) should check for null references ($\Rightarrow$ Chap. 3), or 4. how transactions in JAVA CARD are handled ($\Rightarrow$ Chap. 9).

In order to meet such situations, KeY provides means of disabling or enabling taclets (or other constructs like function symbols) depending on *taclet options* that can be chosen before starting a proof. The usage of taclet options basically consists of three components:

Declaration of Taclet Option Categories

Similarly to schema variables, taclet option categories can be introduced in rule or problem files before defining taclets or symbols. The following declaration, for instance, creates a category `expSideConditions` that has two possible values, `splitting` and `ifThenElse`:

```
─ KeY ─
\optionsDecl { expSideConditions : {splitting, ifThenElse}; }
                                                        ─ KeY ─
```

Conditional Declaration of Rules or Symbols

After their declaration, taclet options can be used to define taclets and symbols that must only occur in proofs or formulae for a particular value of the option. As an example, we define two different versions of the taclet `expSplit` from Fig. 4.2 in the beginning of this chapter. Note, that the two declarations do not clash, even though they are defining two taclets with the same name:

```
─ KeY ─
\rules {
  expSplit (expSideConditions:splitting) {
       \find(exp(a, b)) \sameUpdateLevel
```

---

[8] The names `alpha` and `beta` are common terminology in tableau-style theorem provers.

```
        \replacewith(exp(a, b-c) * exp(a, c));
        \add(==> c >= 0); \add(==> b >= c) };
  expSplit (expSideConditions:ifThenElse) {
        \find(exp(a, b)) \sameUpdateLevel
        \replacewith( \if (c >= 0 & b >= c)
                      \then (exp(a, b-c) * exp(a, c))
                      \else (exp(a, b)) ) };
}
```
——— KeY ——

A complete list of the locations where taclet options can be used is given in
Appendix B.

Definition of the Taclet Option Value

Finally, when starting a proof, the value of taclet options can be chosen,
which determines the versions of taclets that are made available in the proof.
There are two major ways of doing this: values can be set with the keyword
\withOptions in a KeY problem file, or the default values of taclet options
can be chosen in the KeY user interface. An example for the first possibility
is the following line:

—— KeY ——

\withOptions expSideConditions:splitting;
——— KeY ——

It is not possible to change the value of taclet options during the course of a
proof.

4.4.3 Well-Formedness Conditions on Taclets

Not all taclets that can be written using the syntax of Sect. 4.4.1 are mean-
ingful or desirable descriptions of rules. We want to avoid, in particular, rules
whose application could destroy well-formedness of formulae or sequents. In
this section, we thus give a number of additional constraints on taclets that
go beyond the pure taclet syntax. As a note up-front, all of the following is-
sues could also be solved in different ways, possibly leading to a more flexible
taclet language, but experience shows that the requirement to write taclets in
a more explicit way reduces the risk of introducing bugs and unsound taclets.

   In the KeY implementation, non-well-formed taclets are immediately re-
jected and cannot even be loaded.

Sequents Do Not Contain Free Variables

Following Chap. 2, we do not allow sequents of our proofs to contain free
logical variables (in contrast to meta variables ($\Rightarrow$ Sect. 4.3), which are never
bound). Unfortunately, this is a property that can easily be destroyed by
incorrect taclets:

— Taclet —

```
illegalTac1 { \find(==> \forall va; p(va))
             \replacewith(==> p(va)) };
illegalTac2 { \find(==> \forall va; phi)
             \replacewith(==> phi) };
```

— Taclet —

In both examples, the taclets remove quantifiers and possibly inject free variables into a sequent: 1. schema variables of kind \variables could occur free in clauses \add or \replacewith, or 2. a logical variable $\iota(\texttt{va})$ could occur free in the concrete formula $\iota(\texttt{phi})$ that a schema variable phi represents, and after removing the quantifier the variable would be free in the sequent (the same can happen with schema variables for terms). We will rule out both taclets by imposing suitable constraints.

In order to handle taclets like `illegalTac1`, we simply forbid taclets containing \replacewith or \add clauses with unbound schema variables va:

Definition 4.24. *An occurrence of a schema variable* va *of kind* \variables *in a schematic expression is called* bound *if it is in the scope of a quantifier* $\forall$va. *,* $\exists$va. *or a substitution* $\{\backslash subst \text{ sv}; t\}$*, and if it is not itself part of a binder (like* $\forall$va. *).*

Requirement 4.25 (Variables are bound). *All occurrences of a schema variable of kind* \variables *in* \find*,* \assumes*,* \replacewith *or* \add *clauses are either part of a binder or are bound.*

Note 4.26. It would, in principle, be safe to allow schema variables of kind \variables to occur free in \find or \assumes. This would only lead to useless taclets that are never applicable (provided that sequents are not already ill-formed and contain free logical variables).

It is less obvious how taclet `illegalTac2` should be taken care of. According to Sect. 4.2, the schema variable phi can stand for arbitrary formulae containing arbitrary free variables, but as the example shows this is too liberal. Whenever a variable x occurs free in a formula $\iota(\texttt{phi})$, the formula also has to be in the scope of a binder of x. The binder could either occur explicitly in the taclet—like a quantifier $\forall$va. —or for a rewriting taclet it could be part of the context in which the taclet is applied. An example for the latter possibility is shown in Sect. 4.2.4.

We go for a rigorous solution of the problem and require the author of `illegalTac2` to state his or her intention more clearly. The first half of the solution is given in the following definition, where we require that the variables that are explicitly bound in the taclet above occurrences of phi are consistent. The second part is Def. 4.32, describing which logical variables we allow to occur free in a formula $\iota(\texttt{phi})$.

Requirement 4.27 (Unique Context Variables). *Suppose that t is a taclet, that* `sv` *is a schema variable of kind* `\term` *or* `\formula`*, and that* `va` *is a schema variable of kind* `\variables`*. If an occurrence of* `sv` *in* `\find`*,* `\assumes`*,* `\replacewith` *or* `\add` *clauses of t is in the scope of a binder of* `va` *(which could be* $\forall$`va.`*,* $\exists$`va.` *or* $\{\backslash subst$ `sv`*;* $u\}$*), then*

- all *occurrences of* `sv` *in t are in the scope of a binder of* `va`*, or*
- *t has a variable condition* `\notFreeIn(va, sv)`*.*

The variable `va` *is called a* context variable *of* `sv` *in t. More formally, the set of context variables of* `sv` *in t is defined as*

$$\Pi_t(\text{sv}) = \{\text{va} \mid \text{va } \textit{is of kind } \texttt{\textbackslash variables}, \text{sv } \textit{is in the scope of } \text{va}\}$$
$$\setminus \{\text{va} \mid t \textit{ has variable condition } \texttt{\textbackslash notFreeIn(va, sv)}\}$$

Correct versions of the taclet shown above are thus:

—— Taclet ——————————————————————————————
```
legalTac2a { \find(==> \forall va; phi)
             \replacewith(==> {\subst va; t} phi) };
legalTac2b { \find(==> \forall va; phi)
             \varcond(\notFreeIn(va, phi))
             \replacewith(==> phi) };
```
———————————————————————————————— Taclet ——

The context of `phi` in these two taclets, i.e., the sets of variables that are bound for all occurrences of `phi` in the taclets, is $\Pi_{\texttt{legalTac2a}}(\text{phi}) = \{\text{va}\}$ and $\Pi_{\texttt{legalTac2b}}(\text{phi}) = \emptyset$.

Note 4.28. The KeY implementation contains a further well-formedness condition: schema variables of kind `\variables` are allowed to be bound at most once in the `\find` and `\assumes` clauses of a taclet (together). This is not a severe restriction, because it is always possible to apply bound renaming for ensuring that variables are only bound in one place, without changing the intended meaning of a taclet. For the representation of taclets in this chapter, however, it is not necessary to enforce unique binding of variables.

4.4.4 Implicit Bound Renaming and Avoidance of Collisions

As already seen in Sect. 4.2.4 about substitutions, the actual identity of bound variables is not important for the meaning of a formula. If a variable $z \notin fv(\phi)$ does not occur free in ϕ, then formulae $\forall x.\, \phi$ and $\forall z.\, [x/z](\phi)$ will be equivalent. Although it is not strictly necessary for achieving completeness,[9]

---

[9] The calculus in Chap. 2 is complete also without rules for bound renaming or comparing formulae modulo bound renaming. This shows that it is, in principle, sufficient to provide rules for *eliminating* quantifiers.

from a practical point of view it is desirable that the applicability of taclets does not depend on which variables are bound in formulae. We would like to treat sequents like

$$\forall x.\, p(x) \Longrightarrow \forall x.\, p(x)$$
$$\forall x.\, p(x) \Longrightarrow \forall y.\, p(y)$$

in the same way, in particular a taclet like `close` ($\Rightarrow$ Sect. 4.1) should be applicable to one of the sequents if and only if it is applicable to the other sequent. For the second sequent, this would mean that the schema variable `phi` represents two different formulae, which does obviously not work without further measures.

The most common (theoretic) standpoint is to allow implicit renaming steps whenever they are necessary for applying rules. We follow this approach in the scope of this chapter, and in the definitions on the following pages we will only formulate conditions on (bound) variables that possibly have to be established by implicit renaming. In this setting, an application of `close` to the second sequent would be

$$\frac{\overline{\forall y.\, p(y) \Longrightarrow \forall y.\, p(y)}}{\forall x.\, p(x) \Longrightarrow \forall y.\, p(y)} \begin{array}{l} \texttt{close} \\[2pt] (\textsf{rename}) \end{array}$$

where the first step (`rename`) is performed implicitly for preparing the sequent for the actual taclet application. Likewise, in both formulae the variables x (or any other variable) could have been introduced. Renaming steps are often not shown at all in proofs.

Note 4.29. KeY applies bound renaming, whenever it becomes necessary, as part of its taclet application mechanism. As renaming steps are not shown in the proof tree, the user will in most cases not even notice that a renaming has occurred.

Collisions

We have already met the problem of *collisions* in Sect. 4.2.4. A similar phenomenon occurs when applying and evaluating taclets, also here it is possible that the place where a variable is bound changes unintentionally. The reason is that distinct schema variables of kind `\variables` can happen to be instantiated with *the same* logical variable. If an "artificial," but in principle reasonable, taclet for eliminating universal quantifiers in the antecedent

```
── Taclet ──────────────────────────────────────
allExLeft {
    \find (\forall x; \exists y; phi ==>)
    \varcond (\notFreeIn(y, t))
    \add (\exists y; {\subst x; t} phi ==>)   };
──────────────────────────────────── Taclet ──
```

was applied naively, we would have to consider the taclet as unsound. The taclet commutes binders, so that the location where variables are bound can change:

$$\frac{\dfrac{\forall x.\,\exists x.\,x \doteq 0,\ \exists x.\,1 \doteq 0 \Longrightarrow}{\forall x.\,\exists x.\,x \doteq 0,\ \exists x.\,[x/1](x \doteq 0) \Longrightarrow}}{\forall x.\,\exists x.\,x \doteq 0 \Longrightarrow}$$

Here, the lower sequent is not valid (because it only contains a valid formula in the antecedent), whereas the upper one is valid (the assumption $\exists x.\,1 \doteq 0$ is wrong). This means that the application of the taclet has to be considered unsound. In order to make the development of sound taclets easier (feasible), we forbid such taclet applications and demand that the following condition is satisfied when applying taclets (we will refer to this condition when defining taclet applications):

Definition 4.30. *Suppose that t is a taclet and that ι is an instantiation of the variables of t. Then ι has distinct bound variables concerning t if all logical variables $\iota(\mathtt{va})$ represented by schema variables $\mathtt{va}$ of kind* `\variables` *in t are distinct.*

As illustrated in the first part of this section, we assume that the conditions of the previous definition are implicitly established by suitable variable renaming when applying a taclet. In rather special cases, it can also be necessary to duplicate formulae in order to establish distinctness. As an example, we can imagine a taclet that works with terms representing surjective functions:

─── Taclet ───────────────────────────────────────

```
surjectivity {
    \assumes (\forall x; \exists y; x = t ==>)
    \find (\exists z; phi) \sameUpdateLevel
    \varcond (\notFreeIn(x, t), \notFreeIn(y, phi))
    \replacewith (\exists y; {\subst z; t} phi)
};
```

─────────────────────────────────────── Taclet ───

Provided that a term $\mathtt{t}$—in which a variable $\mathtt{y}$ can occur free—describes a surjective mapping, we can use it to rewrite quantified formulae:

$$\frac{\forall a.\,\exists b.\,a \doteq b+1 \Longrightarrow \exists b.\,b+1-1 \doteq 3}{\forall a.\,\exists b.\,a \doteq b+1 \Longrightarrow \exists c.\,c-1 \doteq 3}\ \ \text{surjectivity}$$

In a more pathological application, however, the taclet can be used to modify the quantified formula of the antecedent itself:

$$\frac{\forall a.\,\exists b.\,a \doteq b+1+1 \Longrightarrow}{\forall a.\,\exists b.\,a \doteq b+1 \Longrightarrow}\ \ \text{surjectivity}$$

Strictly speaking, this transformation is only possible if the formula is first duplicated and the variable b is renamed to a new variable b'. The variable condition `\notFreeIn(y, phi)` would otherwise be violated: `phi` becomes instantiated with $a \doteq b + 1$ and `y` with the variable b, which contradicts the variable condition. Hence, a detailed proof tree showing the taclet application looks as follows:

$$
\cfrac{
\cfrac{
\cfrac{
\cfrac{\forall a.\,\exists b.\, a \doteq b + 1 + 1 \Longrightarrow}{\forall a.\,\exists b.\, a \doteq b + 1,\ \forall a.\,\exists b.\, a \doteq b + 1 + 1 \Longrightarrow}\ (\mathsf{weak.})
}{\forall a.\,\exists b.\, a \doteq b + 1,\ \forall a.\,\exists b'.\, a \doteq b' + 1 \Longrightarrow}\ \mathsf{surj.}
}{\forall a.\,\exists b.\, a \doteq b + 1 \Longrightarrow}\ (\mathsf{rename})
}
$$

In the KeY implementation, the steps (rename) and (weaken) would be carried out automatically and not be shown in the proof tree. The example illustrates that it can—in rare cases—be necessary to duplicate formulae and apply renaming before the application of a taclet is possible.

Note 4.31. The condition of Def. 4.30 is sufficient for preventing collisions, but is usually more defensive than necessary. In the KeY implementation, more precise (and complicated) conditions are used that often avoid variable renaming or duplicating formulae.

4.4.5 Applicability of Taclets

This section describes when the application of a taclet is possible. An informal account of this was already given in Sect. 4.4.1, where the different clauses of a taclet are introduced, and is now accompanied with a more rigorous treatment. In order to apply a taclet on a proof goal, several parameters have to be provided:

- If the taclet contains schema variables of generic types ($\Rightarrow$ Sect. 4.2.7), then these types first have to be mapped to concrete types.
- If the taclet contains schema variables, then one has to give an instantiation ($\Rightarrow$ Sect. 4.2.3) that describes how to replace the variables with concrete expressions.
- If the taclet contains a `\find` clause ($\Rightarrow$ Sect. 4.4.1), then it is necessary to select a focus of the taclet application within the goal in question.

Not all values that can be chosen for these unknowns are meaningful and should be admitted. It is obvious that taclet application should not be allowed if, e.g., variable conditions are violated, but there are a number of further and more subtle requirements.

Free Variables in Instantiations of Schema Variables

The examples in the previous sections show that instantiations of schema variables for terms or formulae should be allowed to contain certain logical

variables free. At the same time, it has to be ensured that no free variables are introduced in sequents. In taclet `legalTac2a` ($\Rightarrow$ Sect. 4.4.3), for instance, it should be possible to instantiate variable `va` with x and `phi` with $p(x)$ (where x occurs free), so that the taclet matches on a formula $\forall x.\, p(x)$. The tool for deciding about the free variables that are permitted will be the *context variables* of a schema variable (Requirement 4.27), which are exactly those variables that are guaranteed to be bound (in the taclet) for each occurrence of a schema variable.

For certain rewriting taclets, it is desirable to allow further free variables that are *not* context variables of a schema variable. An example is the taclet `zeroRight` ($\Rightarrow$ Sect. 4.1), which we also would like to apply to a formula like $\forall x.\, p(x + 0)$. The variable `intTerm` does not have any context variables, however, as there are no variables bound at all in the taclet. Situations like this are taken into account as well by the next definition.

Definition 4.32. *Suppose that t is a taclet, that ι is an instantiation of the variables of t and that <u>focus</u> is a potential application focus of t, where we write <u>focus</u> for an occurrence of the formula or term focus, i.e., <u>focus</u> describes not only an expression but also a location within a sequent. We choose <u>focus</u> $= \perp$ if t does not have a* `\find` *clause. We say that ι respects variable contexts concerning <u>focus</u> if, for every schema variable* `sv` *of t of kind* `\term` *or* `\formula` *and all free variables $x \in fv(\iota(\mathtt{sv}))$,*

- *there is a schema variable* `va` $\in \Pi_t(\mathtt{sv})$ *with $\iota(\mathtt{va}) = x$, or*
- *t contains at most one* `\replacewith` *clause,* `sv` *turns up only in* `\find`, `\replacewith` *or* `\varcond` *clauses of t, and x is bound above <u>focus</u>.*

The second item makes it possible to apply taclet `zeroRight` to a formula $\forall x.\, p(x + 0)$. According to Table 4.7, it only applies to rewriting taclets, because for other taclets there will never be any variables bound above <u>focus</u>.

Note 4.33. The requirement of the second item—there is not more than one `\replacewith` clause—is added because quantifiers or other binders do in general not distribute over conjunctions. It would hardly be possible to formulate sound rewriting taclets with more than one `\replacewith` clause if free variables were allowed to turn up. An example is the (artificial, but not unreasonable) taclet performing a case distinction on a term of type boolean

––– Taclet ––

```
splitBool { \find (b) \replacewith (TRUE);
                      \replacewith (FALSE) };
```

––– Taclet –––

which could be used in the following unsound way:

$$\frac{\Longrightarrow \exists x : \text{boolean. TRUE} \mathrel{!\dot{=}} x \qquad \Longrightarrow \exists x : \text{boolean. FALSE} \mathrel{!\dot{=}} x}{\Longrightarrow \exists x : \text{boolean.}\, x \mathrel{!\dot{=}} x}$$

Permitted Taclet Applications

We can now give a complete definition of when we consider a taclet, given a sequent and all necessary parameters, as applicable:

Definition 4.34 (Matching Instantiation). *Suppose that t is a taclet over a generic type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \mathcal{T}_g, \sqsubseteq, \mathcal{R}_g)$ and a set SV of schema variables. A* matching instantiation *of t is a tuple $(\iota_t, \iota, \mathcal{U}, \Gamma \Longrightarrow \Delta, \underline{focus})$ consisting of*

- *a type instantiation ι_t,*
- *a complete instantiation ι of the schema variables of t (apart from those variables that only occur within* \addrules *clauses),*
- *an update $\mathcal{U}$ describing the context of the taclet application ($\mathcal{U}$ can be empty),*
- *a sequent $\Gamma \Longrightarrow \Delta$ to which the taclet is supposed to be applied, and*
- *an application focus $\underline{focus}$ within $\Gamma \Longrightarrow \Delta$ that is supposed to be modified (we write $\underline{focus} = \bot$ if t does not have a* \find *clause)*

that satisfies the following conditions:

1. *the pair (ι_t, ι) is an admissible instantiation of SV under generic types ($\Rightarrow$ Def. 4.22),*
2. *ι satisfies all variable conditions of taclet t (referring to Table 4.6),*
3. *ι respects the variable context of t concerning $\underline{focus}$ ($\Rightarrow$ Def. 4.32),*
4. *ι has distinct bound variables concerning t ($\Rightarrow$ Def. 4.30),*
5. *if t has a* \find *clause, then the position of $\underline{focus}$ is consistent with the state conditions of t (Table 4.7),*
6. *$\mathcal{U}$ is derived from $\underline{focus}$ according to the middle part "Which updates have to occur above* \assumes *and* \add *formulae" of Table 4.7 (for $\underline{focus} = \bot$ and the fields "forbidden combination" we choose the empty update* skip*),*
7. *for each formula ϕ of an* \assumes *clause of t, $\Gamma \Longrightarrow \Delta$ contains a corresponding formula $\mathcal{U}\iota(\phi)$ (on the correct side),*
8. *if t has a clause* \find(f)*, where f is a formula or a term, then $\iota(f) = \underline{focus}$ (the* \find *pattern has to match the focus of the application),*
9. *if t has a clause* \find(f)*, where f is a sequent containing a single formula ϕ, then $\iota(\phi) = \underline{focus}$ and the formulae ϕ and $\underline{focus}$ occur on the same sequent side (both antecedent or both succedent).*

Example 4.35. We show how this definition applies to the taclet instAll ($\Rightarrow$ Fig. 4.10), which is a variant of allLeft and allows to select the term that is supposed to be substituted as focus. We can apply the taclet to the sequent

$$\Gamma \Longrightarrow \Delta \;\; = \;\; \forall o.\, f(o) \doteq o.a, \; f(self) \doteq 1 \Longrightarrow \{i := 2\}(\underline{self}.a \doteq 1)$$

```
─── KeY ───────────────────────────────────────────────────
\sorts {
  \generic G;
}
\schemaVariables {
  \formula phi;      \variables G x;      \term G s;
}
\rules {
  instAll { \assumes (\forall x; phi ==>) \find (s)
          \add ({\subst x; s}phi ==>) };
}
───────────────────────────────────────────── KeY ───
```

Fig. 4.10. The taclet described in Example 4.35.

where the application focus is underlined, the constant *self* and the logical variable o have type A and $f : A \rightarrow \texttt{integer}$ is a function. The remaining components of the matching instantiation are:

- the type instantiation $\iota_t = \{G \mapsto A\}$,
- the instantiation $\iota = \{\texttt{phi} \mapsto (f(o) \doteq o.a),\ \texttt{x} \mapsto o,\ \texttt{s} \mapsto self\}$,
- the (effect-less) update $\mathcal{U} = \texttt{skip}$.

That the instantiation is indeed matching can be observed as follows:

1. We have $\iota_t(\texttt{\textbackslash variables } G) = \texttt{\textbackslash variables } A$, $\iota_t(\texttt{\textbackslash term } G) = \texttt{\textbackslash term } A$. Because of the types of *self* and o, the pair (ι_t, ι) is an admissible instantiation.
2. There are no variable conditions.
3. The variable contexts of $\texttt{instAll}$ are the sets $\Pi_{\texttt{instAll}}(\texttt{phi}) = \{\texttt{x}\}$ and $\Pi_{\texttt{instAll}}(\texttt{s}) = \emptyset$, they are respected by ι as $fv(\iota(\texttt{phi})) = \{o\} = \{\iota(\texttt{x})\}$ and $\iota(\texttt{s})$ does not contain variables.
4. There is only one schema variable of kind $\texttt{\textbackslash variables}$. Hence, ι has distinct bound variables.
5. $\texttt{instAll}$ has no explicit state conditions, and thus all operators are allowed above the focus *self*.
6. According to Table 4.7, $\mathcal{U}$ has to be the empty update $\texttt{skip}$. Note that this is the case even though the application focus is in the scope of an update.
7. The $\texttt{\textbackslash assumes}$ clause contains only one formula, which is correctly mapped to one of the formulae of Γ: $\iota(\forall x.\, \texttt{phi}) = (\forall o.\, f(o) \doteq o.a)$.
8. The $\texttt{\textbackslash find}$ expression is correctly mapped to the term of the application focus: $\iota(\texttt{s}) = self$.
9. (Does not apply.)

Note 4.36. In the KeY implementation, when applying a taclet like $\texttt{instAll}$ one will usually not specify the instantiations of $\texttt{x}$ and $\texttt{phi}$ explicitly, but

KeY will rather search for formulas of the antecedent that could be matched by $\forall$x.phi and derive possible choices for x and phi automatically. If more than one possible formula is found in the antecedent, a dialog will be shown in which the user can choose the formula that should be taken. This is explained in more detail in Chap. 10.

4.4.6 The Effect of a Taclet

Applying a taclet to a goal and a focus will carry out the modification steps that are described by the goal templates of the taclet. Each goal template can alter the focus the taclet is applied to (\replacewith), add further formulae to a goal (\add) and make further taclets available (\addrules). In this section, we concentrate on the first two kinds of effects and postpone a discussion of the latter kind until Sect. 4.4.7.

Definition 4.37 (Applying a Goal Template). *Suppose that a matching instantiation $(\iota_t, \iota, \mathcal{U}, \Gamma \Longrightarrow \Delta, \underline{focus})$ of a taclet t is given. One goal template is applied on $\Gamma \Longrightarrow \Delta$ by performing the following steps (in the given order):*

1. *If the goal template has a clause \replacewith$(rw)$, where rw is a formula or a term, then $\underline{focus}$ is replaced with $\iota(rw)$. If rw is a term and the type A_{new} of $\iota(rw)$ is not a subtype of the type A_{old} of $\underline{focus}$ $(A_{new} \not\sqsubseteq A_{old})$, then $\underline{focus}$ is replaced with $(A_{old})\iota(rw)$ instead of $\iota(rw)$ (a cast has to be introduced to prevent ill-formed terms).*
2. *If the goal template has a clause \replacewith$(rw)$, where rw is a sequent, then the formula containing $\underline{focus}$ is removed from $\Gamma \Longrightarrow \Delta$, and for each formula ϕ in rw the formula $\mathcal{U}\iota(\phi)$ is added (on the correct side).*
3. *If the goal template has a clause \add$(add)$, then for each formula ϕ in add the formula $\mathcal{U}\iota(\phi)$ is added (on the correct side).*

The complete application of a taclet involves duplicating a proof goal and applying each of its goal templates. In case of taclets that do not have any goal templates, this actually closes the proof goal.

Definition 4.38 (Applying a Taclet). *Suppose that a matching instantiation $(\iota_t, \iota, \mathcal{U}, \Gamma \Longrightarrow \Delta, \underline{focus})$ of a taclet t is given, where $\Gamma \Longrightarrow \Delta$ is the sequent of one proof goal g. Carrying out the application of t means performing the following steps (in the given order):*

1. *n new proof goals with sequent $\Gamma \Longrightarrow \Delta$ are created as children of g, where n is the number of goal templates of t. For $n = 0$ the goal g is closed.*
2. *Each of the goal templates of t is applied to one of the new goals, given the matching instantiation $(\iota_t, \iota, \mathcal{U}, \Gamma \Longrightarrow \Delta, \underline{focus})$.*

Example 4.39. We continue Example 4.35 and apply instAll with the matching instantiation shown there. The taclet instAll has only a single goal template, so the first step is to duplicate the initial sequent:

$$\frac{\forall o.\, f(o) \doteq o.a, \quad f(self) \doteq 1 \;\Longrightarrow\; \{i := 2\}(self.a \doteq 1)}{\forall o.\, f(o) \doteq o.a, \quad f(self) \doteq 1 \;\Longrightarrow\; \{i := 2\}(self.a \doteq 1)}$$

Applying the goal template here only means to carry out the `\add` clause. The formula that is to be added is (the update can be left out immediately)

$$\mathcal{U}\iota(\{\texttt{\textbackslash subst x; s}\}\texttt{phi}) \;=\; \{\texttt{skip}\}([o/self](f(o) \doteq o.a)) \;=\; f(self) \doteq self.a$$

Finally, the rule application yields

$$\frac{\forall o.\, f(o) \doteq o.a, \quad f(self) \doteq 1, \quad f(self) \doteq self.a \;\Longrightarrow\; \{i := 2\}(self.a \doteq 1)}{\forall o.\, f(o) \doteq o.a, \quad f(self) \doteq 1 \;\Longrightarrow\; \{i := 2\}(self.a \doteq 1)}$$

4.4.7 Taclets in Context: Taclet-Based Proofs

So far, we have introduced and defined the meaning of taclets as modification steps that can be applied to a proof tree. Taclets can, however, also modify the rule base that is used to construct a proof. Probably the best example for this feature is taclet `applyEqAR` ($\Rightarrow$ Sect. 4.1) for rewriting terms in the presence of an equation in an antecedent. Applying the taclet to an equation that can be matched by `t1` $\doteq$ `t2` results in a new taclet `rewrWithEq` that replaces the term matched by `t1` with the term matched by `t2`. It is clear, however, that the taclet `rewrWithEq` must not be added to the rule base "globally," as it is only correct for those sequents that actually contain an equation $f(a) \doteq b$. `\addrules` is only meaningful if we have a notion of "local" rules that only exist in certain parts of a proof tree, and that are not available elsewhere. To realise such a notion, taclets will get a character that is similar to the formulae of a sequent: to each sequent, a set of taclets is attached that are available for application. If a proof is expanded by adding children to a parent goal, then these goals will inherit all rules from the parent goal, but will possibly also get further rules that were not present in the parent goal (like `rewrWithEq`).

Partially Instantiated Taclets

What is attached to sequents are not only the actual taclets, but also further information that is necessary to restrict the applicability of taclets in the right way. What is actually added when applying `applyEqAR` to the equation $f(a) \doteq b$ is the taclet `rewrWithEq`

——— Taclet ———
```
rewrWithEq { \find (t) \sameUpdateLevel \replacewith (t2) }
```
——————————————————— Taclet ———

together with the following components:

- the type instantiation $\iota_t = \{G \mapsto A\}$ (where A is the type of $f(a)$),
- the instantiation $\iota = \{\iota(\texttt{t1}) \mapsto f(a), \;\; \iota(\texttt{t2}) \mapsto b\}$, and
- the update $\mathcal{U} = \texttt{skip}$.

The two instantiation functions have to be considered as partial in this setting, because an inner taclet like `rewrWithEq` can contain schema variables or generic types that are not part of the parent taclet and, thus, are not yet determined.

Definition 4.40. *A* partially instantiated taclet *is a tuple* $(t, \iota_t, \iota, \mathcal{U})$ *consisting of*

- *a taclet t,*
- *a (partial) type instantiation ι_t,*
- *a (partial) schema variable instantiation ι, and*
- *an update $\mathcal{U}$ describing the context of the taclet application ($\mathcal{U}$ can be empty or $\perp$).*

When applying a partially instantiated taclet, the information already given has to be extended so that the application is possible.

Definition 4.41. *A* matching instantiation *of a partially instantiated taclet* $(t, \iota'_t, \iota', \mathcal{U}')$ *is a tuple* $(\iota_t, \iota, \mathcal{U}, \Gamma \Longrightarrow \Delta, \underline{focus})$ *such that*

- $(\iota_t, \iota, \mathcal{U}, \Gamma \Longrightarrow \Delta, \underline{focus})$ *is a matching instantiation of t ($\Rightarrow$ Def. 4.34),*
- ι_t *is an extension of ι'_t (as function),*
- ι *is an extension of ι' (as function),*
- *if $\mathcal{U}' \neq \perp$ then $\mathcal{U} = \mathcal{U}'$.*

Taclet-Based Proofs

In contrast to a proof tree in an ordinary sequent calculus ($\Rightarrow$ Sect. 2.5), to each node of a taclet-based proof tree a set of partially instantiated taclets is attached. The root of the tree is given the base set of rules, which are partially instantiated taclets $(t, \perp, \perp, \perp)$, i.e., the instantiation mappings are completely undefined, and the update context of taclet applications is not yet determined. During the construction of the proof tree, further taclets can be added to proof nodes below the root using the `\addrules` clause.

Taking this into account, we can extend the definitions of the effect of taclets in the previous section.

Definition 4.42 (Continuation of Def. 4.37).

4. If the goal template has a clause `\addrules`(rules), then for each taclet r in rules the partially instantiated taclet $(r, \iota'_t, \iota', \mathcal{U})$ is added, where
- ι' *is the restriction of ι to the schema variables of r,*
- ι'_t *is the restriction of the mapping ι_t to the types that occur within the kinds k of schema variables `sv` with $\iota(\text{sv}) \neq \perp$.*

Definition 4.43 (Continuation of Def. 4.38).

1b. Each of the new proof goals is given the same set of partially instantiated taclets as the parent goal.

4.5 Reasoning About the Soundness of Taclets

Taclets are a general language for describing proof modification steps. In order to ensure that the rules that are implemented using taclets are correct, we can consider the definitions of the previous sections and try to derive that no incorrect proofs can be constructed using the taclets. This promises to be tedious work, however, and is for a larger number of taclets virtually useless if the reasoning is performed informally: we are bound to make mistakes.

For treating the correctness of taclets in a more systematic way, we would rather like to have some *calculus* for reasoning about soundness of taclets. This is provided in this section for some of the features of taclets.[10] Note, that the following two translation steps correspond to the two main ingredients of taclets in the end of Sect. 4.1 (page 191).

- We describe a translation of taclets into formulae (the *meaning formulae* of taclets), such that a taclet is sound if the formula is valid. This translation captures the semantics of the different clauses that a taclet can consist of. Meaning formulae do, however, still contain schema variables, which means that for proving their validity methods like induction over terms or programs are necessary.
- A second transformation handles the elimination of schema variables in meaning formulae, which is achieved by replacing schema variables with Skolem terms or formulae. The result is a formula of first order logic or dynamic logic (depending on the expressions that turned up in the taclet), such that the original formula is valid if the derived formula is valid. This step is only possible for certain kinds of schema variables; handling schema variables for program entities, in particular, can be difficult or impossible [Bubel et al., 2004]. Depending on the kind of the schema variable, it can happen that only an incomplete transformation is possible, in the sense that the resulting formula can be invalid although the meaning formula actually is valid and the taclet is sound.

The two steps can be employed in different settings:

- The first step can be carried out, and one can reason about the resulting formula using an appropriate proof assistant, e.g., based on higher-order logic. For taclets that contain JAVA CARD programs, this will usually require to have a formalisation of the JAVA CARD semantics for the chosen proof assistant. In this context, some of the assignment rules for JAVA CARD ($\Rightarrow$ Sect. 3.6) have been proven correct by Trentelman [2005] using the Isabelle/HOL proof assistant [Nipkow et al., 2002] and the Bali formalisation of JAVA [Oheimb and Nipkow, 1999]. Ahrendt et al. [2005b] follow a similar strategy and prove the correctness of certain rules for the symbolic execution of JAVA referring to an existing JAVA semantics in

---

[10] The issue of meta variables ($\Rightarrow$ Sect. 4.3), for instance, is not taken into account on the next pages.

rewriting logic [Meseguer and Rosu, 2004]. A more detailed description is given in the sidebar on page 109 in Chap. 3.

- Both steps can be carried out, which opens up for a wider spectrum of provers or proof assistants that the resulting formulae can be tackled with. The formulae can in particular be treated by a prover for dynamic logic itself, such as KeY. This is applicable for *lemma* rules, i.e., for taclets which can be proven sound referring to other—more basic—taclets. The complete translation from taclets to formulae of dynamic logic can automatically be performed by KeY and makes it possible to write and use lemmas whenever this is useful, see [Bubel et al., 2004].

In the following, we first give a recapitulation about when rules of a sequent calculus are sound, and then show how this notion can be applied to the taclet concept. It has to be noted, however, that although reading the following pages in detail is not necessary for defining new taclets, it might help to understand what happens when lemmas are loaded in KeY.

4.5.1 Soundness in Sequent Calculi

In the whole section we write $(\Gamma \Longrightarrow \Delta)^* := \bigwedge \Gamma \rightarrow \bigvee \Delta$ for the formula that expresses the meaning of the sequent $\Gamma \Longrightarrow \Delta$. This formula is, in particular:

$$(\Longrightarrow \phi)^* = \phi \ , \qquad (\phi \Longrightarrow)^* = \neg \phi \ .$$

By the validity of a sequent we consequently mean the validity of the formula $(\Gamma \Longrightarrow \Delta)^*$.

A further notation that we are going to use is the following "union" of two sequents:

$$\left(\Gamma_1 \Longrightarrow \Delta_1\right) \cup \left(\Gamma_2 \Longrightarrow \Delta_2\right) \quad := \quad \Gamma_1, \Gamma_2 \Longrightarrow \Delta_1, \Delta_2 \ .$$

Because antecedents and succedents are defined to be sets, duplicate formulae will automatically disappear in the sequent on the right side.

Definition 4.44 (Soundness). *A sequent calculus C is* sound *if only valid sequents are derivable in C, i.e., if the root $\Gamma \Longrightarrow \Delta$ of a closed proof tree is valid.*

This general definition does not refer to particular rules of a calculus C, but treats C as an abstract mechanism that determines a set of derivable sequents. For practical purposes, however, it is advantageous to formulate soundness in a more "local" fashion and to talk about the rules (or taclets implementing the rules) of C. Such a local criterion can already be given when considering rules in a very abstract sense: a rule R can be considered as an arbitrary (but at least semi-decidable) relation between tuples of sequents (the premisses) and single sequents (the conclusions). Consequently, $(\langle P_1, \ldots, P_k \rangle, Q) \in R$ means that the rule R can be applied in an expansion step

$$\frac{P_1 \quad \cdots \quad P_k}{Q}$$

The following lemma relates the notion of soundness of a calculus with rules:

Lemma 4.45. *A calculus C is sound, if for each rule $R \in C$ and all tuples $(\langle P_1, \ldots, P_k \rangle, Q) \in R$ the following implication holds:*

$$\text{if } P_1, \ldots, P_k \text{ are valid, then } Q \text{ is valid.} \tag{4.1}$$

If condition (4.1) holds for all tuples $(\langle P_1, \ldots, P_k \rangle, Q) \in R$ of a rule R, then this rule is also called *sound*.

4.5.2 A Basic Version of Meaning Formulae

In our case, the rules of a calculus C are defined through taclets t over a set SV of schema variables, and within the next paragraphs we discuss how Lem. 4.45 can be applied considering such a rule. For a start, we consider a taclet whose \find pattern is a sequent and that has the following basic shape:

───── Taclet ─────

```
t1 { \assumes(assum) \find(findSeq) \inSequentState
     \replacewith(rw1) \add(add1);
     ...
     \replacewith(rwk) \add(addk) };
```

───────────────────────────── Taclet ─────

Using text-book notation for rules in sequent calculi (as in Chap. 2), the taclet describes the rule

$$\frac{\texttt{rw1} \cup \texttt{add1} \cup \texttt{assum} \cup (\Gamma \Longrightarrow \Delta) \quad \cdots \quad \texttt{rwk} \cup \texttt{addk} \cup \texttt{assum} \cup (\Gamma \Longrightarrow \Delta)}{\texttt{findSeq} \cup \texttt{assum} \cup (\Gamma \Longrightarrow \Delta)}$$

In order to apply Lem. 4.45, it is then necessary to show implication (4.1) for all possible applications of the rule, i.e., essentially for all possible ways the schema variables that now turn up in the sequents can be instantiated. If ι is such a possible instantiation ($\Rightarrow$ Def. 4.34), and if $\Gamma \Longrightarrow \Delta$ is an arbitrary sequent, then

$$\begin{aligned} P_i &= \iota(\texttt{rw}i \cup \texttt{add}i \cup \texttt{assum}) \cup (\Gamma \Longrightarrow \Delta) \quad (i = 1, \ldots, k), \\ Q &= \iota(\texttt{findSeq} \cup \texttt{assum}) \cup (\Gamma \Longrightarrow \Delta) \end{aligned} \tag{4.2}$$

Implication (4.1)—which is a *global* soundness criterion—can be replaced with a *local* implication:

$$\left(P_1^* \mathrel{\&} \ldots \mathrel{\&} P_k^* \mathrel{->} Q^* \right) \text{ is valid.} \tag{4.3}$$

Inserting the sequents (4.2) extracted from taclet `t1` into (4.3) leads to a formula whose validity is sufficient for implication (4.1):

$$P_1^* \& \ldots \& P_k^* \rightarrow Q^* = \bigwedge_{i=1}^{k} \left(\iota(\texttt{rwi} \cup \texttt{addi} \cup \texttt{assum}) \cup (\varGamma \Longrightarrow \varDelta) \right)^* \quad (4.4)$$
$$\rightarrow \left(\iota(\texttt{findSeq} \cup \texttt{assum}) \cup (\varGamma \Longrightarrow \varDelta) \right)^*$$

In order to simplify the right side of Eq. (4.4), we can now make use of the fact that ι distributes through all propositional connectives ($\rightarrow$, &, |, etc.) and also through the union of sequents. Furthermore, there is a simple law describing the relation between * and the union of sequents:

$$(P \cup Q)^* \equiv P^* \mid Q^* \ .$$

Thus, the formulae of Eq. (4.4) are equivalent to

$$\iota\left(\bigwedge_{i=1}^{k} \left(\texttt{rwi} \cup \texttt{addi} \cup \texttt{assum} \cup (\varGamma \Longrightarrow \varDelta) \right)^* \rightarrow \left(\texttt{findSeq} \cup \texttt{assum} \cup (\varGamma \Longrightarrow \varDelta) \right)^* \right)$$

and can then be simplified to

$$\iota\left(\bigwedge_{i=1}^{k} \left(\texttt{rwi}^* \mid \texttt{addi}^* \right) \rightarrow \left(\texttt{findSeq}^* \mid \texttt{assum}^* \right) \right) \mid (\varGamma \Longrightarrow \varDelta)^*.$$

Showing that this formula holds for all sequents $\varGamma \Longrightarrow \varDelta$, i.e., in particular for the empty sequent, is equivalent to proving

$$\iota\left(\bigwedge_{i=1}^{k} \left(\texttt{rwi}^* \mid \texttt{addi}^* \right) \rightarrow \left(\texttt{findSeq}^* \mid \texttt{assum}^* \right) \right)$$

for all possible instantiations ι. We call the formula

$$M(\texttt{t1}) = \bigwedge_{i=1}^{k} \left(\texttt{rwi}^* \mid \texttt{addi}^* \right) \rightarrow \left(\texttt{findSeq}^* \mid \texttt{assum}^* \right) \quad (4.5)$$

the *meaning formula* of `t1`. From the construction of $M(\texttt{t1})$, it is clear that if $M(\texttt{t1})$ is valid whatever expressions we replace its schema variables with, then the taclet `t1` will be sound. Note, that the disjunctions | in the formula stem from the union operator on sequents. Intuitively, given that the premises of a rule application are true (the formulas on the left side of the implication), it has to be shown that at least one formula of the conclusion is true (the formulas on the right side of the implication).

Meaning Formulae for Taclets that do not contain all clauses

We can easily adapt Eq. (4.5) if some of the clauses of `t1` are missing in a taclet:

- If the `\find` clause is missing: in this case, `findSeq` can simply be considered as the empty sequent, which means that we can set $\text{findSeq}^* = \text{false}$ in Eq. (4.5).
- If `\assumes` or `\add` clauses are missing: again we can assume that the respective sequents are empty and set

$$\text{assum}^* = \text{false}, \qquad \text{add}i^* = \text{false}$$

- If a clause `\replacewith(rwi)` is not present: then we can normalise by setting $\text{rw}i = \text{findSeq}$, which means that the taclet will replace the focus of the application with itself. If both `\replacewith` and `\find` are missing, we can simply set $\text{rw}i^* = \text{false}$.

Example 4.46. We consider the taclet `impRight` ($\Rightarrow$ Fig. 4.3) from Sect. 4.1 that eliminates implications within the succedent. The taclet represents the rule schema

$$\frac{\texttt{phi} \Longrightarrow \texttt{psi}}{\Longrightarrow \texttt{phi} \mathbin{->} \texttt{psi}}$$

and the meaning formula is the logically valid formula

$$
\begin{aligned}
&M(\texttt{impRight})\\
&\quad = (\,\underbrace{!\,\texttt{phi}\mid\texttt{psi}}_{=\text{rw1}^*}\,) \mathbin{->} (\,\underbrace{\texttt{phi}\mathbin{->}\texttt{psi}}_{=\text{findSeq}^*}\,) \equiv\; !(\texttt{phi}\mathbin{->}\texttt{psi})\mid(\texttt{phi}\mathbin{->}\texttt{psi})\;.
\end{aligned}
$$

4.5.3 Meaning Formulae for Rewriting Taclets

The construction given in the previous section can be carried over to rewriting taclets.

──── Taclet ────────────────────────────────────

```
t2 { \assumes(assum) \find(findTerm) \inSequentState
     \replacewith(rw1) \add(add1);
     ...
     \replacewith(rwk) \add(addk) };
```
──────────────────────────────────── Taclet ────

In this case, `findTerm` and `rw1, ..., rwk` are schematic *terms*. We can, in fact, reduce the taclet `t2` to a non-rewriting taclet (note, that the union operator $\cup$ is not part of the actual taclet language).

──── Taclet ────────────────────────────────────

```
t2b { \assumes(assum) \inSequentState
      \add( (findTerm=rw1 ==>) ∪ add1 );
      ...
      \add( (findTerm=rwk ==>) ∪ addk ) };
```
──────────────────────────────────── Taclet ────

We create a taclet that adds equations `findTerm=rw1,...,findTerm=rwk` to the antecedent. Using taclet `t2b` and a general rule for applying equations in the antecedent, the effect of `t2` can be simulated.[11] On the other hand, also taclet `t2b` can be simulated using `t2` and standard rules (cut, reflexivity of equality), which means that it suffices to consider the soundness of `t2b`. Eq. (4.5) and some propositional simplifications then directly give us the meaning formula

$$
M(\texttt{t2b}) \;\equiv\; M(\texttt{t2}) \;=\; \bigwedge_{i=1}^{k} \big(\texttt{findTerm} \doteq \texttt{rw}i \mathbin{\text{->}} \texttt{add}i^{*}\big) \mathbin{\text{->}} \texttt{assum}^{*} \quad (4.6)
$$

In the same way, rewriting taclets for formulae can be treated, if equations are replaced with equivalences:

$$
\bigwedge_{i=1}^{k} \big((\texttt{findFor} \mathbin{\text{<->}} \texttt{rw}i) \mathbin{\text{->}} \texttt{add}i^{*}\big) \mathbin{\text{->}} \texttt{assum}^{*} \quad (4.7)
$$

4.5.4 Meaning Formulae in the Presence of State Conditions

Taclets that do *not* contain the statement `\inSequentState` (i.e., unlike all taclets whose soundness we have tackled so far) require a bit more care when deriving meaning formulae. As introduced in Sect. 4.4.1, there are two further modes that taclets can have, `\sameUpdateLevel` and the "default" mode without any flags. From the soundness point of view, it is meaningful to consider the following two categories of taclets:

- Taclets with mode `\sameUpdateLevel` and non-rewriting taclets with default mode: in contrast to taclets with mode `\inSequentState`, such taclets can also be applied in the scope of updates (see Table 4.7 on page 215). It is ensured that all parts of the taclets work in the same update context, i.e., the same updates occur above the taclet application focus, above assumptions of the taclet (`\assumes`) and above expressions that are modified or added by the clauses `\replacewith` and `\add`.
- Rewriting taclets with default mode: this is the most liberal case, in which the application focus can be in the scope of arbitrary modal operators. This means that the application focus can in particular be located in a different state from assumptions of the taclet (`\assumes`) or formulae that are added by an `\add` clause.

The following paragraphs sketch how meaning formulae for taclets of these two kinds can be created.

---

[11] Strictly speaking, this transformation only works if `findTerm` and `rw`i are not instantiated to terms that contain free variables from the application context, as it is allowed in the second item of Def. 4.32. We can imagine to implicitly add universal quantifiers for such variables.

Taclets with `\sameUpdateLevel`, *Non-Rewriting Taclets with Default Mode*

We only consider a taclet in default mode in which `\find` pattern is a sequent, but the same reasoning applies to rewriting taclets with the statement `\sameUpdateLevel`.

```
━━ Taclet ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
t3 { \assumes(assum) \find(findSeq)
      \replacewith(rw1) \add(add1);
      ...
      \replacewith(rwk) \add(addk) };
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Taclet ━
```

In text-book notation, the rule implemented by the taclet will consequently look as follows:

$$\frac{\mathcal{U}\,\texttt{rw1} \cup \mathcal{U}\,\texttt{add1} \cup \mathcal{U}\,\texttt{assum} \cup (\Gamma \Longrightarrow \Delta)}{\mathcal{U}\,\texttt{findSeq} \cup \mathcal{U}\,\texttt{assum} \cup (\Gamma \Longrightarrow \Delta)}$$

$$\cdots$$

$$\mathcal{U}\,\texttt{rwk} \cup \mathcal{U}\,\texttt{addk} \cup \mathcal{U}\,\texttt{assum} \cup (\Gamma \Longrightarrow \Delta)$$

We write $\mathcal{U}\,(\Gamma \Longrightarrow \Delta)$ for denoting that an arbitrary update $\mathcal{U}$ is added in front of each formula of $\Gamma \Longrightarrow \Delta$. For such a rule, we can derive a meaning formula exactly as in Sect. 4.5.2, with the only difference that the whole formula is preceded with the update $\mathcal{U}$:

$$\mathcal{U} \left(\bigwedge_{i=1}^{k} \left(\texttt{rw}i^* \mid \texttt{add}i^*\right) \mathbin{-\!\!>} \left(\texttt{findSeq}^* \mid \texttt{assum}^*\right) \right) \tag{4.8}$$

Fortunately, now the update $\mathcal{U}$ can be left out: because $\mathcal{U}$ can be the empty update `skip`, the validity of (4.8) entails that also the formula after $\mathcal{U}$ has to be valid. But if the formula after $\mathcal{U}$ is logically valid, i.e., if it is true for all structures and states, then (4.8) also has to hold for arbitrary updates $\mathcal{U}$. We can thus define the meaning formula of `t3` as in Sect. 4.5.2:

$$M(\texttt{t3}) \;=\; \bigwedge_{i=1}^{k} \left(\texttt{rw}i^* \mid \texttt{add}i^*\right) \mathbin{-\!\!>} \left(\texttt{findSeq}^* \mid \texttt{assum}^*\right) \tag{4.9}$$

Rewriting Taclets with Default Mode

The second and more difficult case concerns rewriting taclets where the application focus can be in the scope of arbitrary modal operators. We consider a taclet similar to the one treated in Sect. 4.5.3.

─ Taclet ──────────────────────────────────

```
t4 { \assumes(assum) \find(findTerm)
     \replacewith(rw1) \add(add1);
     ...
     \replacewith(rwk) \add(addk) };
```
──────────────────────────────── Taclet ─

The strategy followed in Sect. 4.5.3 for deriving a meaning formula for `t2` was to find an equivalent non-rewriting taclet. For `t4`, such a taclet would need to have the following shape:

─ Taclet ──────────────────────────────────

```
t4b { \assumes(assum) \inSequentState
      \add( (∀𝒰.𝒰 (findTerm=rw1) ==>) ∪ add1 );
      ...
      \add( (∀𝒰.𝒰 (findTerm=rwk) ==>) ∪ addk ) };
```
──────────────────────────────── Taclet ─

The quantifiers $\forall\mathcal{U}$. have to be added in order to ensure that the inserted equations are also applicable in the scope of modal operators. Such quantifiers over states do not exist in our dynamic logic, but can be added in a straightforward way (they are, in fact, present in the KeY implementation in a similar form). The meaning formula of `t4b` would be

$$M(\texttt{t4b}) \;\; = \;\; \bigwedge_{i=1}^{k} \left(\forall\mathcal{U}.\mathcal{U} \; (\texttt{findTerm} \doteq \texttt{rw}i) \rightarrow \texttt{add}i^* \right)$$
$$\rightarrow \texttt{assum}^*$$

4.5.5 Meaning Formulae for Nested Taclets

So far, only taclets were considered that do not contain the `\addrules` clause ($\Rightarrow$ Sect. 4.4.7). The keyword `\addrules` makes it possible to nest taclets and to use one taclet as part of another. For a start, we consider taclets of the following shape:

─ Taclet ──────────────────────────────────

```
t3 { \assumes(assum) \find(findSeq) \sameUpdateLevel
     \replacewith(rw1) \add(add1) \addrules(s1_1;...; s1_m1);
     ...
     \replacewith(rwk) \add(addk) \addrules(sk_1;...; sk_mk) };
```
──────────────────────────────── Taclet ─

where `s1_1`, ..., `sk_mk` are again taclets (we will call them *sub-taclets* in the next paragraphs). We can construct meaning formulae of such taclets recursively and using a similar argument as in Sect. 4.5.3 about rewriting taclets. Essentially, one can imagine replacing taclet `t3` with a taclet that introduces the meaning formulae of the sub-taclets `s1_1`, ... in the antecedent using the `\add` clause:

```
──── Taclet ───────────────────────────────────────
t3b {
  \assumes(assum) \find(findSeq) \sameUpdateLevel
  \replacewith(rw1) \add((M(s1_1),..., M(s1_m1) ==>) ∪ add1);
  ...
  \replacewith(rwk) \add((M(sk_1),..., M(sk_mk) ==>) ∪ addk) };
──────────────────────────────────── Taclet ──────
```

This is not directly possible, because the meaning formulae of the sub-taclets will contain schema variables whose instantiation is not yet determined when applying t3, but it leads us to the following variant of Eq. (4.5):

$$
\begin{aligned}
M(\texttt{t3}) \;=\;\; & \textstyle\bigwedge_{i=1}^{k} \big(M(\texttt{s}i\_1) \,\&\, \cdots \,\&\, M(\texttt{s}i\_\texttt{m}i) -> (\texttt{rw}i^* \,|\, \texttt{add}i^*)\big) \\
& -> \big(\texttt{findSeq}^* \,|\, \texttt{assum}^*\big)
\end{aligned}
$$

In the same way, Eq. (4.6) and Eq. (4.7) can be extended to take sub-taclets into account.

Example 4.47. In order to illustrate meaning formulae for nested taclets, we consider the taclet applyEqAR ($\Rightarrow$ Sect. 4.1). The meaning formulae for the sub-taclet rewrWithEq and the complete taclet are

$$
\begin{aligned}
M(\texttt{rewrWithEq}) \;\;=\;\;& \texttt{t1} \doteq \texttt{t2} \\
M(\texttt{applyEqAR}) \;\;=\;\;& M(\texttt{rewrWithEq}) \,|\, \texttt{t1} \,!\!\doteq \texttt{t2} \\
=\;\;& \texttt{t1} \doteq \texttt{t2} \,|\, \texttt{t1} \,!\!\doteq \texttt{t2}
\end{aligned}
$$

Obviously, $M(\texttt{rewrWithEq})$ is not a valid formula for most instantiations of the variables t1 and t2, which reflects the observation from Sect. 4.4.7 that the taclet is not correct in general. As $M(\texttt{applyEqAR})$ is a tautology, however, rewrWithEq is correct in situations in which applyEqAR can be applied, which distinguishes admissible instantiations of t1 and t2.

Unfortunately, there is one difficulty when dealing with nested taclets. For some taclets, which we consider in the following as ill-formed, the meaning formulae defined so far do not ensure soundness:

Example 4.48. We derive the meaning formula of the following taclet, which—at first glance—seems to implement the cut rule, but which in fact can be used to add arbitrary formulae to a sequent:

```
──── Taclet ───────────────────────────────────────
illegalTac3 { \addrules( introduceRight { \add(==> phi) } );
             \addrules( introduceLeft  { \add(phi ==>) } ) };
──────────────────────────────────── Taclet ──────
```

The "meaning formula" is the tautology $M(\texttt{illegalTac3}) \equiv \texttt{phi} \mid \texttt{!phi}$, however. This does not reflect that the two occurrences of `phi` can be instantiated independently when applying `introduceRight` and `introduceLeft`.

The problem with taclets like this is that different instantiations of schema variables can be chosen when applying the sub-taclets, whereas one schema variable only represents one and the same expression in the meaning formula. `illegalTac3` can be corrected to a taclet that better reflects the nature of schema variables in sub-taclets:

―― Taclet ――――――――――――――――――――――――――――――――――――

```
legalTac3 { \addrules( introduceRight { \add(==> phi1) } );
            \addrules( introduceLeft  { \add(phi2 ==>) } ) };
```
―――――――――――――――――――――――――――――――――――――― Taclet ――

Now, the meaning formula is $M(\texttt{legalTac3}) \equiv \texttt{!phi1} \mid \texttt{phi2}$ and is no longer valid.

The following requirement prohibits taclets like `illegalTac3` and could be seen as an item that belongs to Sect. 4.4.3 about well-formedness of taclets. It is, however, only important when deriving meaning formulae of taclets (it is irrelevant for the effect of taclets according to Sect. 4.4.6), and we assume only in this section that it is satisfied by considered taclets. We demand that common schema variables of sub-taclets of a taclet t also appear in t outside of sub-taclets, which entails that they are already instantiated when applying t. Arbitrary taclets can easily be transformed into equivalent taclets that respect this property.

Requirement 4.49 (Uniqueness of Variables in Sub-Taclets). *If a taclet t has two sub-taclets containing a common schema variable* `sv`*, then* `sv` *also appears in t outside of* `\addrules` *clauses.*

4.5.6 Elimination of Schema Variables

Meaning formulae of taclets in general contain schema variables, i.e., placeholders for syntactic constructs like terms, formulae or programs. In order to prove a taclet sound, it is necessary to show that its meaning formula is valid for all possible instantiations of the schema variables. Looking at Example 4.47, for instance, we would have to prove the formula

$$M(\texttt{applyEqAR}) \quad = \quad \texttt{t1} \doteq \texttt{t2} \mid \texttt{t1} \mathrel{!\dot=} \texttt{t2}$$

for all terms $\iota(\texttt{t1})$, $\iota(\texttt{t2})$ that we can substitute for `t1`, `t2`. Note, that this *syntactic* quantification ranges over terms and is completely different from a first-order formula $\forall x : integer. \, p(x)$, which is *semantic* and expresses that x ranges over all integers.

Instead of explicitly enumerating instantiations using techniques like induction over terms, it is to some degree possible, however, to replace the

syntactic quantification with an implicit semantic quantification through the introduction of Skolem symbols. For $M(\texttt{applyEqAR})$, it is sufficient to prove the formula

$$\phi \quad = \quad c \doteq d \mid c \mathrel{!\dot=} d$$

in which c, d are fresh constant symbols. The validity of $M(\texttt{applyEqAR})$ for all other instantiations follows, because the symbols c, d can take the values of arbitrary terms $\iota(\texttt{t1})$, $\iota(\texttt{t2})$. Fortunately, ϕ is only a first-order formula that can be tackled with a calculus as defined in Chap. 2.

We will only sketch how Skolem expressions can be introduced for some of the schema variable kinds that are described in Sect. 4.2. Schema variables for program entities are left out at this point, a detailed description that also covers such variables can be found in [Bubel et al., 2004]. Also, more involved features like generic types are not considered here. For the rest of the section, we assume that a taclet t and its meaning formula $M(t)$ are fixed. We then construct an instantiation ι_{sk} of the schema variables that turn up in t with Skolem expressions. In the example above, this instantiation would be

$$\iota_{\mathrm{sk}} = \{\texttt{t1} \mapsto c, \ \texttt{t2} \mapsto d\}$$

Variables: \variables A

Because of Def. 4.30, instantiations of schema variables $\texttt{va}$ for logical variables are always distinct. Such variables only occur bound in taclets and the identity of bound variables does not matter. Therefore, $\iota_{\mathrm{sk}}(\texttt{va})$ can simply be chosen to be a fresh logical variable $\iota_{\mathrm{sk}}(\texttt{va}) = x$ of type A.

Terms: \term A

As already shown in the example above, a schema variable $\texttt{te}$ for terms can be eliminated by replacing it with a constant or a function term. In general, also the context variables $\Pi_t(\texttt{te})$ of $\texttt{te}$ have to be taken into account and have to appear as arguments of the function symbol. The reason is that such variables can occur in the term that is represented by $\texttt{te}$. We choose the instantiation $\iota_{\mathrm{sk}}(\texttt{te}) = f_{\mathrm{sk}}(x_1, \ldots, x_l)$, where

- $x_1, \ldots, x_l$ are the instantiations of the schema variables $\texttt{va}_1, \ldots, \texttt{va}_l$, i.e., $x_i = \iota_{\mathrm{sk}}(\texttt{va}_i)$,
- $\texttt{va}_1, \ldots, \texttt{va}_l$ are the (distinct) context variables of the variable $\texttt{te}$ in the taclet t: $\Pi_t(\texttt{te}) = \{\texttt{va}_1, \ldots, \texttt{va}_l\}$,
- $f_{\mathrm{sk}} : A_1, \ldots, A_l \to A$ is a fresh function symbol,
- $A_1, \ldots, A_l$ are the types of $x_1, \ldots, x_l$ and $\texttt{te}$ is of kind \term A.

As a further complication, the symbol f_{sk} has to be *non-rigid* (unless the schema variable modifier rigid is used), because the term that is represented by $\texttt{te}$ can also be non-rigid. This entails that updates in front of f_{sk} matter, in contrast to rigid function symbols where such updates can immediately be removed, e.g.

$$\{o.a := 3\} f_{\mathrm{sk}}(x) \quad \neq \quad f_{\mathrm{sk}}(x)$$

Formulae: `\formula`

The elimination of schema variables `phi` for formulae is very similar to the elimination of term schema variables. The main difference is, obviously, that instead of a non-rigid function symbol a non-rigid predicate symbol has to be introduced: $\iota_{\mathrm{sk}}(\texttt{phi}) = p_{\mathrm{sk}}(x_1, \ldots, x_l)$, where

- $x_1, \ldots, x_l$ are the instantiations of the schema variables $\texttt{va}_1, \ldots, \texttt{va}_l$, i.e., $x_i = \iota_{\mathrm{sk}}(\texttt{va}_i)$,
- $\texttt{va}_1, \ldots, \texttt{va}_l$ are the (distinct) context variables of the variable `te` in the taclet t: $\Pi_t(\texttt{te}) = \{\texttt{va}_1, \ldots, \texttt{va}_l\}$,
- $p_{\mathrm{sk}} : A_1, \ldots, A_l$ is a fresh predicate symbol,
- $A_1, \ldots, A_l$ are the types of $x_1, \ldots, x_l$.

Skolem Terms: `\skolemTerm A`

Schema variables of kind `\skolemTerm` A are responsible for introducing fresh constant or function symbols in a proof. Such variables could in principle be treated like schema variables for terms, but this would strengthen meaning formulae excessively (often, the formulae would no longer be valid even for sound taclets).

We can handle schema variables `sk` for Skolem terms more faithfully: if in implication (4.1) the sequents $P_1, \ldots, P_k$ contain symbols that do not occur in Q, then these symbols can be regarded as universally quantified. Because a negation occurs in front of the quantifiers in (4.3) (the quantifiers are on the left side of an implication), the symbols have to be considered as existentially quantified when looking at the whole meaning formula. This entails that schema variables for Skolem terms can be eliminated and replaced with existentially quantified variables: $\iota_{\mathrm{sk}}(\texttt{sk}) = x$, where x is a fresh variable of type A.[12] At the same time, an existential quantifier $\exists x.$ has to be added in front of the whole meaning formula.

Example 4.50. The meaning formula of the taclet `allRight` ($\Rightarrow$ Sect. 4.1) is

$$M(\texttt{allRight}) \quad = \quad \{\texttt{\textbackslash subst x; cnst}\}(\texttt{phi}) -> \forall\texttt{x.phi}$$

In order to eliminate the schema variables of this taclet, we first assume that the generic type `G` of the taclet is instantiated with a concrete type A. Then, the schema variable `x` can be replaced with a fresh logical variable $\iota_{\mathrm{sk}}(\texttt{x}) = y$ of type A. The schema variable `phi` is eliminated through the instantiation $\iota_{\mathrm{sk}}(\texttt{phi}) = p_{\mathrm{sk}}(y)$, where p_{sk} is a fresh non-rigid predicate symbol. Finally, we can replace the schema variable `cnst` for Skolem terms with a fresh logical variable $\iota_{\mathrm{sk}}(\texttt{cnst}) = z$ of type A and add an existential quantifier $\exists z.$. The resulting formula without schema variables is

$$\exists z. \, \iota_{\mathrm{sk}}(M(\texttt{allRight})) \quad = \quad \exists z. \, (p_{\mathrm{sk}}(z) -> \forall y. \, p_{\mathrm{sk}}(y))$$

---

[12] Strictly speaking, this violates Def. 4.9, because schema variables for Skolem terms must not be instantiated with variables according to this definition. The required generalisation of the definition is, however, straightforward.

4.5.7 Introducing Lemmas in KeY

The derivation of meaning formulae for taclets and the elimination of schema variables is implemented in KeY and is automatically carried out when lemma taclets are loaded. This ensures that only those taclets can be introduced during a proof that can be derived and justified based on already existing taclets. At the time of writing this book, the implementation does not completely support schema variables for program entities or more involved features like generic types. Loading lemmas is, therefore, mostly applicable for first-order rules or for rules that describe properties of theories, such as the lemma `expSplit` in Fig. 4.2 in the introduction.

Part II

Expressing and Formalising Requirements

5

Formal Specification

by
Andreas Roth
Peter H. Schmitt

This chapter serves as an introduction to formal specifications. In Sect. 5.1 we reconsider in greater detail, but still on a fairly general level, the basic building blocks of formal specification—pre- and postconditions, invariants, and modifies clauses—that have already been informally introduced in Sect. 1.3. The next two sections then show how these notions can be formulated in two popular specification languages, OCL and JML. A short comparison between the two languages in Section 5.4 concludes this chapter.

Methodological questions like: How should operation contracts be inherited by subclasses? At which system states are invariants required to hold? How can modular specification and verification be effected? will be postponed till Chapter 8.

5.1 General Concepts

Specifications may be used at different stages in the software development process. They may be attached to a coarse design model or to runnable code or at any stage in between. What is essentially needed for the kind of specifications we treat in this book are

1. a notion of a state, e.g., the state or snapshot of a system model or the state of computation of a JAVA program;
2. a notion of a transition from pre-state to post-state effected by an operation;
3. a language to formulate specifications. It is understood that we should be able to determine whether a statement in this language is true or false in any given state.

In the example in Sect. 1.3 user authentication was not considered. Let us address this task now. We think of a user having inserted her bankcard into an automatic teller machine (ATM). After some basic initalisation the machine

performs an operation we choose to call `enterPIN`. The user is prompted to enter her pin.

Let us start thinking about what specifications we may want for `enterPIN`.

5.1.1 Operation Contracts

We decide to allow three attempts to enter the correct pin. So, it is natural to introduce an attribute `wrongPINCounter` that counts the number of unsuccessful attempts. An operation contract for `enterPIN` may then look like this:

> *precondition*   card is inserted, user not yet authenticated
> *postcondition*   if entered pin is correct
>   then the user is authenticated,
>   if entered pin is incorrect
>   and `wrongPINCounter < 2`
>   then `wrongPINCounter` is increased by 1
>   and user is not authenticated,
>   if entered pin is incorrect
>   and `wrongPINCounter >= 2`
>   then the card is confiscated
>   and user is not authenticated.

With this concrete example in mind we are ready for the general definition. Here and in the following we will use the word *operation* in a general sense and refer to implementations of operations as *methods*.

Definition 5.1. *A contract with precondition and postcondition for an operation op is satisfied if:*

> *When op is called in any state that satisfies the precondition then op terminates and in any terminating state of op the postcondition is true.*

This definition looks innocuous enough, but it is worth stressing the following facts.

1. No guarantees are given when the precondition is not satisfied. If for some reason the `enterPIN` method is called when no card is inserted, there is no telling what happens.
2. By default, termination is part of the contract. There are other options though and we will come back to this a bit later in this subsection.
3. The terminating state may be reached by normal or by abrupt termination, i.e., termination by *op* throwing an exception.

It is usual to allow more than one pre-/postcondition pair in a contract. The above example could be rephrased as:

precondition	card is inserted, user not yet authenticated, pin is correct
postcondition	user is authenticated,
precondition	card is inserted, user not yet authenticated, pin is incorrect and `wrongPINCounter < 2`
postcondition	`wrongPINCounter` is increased by 1, and user is not authenticated,
precondition	card is inserted, user not yet authenticated, pin is incorrect and `wrongPINCounter >= 2`
postcondition	card is confiscated and user is not authenticated.

In this example, the preconditions are mutually exclusive. This is not required in general.

In non-trivial contexts it is not easy to come up with a postcondition that precisely defines an operation. One particular difficulty is to state what items do not change. This is a notorious problem in many areas of computer science that deal with some notion of *action* and has been given a special name: the *frame problem*. The first observation towards a solution of the frame problem, at least in our context, is that we should make no attempt to enumerate the items that do not change. There are hopelessly many. Rather we should try to determine those items that at most may get changed. We thus add an explicit list of items that may be modified by a method to its specification. This is not an uncommon approach, and has consequently been pursued in the book [Morgan, 1990].

In our `enterPIN` example the modification list might look like this:

modifies	`wrongPINCounter` all attributes needed for user authentication all attributes needed for confiscating the card

Since we have at this level of the design not fixed all attributes we have to be a bit vague about the modification required for user authentication. It is easy to imagine how a card gets confiscated in the real world; it ends up in a special box of the machine to be picked up by a clerk. We will see below how this can be modelled in a specification.

The requirements given in Definition 5.1 go in program verification theory by the name of *total correctness*. When termination is not required we speak of *partial correctness*. The choice between these alternatives is realised by adding another clause `termination` to the contract. Possible values for this clause could be `required`, `not required` or `normal termination`. In the last case we do not want termination to be brought about by an exception.

For a uniform treatment we stipulate that all operations have a contract. If contract parts are not explicitly given, we assume the defaults in Table 5.1.

Table 5.1. Default contracts

Default Contracts	
precondition	*true*
postcondition	*true*
modifies	*everything*
termination	*required*

5.1.2 Invariants

Another frequently used specification method is that of an invariant, i.e., a statement that is required to be true in all system states. A possible invariant in the `enterPIN` scenario is:

> *invariant*   `wrongPINCounter` is always ≥ 0 and ≤ 2

Since in our model of the banking world the `wrongPINCounter` attribute is attached to the class `ATM`, this invariant says that in all system states, for all ATM-machines `m` that exist in this state, $0 \leq$ `m.wrongPINCounter` ≤ 2 is satisfied.

In object-oriented programming and design it is has become customary to attach invariants to classes or interfaces. We will follow this practice. Frequently invariants address only one instance of a class. This has lead to the figure of speech of an object satisfying an invariant throughout its lifetime. Though this view is helpful in many cases, it fails to cover invariants that address only static fields. Also, invariants addressing two instances of a class are not covered naturally since one of them has to be chosen as the main actor, arbitrarily. An example of the latter type of invariants is the following, attached to the class `BankCard` (see Figure 5.1 below):

> *invariant*   no two cards have the same `cardNumber`

Invariants may even address fields and objects from different classes. To give an example we expand our scenario by considering a central host to which the ATM is connected. In particular we imagine that this central host provides an attribute `validCardsCount` satisfying:

> *invariant*   `validCardsCount` equals the number of issued
>   bank cards that are still valid

The next step towards a thorough understanding of the concept of an invariant is the answer to the question: When should an invariant hold? The first and quick description we used at the beginning of this subsection, that an invariant should hold in all system states, is in most cases far too strong. This leads us to consider two notions of invariants:

- *strong invariants* that really hold in all system states
  (these will be treated in Sections 9.2 and 9.4),

- *invariants* without further qualification
  (we will continue to consider these here).

As a first approximation we may require an invariant to be true as long as
no operation is executing. The method `ATM::confiscateCard()` will prob-
ably first set `insertedCard = null` which will destroy the above invariant
on `validCardsCount`. Only after `ATM::confiscateCard()` updates the field
`validCardsCount` will it be true again. We record the present state of our
discussion in the following definition:

Definition 5.2. *A class C satisfies an invariant Inv if,*

1. *for any operation op and any state s satisfying at least one precondition
   of op and Inv, the invariant Inv is also true in any terminating state.*
2. *Inv is true in the state reached after execution of any constructor.*

Notice, that nothing is said here about termination or truth of the postcon-
dition. These issues are settled in the operations contract for *op*.

This is a first working definition that will be sufficient for the purposes of
the present chapter. It does not address the special case of invariants involv-
ing only static fields and leaves open which operations should be considered.
Intuitively all exported operations of the class to which the invariant is at-
tached should suffice. This is not true in all cases ($\Rightarrow$ Chap. 8).

As can be seen from Definition 5.2, invariants are in principle superfluous,
one could add them to pre- and postconditions of every operation contract.
Obviously, this is not a very practical alternative in particular in a context
where new subclasses are added to an existing program.

The reader might wonder at this point how invariants and operation con-
tracts behave with respect to the class hierarchy. Our position on this issue
is:

- an invariant of a class is inherited by all its subclasses,
- on the other hand an operation redefined in a subclass does not inherit
  the operation contract from the superclass.

Given the number of subtyping disciplines that have been proposed in the
literature this non-committal approach for operation contracts seems to be
best suited.

An example of another kind of invariant that is useful at a later stage in
the software development when code is already present are loop invariants:

```
/* loop invariant Inv */
while ( guard ) {
        body
       }
```

The intention is that `Inv` is true in the state before the while loop is entered
and again in the states after each execution of its body. No commitment is
made on the termination of the loop.

5.2 Object Constraint Language

The Object Constraint Language (OCL) is part of the OMG standard Unified Modeling Language, UML[1]. An easy introduction is available through the book [Warmer and Kleppe, 2003]. Material on a precise semantics of OCL is contained in the volume [Clark and Warmer, 2002], in particular [Gogolla and Richters, 2002]. Another source for a formal semantics is [Brucker and Wolff, 2002]. There is also Chapter 10 of the standard describing the semantics in terms of UML plus OCL. But, not many people found this account accessible. Our text follows the draft of the OCL Version 2.0 standard as of June 6, 2005, [OCL 2.0].

OCL was introduced to express those parts of the meaning that diagrams cannot convey by themselves. It was first developed in 1995 by Jos Warmer and Steve Cook. The most extensive use of OCL so far is within the UML standard itself, where it is used in the semantics description of the UML metamodel. For an example of the use of OCL in API specification see [Larsson and Mostowski, 2004].

OCL is perceived by its creators as a *formal* language. On the other hand they put emphasis on the fact that OCL is not designed for people who have a strong mathematical background. We quote from [Warmer and Kleppe, 1999a, Preface]:

> The users of OCL are the same people as the users of UML: software developers with an interest in object technology. OCL is designed for usability, although it is underpinned by mathematical set theory and logic.

5.2.1 OCL by Example

Before we enter into a systematic treatment of OCL we start with some instructive examples. The UML standard allows one to add constraints to almost every diagram type. In this chapter we exclusively consider constraints in UML class diagrams and use the diagram in Figure 5.1 as our running example. It is in fact the UML class model for the scenario previously sketched in Sect. 5.1. It contains the three classes `ATM`, `BankCard` and `CentralHost`. Of the attributes for the `ATM` class we have already encountered `wrongPINCounter`. The attribute `insertedCard` is either `null` or points to the instance of class `BankCard` that is currently inserted in the ATM. We use the Boolean field `customerAuthenticated` to model whether the inserted card is authenticated or not. The last attribute `centralHost` points to the central host the ATM in question is attached to.

Figure 5.2 shows how the informal contract for the `enterPIN` operation given above translates into OCL.

---

[1] See http://www.uml.org
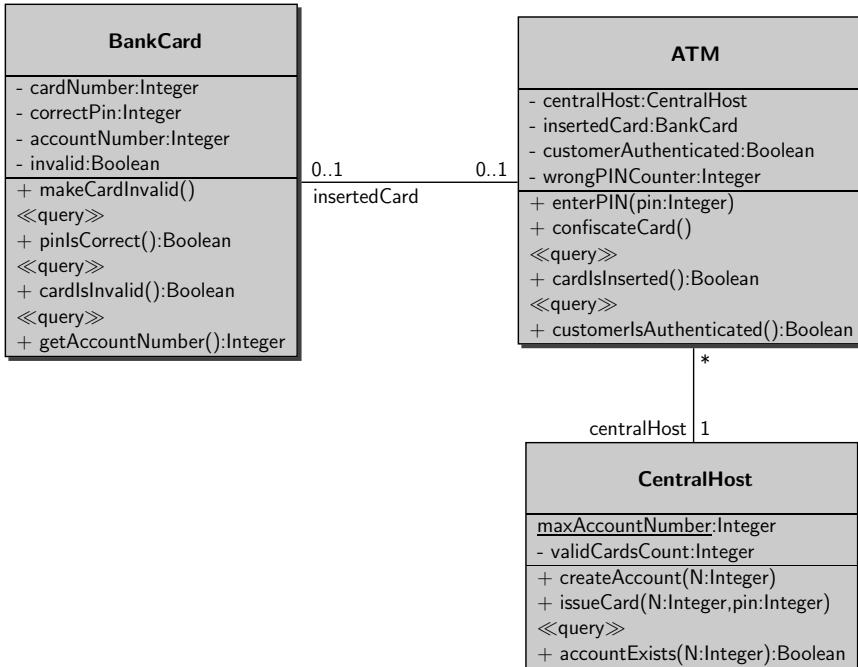
Fig. 5.1. Class diagram for the ATM scenario

This specification uses two features that are not present in the OCL standard but are implemented in the KeY system. The modifies clause we have simply added since it proved indispensable for our purposes and also since it increases compatibility with JML ($\Rightarrow$ Sect. 5.3). The literal `null` is the only element in the OCL type `VoidType` and serves us to denote JAVA's null object. We will comment on this later, Section 5.2.4. This being said let us have a closer look at this OCL operation contract.

First we observe that the syntactical entities used either come from the class diagram or are OCL built-ins. From the class diagram we are allowed to use in OCL: names of classes (does not occur in this example, but will appear shortly), attributes, association ends (in the present example this does not show since, incidentally, any association end also occurs as an attribute), and queries. Arbitrary operation names may not occur outside the context declaration since OCL is intended to be side-effect free.

Next you would expect that in an object-oriented setting an attribute like `insertedCard` in the above precondition is applied to a particular object. In fact it is, you just do not see it. By a frequently used shorthand, object references may be omitted and will be replaced by default with the reserved variable `self` which plays in OCL the same role that `this` plays in JAVA. Thus the precondition from Fig. 5.2 reads in full:

```
———— OCL ————————————————————————————————
context  ATM::enterPIN(pin: Integer)
modifies: customerAuthenticated, wrongPINCounter,
          insertedCard, insertedCard.invalid
pre:      insertedCard <> null and not customerAuthenticated
post:     if pin = insertedCard@pre.correctPIN
             then customerAuthenticated
             else
               if wrongPINCounter@pre < 2
                  then wrongPINCounter = wrongPINCounter@pre + 1
                       and not customerAuthenticated
                  else
                     insertedCard = null
                     and insertedCard@pre.invalid
                     and not customerAuthenticated
                  endif
           endif
—————————————————————————————————— OCL ——
```

Fig. 5.2. An OCL contract for the `enterPIN` operation

```
——— OCL ——————————————————————————————————
context ATM::enterPIN(pin: Integer)
pre:    self.insertedCard <> null and
        not self.customerAuthenticated
—————————————————————————————————— OCL ——
```

OCL offers as a third possibility that you declare your own local reference, e.g., `atm1`:

```
——— OCL ——————————————————————————————————
context atm1:ATM::enterPIN(pin: Integer)
pre:    atm1.insertedCard <> null and
        not atm1.customerAuthenticated
—————————————————————————————————— OCL ——
```

These references `self` or `atm1` are implicitly universally quantified, i.e., the intended meaning of an operation contract is: in all system states and for all instances `atm1` of class `ATM`, if the precondition for `atm1` is satisfied, then the postcondition is satisfied for `atm1`. Implicit universal quantification also applies to any other reference occurring in the contract, like the argument `pin` in our example.

Still looking at Figure 5.2, we notice the peculiar `@pre` symbol. Only in postconditions it may be attached to attributes, associations or queries and refers to the value of the corresponding model element before execution of the operation.

Let us look at some more examples of OCL constraints. The following contract explains how we model confiscation of cards.

```
─── OCL ────────────────────────────────────────
context ATM::confiscateCard()
pre:    insertedCard <> null
post:   insertedCard = null and insertedCard@pre.invalid
                                            ─── OCL ───
```

The attribute `insertedCard` is reset to `null`. This is obvious, since after confiscation there is no card inserted. In our model, however, the card in question is still an instance of class `BankCard` undistinguished from all other instances. To avoid this we introduced the attribute `invalid` of `BankCard` which is set to `true` when the card gets confiscated. Notice, we have to use `insertedCard@pre` since in the post state `insertedCard` is `null`.

So much for operation contracts. Let us now present examples of OCL invariants.The simplest invariant from Section 5.1.2 is formalised in OCL as:

```
─── OCL (5.1) ───────────────────────────────────
context ATM
inv:    0 <= wrongPINCounter && wrongPINCounter <= 2
                                            ─── OCL ───
```

In greater detail we would add explicitly the variable `self` which is thought of as universally quantified:

```
─── OCL ────────────────────────────────────────
context ATM
inv:    0 <= self.wrongPINCounter &&
        self.wrongPINCounter <= 2
                                            ─── OCL ───
```

The next invariant gives us the occasion to use some of the more advanced built-in concepts.

```
─── OCL (5.2) ───────────────────────────────────
context BankCard
inv:    BankCard::allInstances() ->  forall(p1,p2|
        p1<>p2 implies p1.cardNumber<>p2.cardNumber)
                                            ─── OCL ───
```

Note, that now the context is the class `BankCard`. The intended meaning of this invariant is evident. Here `allInstances()` is a query that is available for most classes. (More precisely it is inherited from the class `OclAny` for all subclasses of `OclAny`.) In any snapshot, `A::allInstances()` evaluates to the set of all existing elements of class `A`. In the OCL standard the use of `allInstances()` is restricted to classes with finitely many elements and

required to yield an undefined result when applied to a class with infinitely many elements like `String` or `Integer`. This is a viable position when you use OCL in simulation tools or for runtime checking. It is too restrictive for formal verification in general.

`BankCard::allInstances()` is our first example of an OCL expression that evaluates to a collection of objects rather than to a single object or a single value. This also accounts for the `->` symbol following `BankCard::allInstances()`. To provide for an easier reading, OCL uses `->` instead of a simple dot when applying an operation to a collection. In fact, if you change all `->` to dots in an OCL expression and then hand it to me I will be able to restore all arrows (except in one very special exceptional case). The operation that is applied to the collection `BankCard::allInstances()` in the present case is universal quantification. Unlike in other languages where you may always add quantifiers to a formula, e.g., $\forall p_1.\forall p_2.(p_1 \neq p_2 \rightarrow c(p_1) \neq c(p_2))$ in predicate logic, in OCL you first have to provide a collection that the quantifier, universal or existential, will range over. Notice also that OCL allows you to quantify two variables by one operator, just as some logic notations would allow you to write $\forall p_1, p_2.(p_1 \neq p_2 \rightarrow c(p_1) \neq c(p_2))$.

The next example introduces more operations on collections.

―― OCL (5.3) ―――――――――――――――

```
context CentralHost
inv: validCardsCount =
    BankCard::allInstances() ->
            select(not invalid) -> size()
```
――――――――――――――――――― OCL ――

We again encounter a shorthand here. The full version could look as follows:

―― OCL ―――――――――――――――――

```
inv: validCardsCount =
    BankCard::allInstances() ->
            select(c | c.invalid) -> size()
```
――――――――――――――――――― OCL ――

Now, this might look familiar to readers with a background in basic set theory. Assume that `BankCard::allInstances()` evaluates to a set A then the whole expression evaluates to the set of those elements $c \in A$ that satisfy the condition after the | symbol.

You can think of the UML class diagram and its OCL constraints as a specification, as a blue print for a system to be built. At this level one can check consistency of the specification or try to derive other properties of the specification alone. In addition, if code has been written, you will want to prove that it satisfies the constraints. A possible implementation of the `enterPIN` method is shown in Figure 5.3. If you are interested to try out

for yourself the verification of this operation contract with KeY you will find
assistance in Section 10.3.

— JAVA ——————————————————————————————

```java
public void enterPIN (int pin) {
  if ( !( cardIsInserted () &&
          !customerIsAuthenticated () ) ) {
    throw new RuntimeException ();
  }

  if ( insertedCard.pinIsCorrect ( pin ) ) {
    customerAuthenticated = true;
  } else {
    ++wrongPINCounter;
    if ( wrongPINCounter >= 3 )
    confiscateCard ();
  }
}
```

———————————————————————————————— JAVA —

Fig. 5.3. The `enterPIN` method

The examples we have seen so far were close to program code. The diagram
in Figure 5.4, however, may occur very early in system modelling. It identifies
the main classes in a role-based access scenario, `User`, `Role`, `Permission`. In
addition there are associations connecting users and roles and also roles with
permissions. No commitment is made at this point on how these relations will
be realised in code. Notice the asterisks at each association end. They stand
for multiplicities and signal that a user may have an unbounded number of
roles, a role may be assigned to an unlimited number of users, a role may be
granted an unlimited number of permissions and a permission may be part
of an unbounded number of roles.
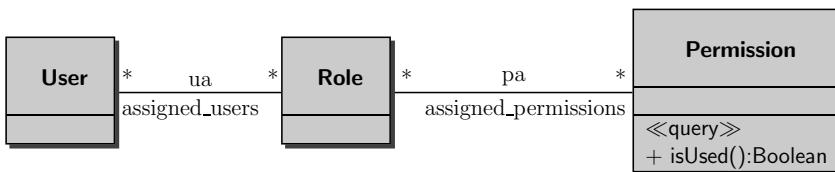
Fig. 5.4. UML class diagrams for role-based access scenario

We can think, even at this general level, of useful contracts, e.g., that every
permission is used:

───── OCL ───

```
context Permission::isUsed():Boolean
post:    result = role.assigned_users -> notEmpty()
```
── OCL ──

We already know that the postcondition is a shorthand of:

───── OCL ───

```
post:    self.result = self.role.assigned_users -> notEmpty()
```
── OCL ──

Furthermore **result** is an OCL keyword that can only be used in postconditions of operations that return a result, exactly for specifying what this result should be. Notice, that this offers the possibility to not only give a condition that should be satisfied after execution of the operation, but to uniquely define its result.

Let us look at the expression at the right hand side of the = sign. It shows that we can string together several dot-operations. That is what in OCL jargon is called *navigation*, because it has the effect of navigating through the UML class diagram. The first leg of this navigation is towards an unnamed association end. In this case the default is to use the name of the class to which the association end is attached, spelled in lower case. This first leg yields the set $R = \{r_1, \ldots, r_k\}$ of roles that are attached to the permission represented by self. The whole expression self.role.assigned_users is a much used shorthand for:

───── OCL (5.4) ───────────────────────────────────

```
post:    self.role -> collect( r | r.assigned_users)
```
── OCL ──

If for $r = r_i$ the OCL expression r.assigned_users evaluates to a set U_i of users then the whole expression evaluates to the union $U_1 \cup \ldots \cup U_k$. More precisely, this union is considered as a bag or multi-set with the consequence, that a user that is assigned to more then one role will occur more than once in the result.

5.2.2 OCL Syntax

In this subsection we try to convey a basic understanding of the syntax and semantics of OCL. The main reference for a full definition is the OMG standard draft [OCL 2.0]. This is still not the final document and does not settle all issues, but it will be more than sufficient for what we need here.

The Object Constraints Language, OCL, is a typed language; every expression has a unique type. Evaluating an expression e in a snapshot yields a value of the type of e. Figure 5.5 presents a survey of the types available in OCL. Abstract classes, i.e., classes whose instances are all instances of one

of its subclasses, are written in italics. It deviates from [OCL 2.0, Chapter 8, Figure 5] in that

- it does not show ElementType, which is mainly used to connect to state machine diagrams, which we do not consider here,
- it does not show InvalidType, since we do not use it, see Section 5.2.4.
- it does not show TypeType since the meaning of this remains unclear,
- we chose to omit OrderedSetType for brevity,
- it shows AnyType as a subclass of PrimitiveType rather than as direct subclass of Classifier. The standard's position on this is still inconsistent and the difference does not have an impact on what we have to say here.
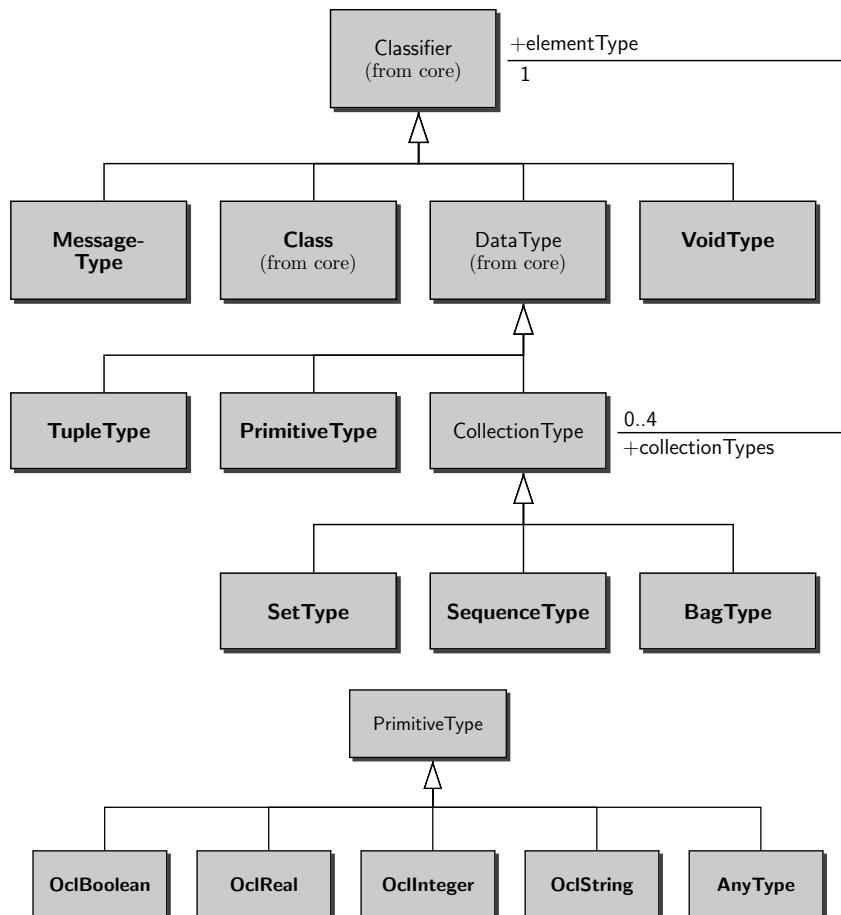
Fig. 5.5. The hierarchy of OCL types

The classes shown in Figure 5.5 are metaclasses. To illustrate what is meant by this look at `OclInteger`. This class has exactly one instance, which is named `Integer`. The instances of `Integer` in turn are the well known numbers $\ldots -1, 0, 1, \ldots$.

OCL expressions are always placed into the context of an UML class model. The classes, say $C_1, \ldots, C_k$, appearing there will be the instances of the metaclass `Class` from Figure 5.5. Evaluation of an OCL expression is always done with respect to a fixed snapshot s of the modelled system. To evaluate expressions we need to know how to evaluate types in s. This is easy for types that derive from the class diagram: the type C_i evaluates to the set of all instances of class C_i that exist in s.

`CollectionType` is an abstract class, that is to say, that any instance of it has to be an instance of one of its subclasses. For any instance C in the metaclass `Class` we have the instances `Bag(C)`, `Set(C)`, `Sequence(C)` of `BagType`, `SetType`, and `SequenceType`, respectively. The evaluation of these are, naturally, the bags, sets, sequences of elements of class `C` existing in a given snapshot s. Also `Set(Integer)`, `Set(String)` etc. occur in `SetType` and even `Set(Set(C))` and `Set(Set(Integer))` are legitimate types with the usual intended meaning.

We skip commenting on tuple types, since they are what you expect. Thus `VoidType` and `AnyType` remain. The metaclass `VoidType` has exactly one instance `OclVoid`. The only instance of `OclVoid` is the element `null`. The only instance of `AnyType` is the OCL type `OclAny`. This type is meant to be the big grab bag of almost everything. At any snapshot every instance of a model class `C`, every instance of a primitive type is also an instance of `OclAny`. The OCL standard decided that this should not apply for instances of collection or tuple types. The main reason is to steer clear of semantical problems. OCL not only uses types, but also declares a subtype relation among them, much in the same way as in the first-order logic introduced in Chapter 2. This relation is called *conformance* and is defined as follows:

Definition 5.3. *The* conforms_to *relation is the least reflexive and transitive relation on the set of all OCL types satisfying the following conditions*

1. `Integer` *conforms_to* `Real`,
2. C_1 *conforms_to* C_2 *for instances* C_i *of* `Class` *iff* C_1 *is a subtype of* C_2 *in the UML class diagram,*
3. $S(T_1)$ *conforms_to* $S(T_2)$ *for* S *one of* `Collection`, `Set`, `Bag` *or* `Sequence` *iff* T_1 *conforms_to* T_2,
4. T *conforms_to* `OclAny` *for any type* T *that is not a collection or a tuple type,*
5. `OclVoid` *conforms_to every other type,*
6. *for tuple types we have:* $Tuple(name_1{:}T_1, \ldots, name_k{:}T_k)$
   conforms_to $Tuple(name'_1{:}S_1, \ldots, name'_k{:}S_k)$
   iff $\{name_1, \ldots, name_k\} = \{name'_1, \ldots, name'_k\}$ *and for* $name_i = name'_j$
   we have T_i *conforms_to* S_j.

According to item 6 above, the OCL expression

$$Tuple(first{:}Integer, second{:}Integer, node{:}C)$$

conforms to

$$Tuple(second{:}Integer, node{:}C, first{:}Real)$$

if C conforms to D.

Having completed our survey of OCL types we are now ready to explain what OCL expressions are. It has become popular to present formal languages by a metamodel. Figure 5.6 shows the top level of the metamodel for the abstract syntax of OCL expressions. We explain bit by bit how to read this model. Classes with the label "(from core)" are classes from the metamodel of UML. They serve to connect OCL expressions to the class diagram to which they belong. The top-level class for OCL is the abstract class *OCLExpression*. If you are more comfortable with grammar rules you could express this information equivalently by the production rule
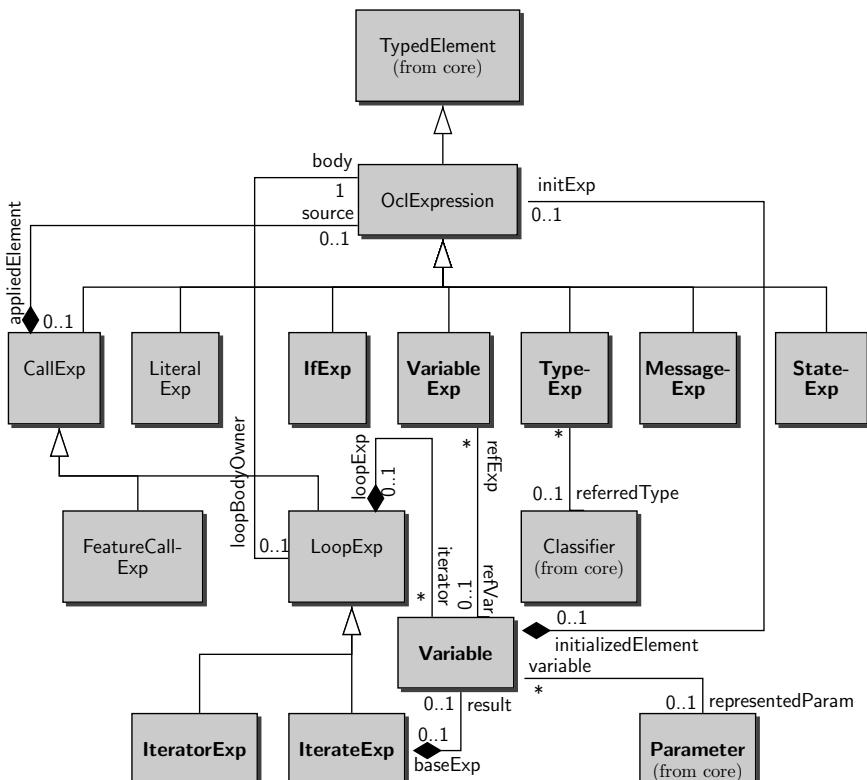
Fig. 5.6. Toplevel metaclass diagram for OCL expressions
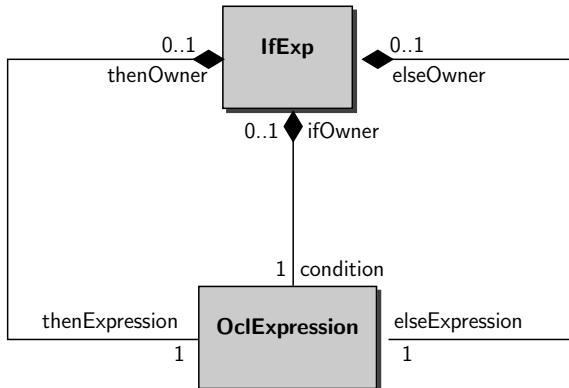
Fig. 5.7. Metamodel for conditional expressions

```
OclExpressionCS ::= CallExpCS    | VariableExpCS |
                    LiteralExpCS | LetExpCS    |
                    MessageExpCS | IfExpCS
```

We use the postfix `CS` to distinguish between the names of metaclasses and non-terminal symbols in the concrete syntax grammar. Notice that there are no non-terminals for the metaclasses `TypeExp` and `StateExp`. These will occur as parts of OCL expressions, but cannot stand alone as an OCL expression. The non-terminal `LetExpCS` has no corresponding class in Figure 5.6, because we left it out to not clutter the diagram even further.

Before we look closer into the top-level diagram, we describe the general workings of the abstract syntax model by looking at the simple metamodel for conditional expressions in Figure 5.7. The diagram shows that a conditional expression consists of OCL expressions, referred to by the association ends `condition`, `thenExpression`, and `elseExpression`. These three expressions are considered as parts of the conditional expression as signalled by the composition icons. The multiplicities at the corresponding ends, that is 1 in all cases, show that none of these may be missing. This is an elegant way to describe conditional expressions abstractly without imposing a concrete syntax.

Certainly, not every OCL expression can serve as a value for the `condition`. This restriction cannot be expressed in the metaclass diagram. Instead OCL constraints are added. For conditional expressions these are the invariants:

--- OCL ---
```
context IfExp
inv:  self.condition.type.name = 'Boolean'
inv:  self.condition.type.oclIsKindOf(PrimitiveType)
inv:  self.type = thenExp.type.commonSuperType(elseExp.type)
```
--- OCL ---

The attribute `type` is inherited from the UML metaclass `TypedElement`. The first invariant says that the type of the condition expression has to be named Boolean. Since somebody might draw a class diagram with a class named Boolean the second invariant requires that the type of the condition expression be primitive. The operation `oclIsKindOf(T)` may be found in the OCL standard library as a Boolean operation on the class `OclAny` with the explanation that `s.oclIsKindOf(T)` is true if `s` is a (not necessarily immediate) subtype of `T`. The third invariant determines the type of the *if* expression. `s.commonSuperType(t)` is a defined OCL expression that returns the least common supertype of `s` and `t` if it exists and undefined otherwise. A complete OCL definition of the `commonSyperType` operation will be given at the end of Section 5.2.2. The type of

```
if c:Boolean then t:Integer else s:Real endif
```

thus is `Real`. Finally, here is the grammar rule for the concrete syntax.

```
IfExpCS ::= 'if' OclExpCS 'then' OclExpCS
            'else' OclExpCS  'endif'
```

Let us now turn back and look at Figure 5.6 again. For abstract classes it is easy to read off the grammar rules from the metamodel:

```
CallExpCS        ::= FeatureCallExpCS | LoopExpCS
```
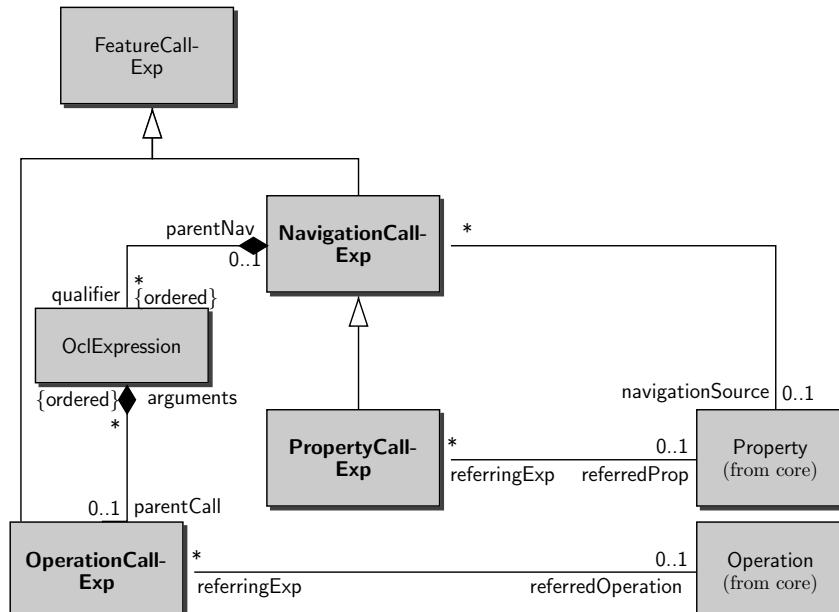

Fig. 5.8. Metamodel for OCL `featureCall` expressions

Looking at Figure 5.8 we find furthermore:

```
FeatureCallExpCS ::= NavigationCallExpCS |
                     PropertyCallCS | OperationCallExpCS
```

To gain an initial understanding, let us look at some examples taken from the constraints we had already encountered in Section 5.2.1.

The expressions `p1.cardNumber`, `p2.cardNumber` are property call expressions. So are `insertedCard@pre` and `self.insertedCard@pre`. Examples of operation call expressions are `p1.cardNumber <> p2.cardNumber`, `p1 <> p2`, `BankCard::allInstances()`. An example for the last category of feature call expressions, i.e., an example for a navigation call expression (taken from Figure 5.4 on page 255) is `self.role.assigned_users` or `role.assigned_users`.

Next we have a closer look at `OperationCallExp`. The concrete syntax grammar rule reads as:

```
OperationCallExpCS ::=
(A)  OclExpCS(1) sNameCS OclExpCS(2) |
(B)  OclExpCS '->' sNameCS'(' argumentsCS?')' |
(C)  OclExpCS'.'sNameCS ismarkedPreCS?'('argumentsCS?')' |
(D)  sNameCS ismarkedPreCS?'('argumentsCS?')' |
(G)  pathNameCS'('argumentsCS?')' |
(H)  sNameCS OclExpCS
```

As usual in grammar formalisms, a non-terminal with a question mark is optional. The rule in the standard lists additional clauses (E) and (F), which in our version are subsumed by (D) and (C). Here are typical examples for all six cases of operation call expressions:

```
(A)  wrongPINCounter + 1
     wrongPINCounter < 2
     wrongPINCounter = wrongPINCounter + 1
     insertedCard <> null
(B)  self.role.assigned_users -> notEmpty()
     s -> union(s2)
(C)  self.insertedCard.pinIsCorrect()
     self.insertedCard.pinIsCorrect@pre()
(D)  pinIsCorrect()
     pinIsCorrect()@pre
(G)  BankCard::allInstances()
(H)  -wrongPINCounter
     not cardIsInserted()
```

In the above, `sNameCS` is our shorthand notation for `simpleNameCS`. This is a string of symbols without further restrictions. There are of course additional well-formedness conditions for each of the eight production rules that make sure that the instance of `sName` is the name of an operation with the

correct typing that is available in the OCL library or in the UML model
the expression is attached to. First, `pathNameCS` is a non-empty sequence of
simple names separated by "`::`". Applying rule (G) requires to check that
`pathNameCS` ends in `className::opName()` where `className` does occur in
the context diagram and `opName()` is a static operation declared in this class.
`className` may optionally be prefixed by package names or might be implicit.
Finally, we observe that case (D) is the same as (C) only with implicit source
expression. Typically the implicit source could be the variable `self`.

Now let us look at the other two subclasses of feature call expressions. In
the rules to follow we skip the rule versions for implicit source expressions.

```
(A)   PropertyCallExpCS ::= OclExpressionCS'.'
                            sNameCS isMarkedPreCS?
(C)   PropertyCallExpCS ::= pathNameCS
```

Here `sNameCS` must match a suitable attribute name. (C) covers the case that
the attribute is static.

```
(A)   NavigationCallExpCS ::= AssociationEndCallExpCS
(B)   NavigationCallExpCS ::= AssociationClassCallExpCS
(A)   AssociationEndCallExpCS ::= OclExpressionCS'.'
          sNameCS('['argumentsCS']')? isMarkedPreCS?
(A)   AssociationClassCallExpCS ::= OclExpressionCS'.'
          sNameCS('['argumentsCS']')? isMarkedPreCS?
```

Note that the rules for `AssociationEndExpCS` and `AssociationClassExpCS`
are literally identical. The difference is that in the first rule `sNameCS` has
to match the name of an association end and in the second rule `sNameCS`
has to match the name of an association class available in the context UML
model. The optional arguments within square brackets take care of qualifiers
attached to association ends or classes.

This is all we want so say on feature call expressions. Next we turn to
loop expressions, consult Figure 5.6. Of the two subclasses of the metaclass
`LoopExp` we consider `IteratorExp` here and defer `IterateExp` to Sect. 5.2.4
on advanced topics.

```
(A)   IteratorExpCS ::=
          OclExpressionCS '->' sNameCS
          ('(VarDecl, (',' VarDecl)? '|')? OclExpressionCS')'
(B)   IteratorExpCS ::= OclExpressionCS'.'sNameCS'('argCS?')'
(C)   IteratorExpCS ::= OclExpressionCS'.'sNameCS
(D)   IteratorExpCS ::=
          OclExpressionCS'.'sNameCS ('['argumentsCS']')?
(E)   IteratorExpCS ::=
          OclExpressionCS'.'sNameCS ('['argumentsCS']')?
```

Table 5.2. Iterators from the OCL standard library

exists	any	select
forAll	one	reject
isUnique	collect	collectNested
sortedBy		

A complete listing of the built-in choices for `sNameCS` in (A) is shown in Table 5.2. New iterators may be added. The following are correct iterator expressions:

```
(A1)  source '->' 'select' '(' p '|' body ')'
(A2)  source '->' 'select' '('  body ')'
```

Assume that `source` evaluates to a collection $s = \{a_1, \ldots, a_2\}$. Then the whole expression evaluates to the subset of those elements $a_i \in s$ that satisfy `body`. If the type of `source` is not a collection type it is implicitly turned into one, with the understanding that in the evaluation an object is replaced by the singleton set containing this object.

For the remaining rules (B) to (C) it is required that the source expressions be of collection type. They are all shorthand notations for a `collect` iterator. For example, an expression `source.attribute` is shorthand for

```
source -> collect(p | p.attribute) .
```

If again `source` evaluates to $s = \{a_1, \ldots, a_n\}$, then the result of the whole expression is the set $\{a_i.attribute \mid a_i \in s\}$.

So far we have considered stand-alone OCL expressions. We now turn to the syntax OCL offers to explicitly relate constraints to UML model elements. We only discuss invariants, pre- and postconditions, and definitions, ignoring initial value, derived invariants, body, and guard expressions.

The generic form of invariants is:

—— OCL ——
```
context (x1,..,xk:)?classPath
inv (invName)?: expression
```
—————————————————— OCL ——

For operations contracts the generic form looks like this:

—— OCL ——
```
context (x1,..,xk:)?classPath::op(p1:T1,..,pn:Tn):T
pre     (prename1)?:  precondition1
post    (postname1)?: postcondition1
        :
pre     (prenamek)?:  preconditionk
post    (postnamek)?: postconditionk
```
—————————————————— OCL ——

For definitions, the generic context is shown below, where the left-hand sides
are either variables or operation declarations:

```
—— OCL ——————————————————————————————
context classPath
def:    lhs1 = ex1
        :
        lhsk = ex2
                                                    —— OCL ——
```

As an example for a definition we reproduce the definition of the least com-
mon supertype from the UML metaclass `Classifier` that had been used in
the discussion of `if` expressions.

```
—— OCL ——————————————————————————————
context Classifier
def:    commonSuperType(c:Classifier):Classifier =
        Classifier.allInstances() -> select(cst |
          c.conformsTo(cst) and self.conformsTo(cst) and
          not Classifier.allInstances() -> exists(t |
           c.conformsTo(t) and self.conformsTo(t) and
           t.conformsTo(cst) and t <> s))
         -> any()
                                                    —— OCL ——
```

For an explanation of `any()` see Figure 5.5. There is also a construct for
definitions local to a single expression, e.g.:

```
—— OCL ——————————————————————————————
context ATM::enterPIN(pin: Integer
pre:    let cardInserted = self.insertedCard <> null
        in
        cardInsterted and not self.customerAuthenticated
                                                    —— OCL ——
```

In both constructs `def` and `let` the variable or operation to be defined may
also occur on the right-hand side, i.e., arbitrary, mutual recursive definitions
are possible.

5.2.3 OCL Semantics

We define a precise meaning for OCL expressions indirectly by translating
them to first-order logic, in some cases to dynamic logic, and then referring
to the semantics explained in Chapters 2 and 3.

Signature

First we fix the signature of the target language. The types occurring in the context of the OCL expressions to be translated directly constitute a type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ as defined in Sect. 2.1. This hierarchy includes the types $\bot$ and $\top$, even though they have no corresponding OCL type. For every type B there are the types $Set(B)$, $Sequence(B)$, etc. The functions and predicates in the signature Σ of the target language are determined as follows:
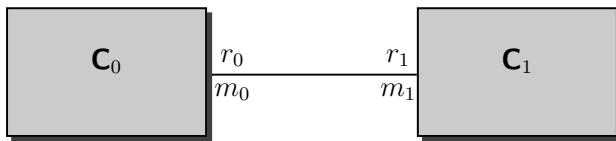
Fig. 5.9. A generic association

1. For every binary association r between classes C_0 and C_1 with role names r_0, r_1 and multiplicities m_0, m_1, see Figure 5.9, there are non-rigid function symbols in Σ with the following typing

$$\begin{array}{ll} r_1 : C_0 \rightarrow C_1 & \text{if } m_1 = 1 \\ r_1 : C_0 \rightarrow Set(C_1) & \text{if } m_1 \neq 1 \\ r_1 : C_0 \rightarrow Sequence(C_1) & \text{if } m_1 \neq 1 \text{ and the association end at } C_1 \\ & \qquad \text{is marked } \ll ordered \gg \end{array}$$

   Likewise there is a function symbol $r_0 : C_1 \rightarrow C_0$ if $m_0 = 1$ etc. In case that $m_0 = m_1 = *$, a binary predicate symbol $r : C_0 \times C_1$ is introduced in addition.
2. For n-ary associations r, an n-ary predicate symbol of the appropriate typing occurs in Σ.
3. For every attribute att in class C with result type C_r, there is a unary function $att : C \rightarrow C_r$ in Σ (if the attribute is static, the function is a constant of type C_r).
4. For every query operation op in class C with parameters of types $C_1, \ldots C_n$ and result type C_r, there is an $(n+1)$-ary function symbol op in Σ with $op : C \times C_1 \times \ldots \times C_n \rightarrow C_r$ (if the operation is a static the typing reduces to $op : C_1 \times \ldots \times C_n \rightarrow C_r$).
5. The signature Σ contains names for all operations in the OCL standard library.
6. If C is an association class attached to an association r between classes C_1 and C_2 then function symbols $c_1 : C \rightarrow C_1$ and $c_2 : C \rightarrow C_2$, or $c_1 : C \rightarrow Collection(C_1)$ and $c_2 : C \rightarrow Collection(C_2)$ depending on the multiplicities of association r, are available in Σ.

All the functions and predicate symbols introduced in the above list are non-rigid symbols.

In this presentation we use the same names for the OCL entities and their counterparts in first-order logic with the exceptions shown in Table 5.3. Functions and predicates introduced in predicate logic as counterparts of functions in the OCL library are rigid symbols.

Table 5.3. Traditional names for Boolean and set operations

OCL	Logic	OCL	Logic
not	!	x.intersection(y)	$x \cap y$
and	&	x.union(y)	$x \cup y$
or	$\mid$	x.includes(y)	$y \in x$
implies	$\rightarrow$	x.excludes(y)	$y \notin x$
x.including(y)	$x \cup \{y\}$	x.includesAll(y)	$y \subseteq x$
x.excluding(y)	$x \setminus \{y\}$	x.isEmpty()	$x \doteq \emptyset$
x.excludesAll(y)	$x \cap y \doteq \emptyset$	x.notEmpty()	$x \neq \emptyset$
x.equals(y)	$x \doteq y$	x <> y	$x \neq y$

The expression `allInstances()` is a static method in the OCL standard library. It is inherited by all types `T` extending `OclAny`, which is in particular the case for all classifiers from the UML model. For any such T there is a non-rigid constant $T :: allInstances()$ of type $Set(T)$ in our signature Σ.

The translation of iterators will be deferred for the moment.

Semantics of Expressions Without Iterators

Once a type hierarchy and a signature Σ are fixed, we can form well-sorted terms ($\Rightarrow$ Sect. 2.3). Translating OCL expressions, for the moment without iterators, into terms of first-order typed logic amounts to nothing more than changing from one concrete syntax to another. In addition we view Boolean functions as predicates. The OCL expression

```
insertedCard <> null and not customerAuthenticated
```

from Figure 5.2 on page 252 now reads

$$insertedCard(\text{self}) \neq null \; \& \; !\, customerAuthenticated(\text{self}) \;.$$

The context information tells us, that this expression plays the role of an invariant. The variable `self` is thus implicitly understood as quantified over all existing instances of `ATM`. The translation of the invariant into the KeY input language thus is:

─── KeY ───
```
\forall ATM x;(x.<created> ->
   insertedCard(x) != null & customerAuthenticated(x))
```
─── KeY ───

In our logic quantification ranges over all instances of a class, also over those not yet created. So, the restriction x.<created> had to be added to capture the meaning of the OCL constraint correctly ($\Rightarrow$ Sect. 3.6.6). Since this will happen frequently in rest of this section we use $\dot\forall x.\phi$ and $\dot\exists x.\phi$ as abbreviations for $\forall x.(x.\texttt{<created>} \rightarrow \phi)$ and $\exists x.(x.\texttt{<created>} \ \& \ \phi)$, respectively. The above invariant could thus be written as:

$$\dot\forall ATM \ x.(insertedCard(x) \mathrel{\dot!=} null \ \& \ customerAuthenticated(x)) \ .$$

In addition to what we have said so far there are also symbols f@pre in Σ for any $f \in \Sigma$ that is not already suffixed with @pre. Thus,

$$\texttt{pin = insertedCard@pre.correctPIN}$$

translates to

$$pin(\text{self}) = correctPIN(insertedCard@pre(\text{self})) \ .$$

In translating associations, see again Figure 5.9, one of the function symbols added to Σ already carries all the information. Nevertheless the redundancy to have one function symbol for each direction is highly desirable. But, when reasoning with terms over Σ we need axioms expressing the interrelations between them:

$$\dot\forall C_0 \ x.\dot\forall C_1 \ y.(r_1(x) \doteq y \Longleftrightarrow r_0(y) \doteq x) \text{ if } m_0 = m_1 = 1$$
$$\dot\forall C_0 \ x.\dot\forall C_1 \ y.(y \in r_1(x) \Longleftrightarrow r_0(y) \doteq x) \text{ if } m_0 = 1, m_1 \neq 1$$
$$\dot\forall C_0 \ x.\dot\forall C_1 \ y.(r_1(x) \doteq y \Longleftrightarrow x \in r_0(y)) \text{ if } m_0 \neq 1, m_1 = 1$$

Similar formulas have to be added for multiplicities m different from 1 and $*$. Finally, we need axioms to reason about constants of the form B::allInstances():

$$\dot\forall Object \ x.(x \in \text{B::}allInstances() \Longleftrightarrow x \mathrel{\dot\in} B) \ .$$

It is important to notice that the type $Set(T)$ is treated on the same footing as any other type in our first-order logic. There is no commitment that in an interpretation I the domain $I(Set(T))$ consists of all (finite) subsets of $I(T)$. A formula like

$$\dot\forall T \ x.\dot\exists Set(T) \ u.\forall T \ z.(z \in u \Longleftrightarrow \psi(x))$$

need not be universally valid for arbitrary ψ. If we want certain relationships between $I(Set(T))$ and $I(T)$ to hold, we have to add axioms to enforce it. We insist that the basic set theoretic operations are defined and have their usual

Table 5.4. First-order translations of some iterators

OCL	`e0->forAll(x | e1)`
FOL	$\dot\forall x.(x \in [e0] \mathbin{-\!\!>} [e1])$
OCL	`e0->exists(x | e1)`
FOL	$\dot\exists x.(x \in [e0] \mathbin{\&} [e1])$
OCL	`e0->select(x | e1)`
FOL	$s_{e0,e1}$ (new symbol) with definition
	$\dot\forall x.(x \in s_{e0,e1} \mathbin{<\!\!-\!\!>} (x \in [e0] \mathbin{\&} [e1]))$
OCL	`e0->collect(x | e1)`
FOL	$c_{e0,e1}$ (new symbol) with definition
	$\dot\forall z.(z \in c_{e0,e1} \mathbin{<\!\!-\!\!>} \dot\exists x.(x \in [e0] \mathbin{\&} z \doteq [e1]))$
OCL	`e0->isUnique(x | e1)`
FOL	$\dot\forall x.\dot\forall y.(x \in [e0] \mathbin{\&} y \in [e0] \mathbin{\&} [e1] \doteq \{x/y\}[e1] \mathbin{-\!\!>} x \doteq y)$
OCL	`e0->any(x | e1)`
FOL	$sk_{x,e0,e1}$ (new symbol) with definition
	$\dot\exists x.(x \in [e0] \mathbin{\&} [e1]) \mathbin{-\!\!>} sk_{x,e0,e1} \in [e0] \mathbin{\&} \{x/sk_{x,e0,e1}\}[e1]$

meaning. Thus we know, that for every finite subset $\{t_1, \ldots, t_n\}$ of $I(T)$ there is an $s \in Set(T)$ such that $t \in s$ is exactly true for $t = t_i$ for some $1 \le i \le n$. This is known as the *Henkin semantics* of higher order logic. We also insist that for every $Set(T)$ the following axiom is satisfied:

$$\dot\forall Object\ x.\dot\forall Set(T)\ u.(x \in u \mathbin{-\!\!>} x \mathrel{\dot\in} T)\ .$$

Semantics of Iterators

The OCL Standard Library is not systematic in the way it defines the meaning of its expressions. The union operation `union(s:Set(T)):Set(T)` on sets e.g., is defined via postconditions:

— OCL —
```
post: result->forAll(elem |
               self->includes(elem) or s->includes(elem))
post: self ->forAll(elem | result->includes(elem))
post: s ->forAll(elem | result->includes(elem))
```
— OCL —

It could just as well have been defined using the `iterate` construct:

— OCL —
```
self -> union(s : Set(T)) : Set(T) =
self -> iterate( x ; u:Set(T) = s | u->including(x))
```
— OCL —

On the other hand, `select` is defined in the standard as an iterator, but could just as well have benn characterised by postconditions:

```
——— OCL ———————————————————————————
source -> select(iterator | body)

post: result -> forAll(e | source.includes(e))
post: result -> forAll(e | body)
post: source -> forAll(e | body implies result.includes(e))
—————————————————————————————— OCL ——
```

It is easy to see that in fact all iterators can be defined this way using only
`forAll` and `exists`. to translate OCL iterators into first-order logic. See the
summary in Table 5.4, where [e] denotes the first-order logic (FOL) trans-
lation of OCL expression e,[2] and $\{x/t_0\}t_1$ is the term resulting from t_1 by
replacing all occurrences of variable x by the term t_0.

   To illustrate how the new symbols, introduced in Table 5.4, are used, let
us reconsider the expression from the invariant (5.3) on page 254:

```
validCardsCount = BankCard::allInstances() ->
                     select(not invalid ) -> size()
```

which translates to

$$validCardsCount(\text{self}) = size(a)$$

plus the definition

$$\dot\forall x.(x \in a <\!-\!> x \in BankCard :: allInstances() \ \& \ invalid(x)) \ .$$

Comparing the first-order logic translation of `any(x|e)` with its definition
in Figure 5.5 one might object that it does not take into account that the
new constant should denote the first element of its kind. But notice that
the operation `asSequence` is performed with respect to an unknown order.
Choosing the first element in an arbitrary order amounts to choosing an
arbitrary element.

Operation Contracts

An operation contract

```
——— OCL ———————————————————————————
context C::op()
pre:    pre
post:   post
—————————————————————————————— OCL ——
```

---

[2] The same notation will later be used to denote translated JML expressions, but
   there will hardly be occasions for confusing both.

is translated into the dynamic logic formula

$$[pre] \implies \langle \texttt{C::op()} \rangle [post] \ .$$

We notice, that this translation adopts the total correctness semantics, i.e., termination of the operation is required. We should also point out that the above translation treats the contract in isolation. The whole picture would also include invariants that can be assumed in proving the above implication ($\Rightarrow$ Chap. 8).

The KeY tool also offers the partial correctness semantics translation

$$[pre] \implies [\texttt{C::op()}][post]$$

as an option. Using the modal operator $\langle \rangle$ in the total correctness semantics treats abrupt termination as non-termination. If you want a postcondition to also hold after abrupt termination the contract is translated:

$$[pre] \implies \langle \texttt{try\{C::op()\}catch(java.lang.Throwable exc)\{\}} \rangle [post] \ .$$

See also the definition of $\text{Prg}_{op}()$ in Sect. 8.2.3. How abrupt termination is handled is explained in Sect. 3.6.7.

If the postcondition **post** contains as a subexpression **a@pre(exp)** the translation is [Baar et al., 2001]:

$$([pre] \ \& \ \forall x.(a@pre(x) \doteq a(x))) \implies \langle \texttt{C::op()} \rangle [post] \ .$$

Here, $a@pre$ is a new function symbol, which in particular does not occur in the body of op. On the other hand a will normally occur in the code of op. The newly added premiss $\forall x.(a@pre(x) \doteq a(x))$ outside the scope of the modal operator allows one to conclude that the value of $a@pre(x)$ after execution of C is the value of $a(x)$ before.

Paper
authors[*]:Person
number:Int
pages:Int
totalnumber:Int
sumpages:Int

```
─── OCL ──────────────────────
context Paper
inv: Paper::allInstances() ->
     iterate(x:Paper;y:Integer = 0|y+x.pages)
     = Paper::totalnumber
─────────────────────────── OCL ──
```
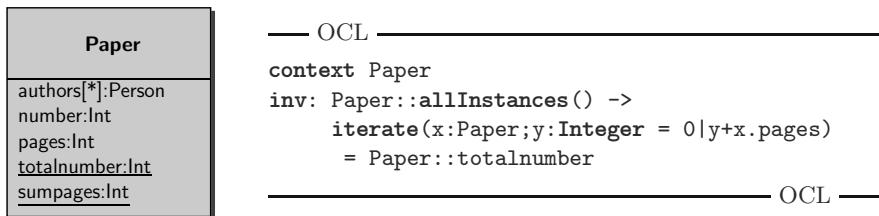
Fig. 5.10. Example of a constraint with `iterate`

5.2.4 Advanced Topics

The Iterate Construct

As the first advanced construct we now consider the `iterate` construct, the second kind of loop expression, that we had skipped in Sect. 5.2.2, see Figure 5.6 for its position within the metamodel. Figure 5.10 shows an invariant

of the class `Paper` which expresses that the static attribute `totalnumber` of this class equals, at all times, the sum of the `pages` attribute taken over all instances in the class. The general form of an iterate expression is given in Figure 5.11, which uses the names for association ends from the metamodel. The following restrictions apply:

1. variable y ist different from x,
2. variable y does not occur in the term t,
3. variables x and y do not occur in t_0,
4. the types of y and u coincide,
5. the type of t is a collection type $Collection(S)$ and x is of type S.

Given a model $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$ and an assignments β to local variables. The interpretation $\mathrm{val}_{\mathcal{M},\beta}(exp)$ for

$$exp \quad = \quad \text{t ->iterate(x;y = t0 | u)}$$

is obtained as follows. Let $A = \{a_1, \ldots, a_n\}$ be the evaluation $\mathrm{val}_{\mathcal{M},\beta}(t)$ of the source expression t. For the purposes of this definition, for any variable assignment γ, we use $\gamma[a, b]$ to denote

$$\gamma[a, b](z) := \begin{cases} a & \text{if } z = x \\ b & \text{if } z = y \\ \gamma(z) & \text{otherwise} \end{cases}$$

Using this notation we define

$$\beta_1 = \beta[a_1, \mathrm{val}_{\mathcal{M},\beta}(t0)]$$
$$\beta_{k+1} = \beta_k[a_{k+1}, \mathrm{val}_{\mathcal{M},\beta_k}(u)] \quad \text{for } k < n$$

Then, $\mathrm{val}_{\mathcal{M},\beta}(exp) = \mathrm{val}_{\mathcal{M},\beta_n}(u)$.

This definition depends in general on the ordering of the set $\{a_1, \ldots, a_n\}$. If t is of type $Sequence$ then, naturally, we use the order given by the sequence. In the other cases, it is the responsibility of the user to ensure independence from the order of evaluation.

All iterator expressions can in fact be defined in terms of an iterate expression, see Figure 5.5 below. The OCL standard is not systematic with respect to the definition of set theoretic operations. The union of two sets, e.g., is specified by an operation contract, but could just as well have been defined using `iterate`. The `select` operation on the other hand is defined via `iterate`, but could just as well have been specified by a postcondition.

We strongly recommend to avoid iterate expressions in OCL specifications, they are hard to read, they are at a low level of abstraction and they put an excessive burden on verification. If need arises, a new iterator could be defined. Its definition may use the iterate construct but in the specification only the iterator occurs.
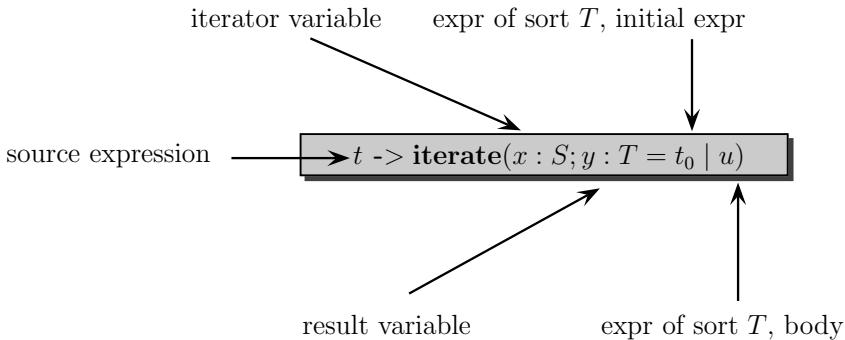
Fig. 5.11. Syntax of the `iterate` construct

Table 5.5. Definitions for some iterators

```
t->forAll(x|a)          =  t -> iterate(x;y = true| y and a )
t->exists(x|a)          =  t -> iterate(x;y = false| or a )
t-> collectNested(x|u)  =  t -> iterate(x;y = Bag{} |
                                  y ->including(u))
t->collect(x|u)         =  t->collectNested(x|u) -> flatten()
t->select(x|a)          =  t-> iterate(x;y = Collection{} |
                                  if a then y.including(x) else y
t->any(x|e)             =  t->select(x|e)-> asSequence()-> first()
t-> flatten() ᵃ         =  if
                             t.type.elementType.
                                   oclIsKindOf(CollectionType)
                           then
                             t -> iterate(c;acc:Bag = Bag{} |
                                   acc -> union(c->asBag))
                           else
                             t
                           endif
```

ᵃ This is the definition from the OCL standard, which only works for set nestings of level 2.

---

collectNested

In OCL2.0 the `collect` operation is defined via the more general operation called `collectNested`. The expression

```
self.role -> collectNested( r | r.assigned_users) ,
```

similar to the one considered in OCL example 5.4 on page 256, evaluates to $\{U_1, \ldots, U_k\}$ if for $r = r_i$ the OCL expression `r.assigned_users` evaluates to a set U_i.

Type Dependent Operations

There are three families of operators defined in `OclAny` that depend on types.

1. `oclAsType(T:TypeExp):OclAny` $\rightarrow$ `T`
2. `oclIsTypeOf(T:TypeExp):OclAny` $\rightarrow$ `Boolean`
3. `oclIsKindOf(T:TypeExp):OclAny` $\rightarrow$ `Boolean`

Note, that the type expression is not an argument to these operations. It can be viewed as part of the operation's name. Translations to first-order logic are straight forward:

$$[\texttt{e.oclAsType(T:TypeExp)}] = (T)[\texttt{e}]$$
$$[\texttt{e.oclIsTypeOf(T:TypeExp)}] = [\texttt{e}] \equiv \texttt{T}$$
$$[\texttt{e.oclIsKindOf(T:TypeExp)}] = [\texttt{e}] \sqsubseteq \texttt{T}$$

Exceptions in OCL

In contrast to JML ($\Rightarrow$ Sect. 5.3), the OCL language does not offer built-in support for talking about exceptions. This can be remedied by adding a new Boolean attribute `excThrown('T:TypeExp')` to any class. Types are attached to the `excThrown` attribute in the same way types are attached to the operations in the previous section.[3]

The use of `excThrown` only makes sense in the later stages of design when, e.g., the classes in the design model can be related to JAVA classes. The type `T` should then be a subtype of `Exception`. The easiest way to technically realize the use of `excThrown` would then be to automatically add the corresponding attribute `excThrown` to the JAVA class `Object`. Also `excThrown` can only be used in postconditions. A constraint

—— OCL ——
```
context C0::op(x1:D1,..,xn:Dn):C1
pre:    e0
post:   e1
```
—————————————————————— OCL ——

with `excThrown('T1:TypeExp')`, ..., `excThrown('Tk:TypeExp')` occurring in the postcondition `e1` is translated into the dynamic logic proof obligation

—— KeY ——
```
 ==>
\forall T1 x1; .. \forall Tn xn;(
  x1.<created> & .. & xn.<created> & [e0]
  ->
```

---

[3] This feature is at the time of this writing only implemented in a simplified form in KeY.

```
\< boolean thrownT1 = false;
     :
     boolean thrownTk = false;
     try {C0::self.op(x1,..,xn);}
     catch (java.lang.Throwable exc) {
      thrownT1 = exc instanceof T1;
      :
      thrownTk = exc instanceof Tk; }
  \>
[e1]*
```
———————————————————————————————————— KeY ——

———— JAVA ————
```
/**
 * @postconditions self.x< 0 implies
 *                        excThrown('IllegalArgumentException')
 */
public void positive()
  throws IllegalArgumentException {
    if (this.x >= 0) { do something; }
    else { throw new IllegalArgumentException(); };
}
```
—————————————————————————————————— JAVA ——

Fig. 5.12. Postcondition referring to exceptions

Here `thrownTi` are new local Boolean program variables, whose names are composed of the string `Thrown` appended with the string `Ti`, and `[e1]*` arises from `[e1]` by replacing all occurrences of `excThrown('Ti:TypeExp')` by `thrownTi`.

Figure 5.12 contains a concrete example of a JAVA program with a postcondition that refers to exceptions. For a change we have used a different way to attach an OCL condition to an operation, i.e., by placing it as a specially tagged comment directly into JAVA code in front of the method it refers to. Here is the translated proof obligation in dynamic logic:

—— KeY ——
```
\< boolean thrownIllegalArgumentException = false;
   try { TrivialExc()::self.positive (); }
   catch (java.lang.Throwable thrownExc) {
     thrownIllegalArgumentException=thrownExc
     instanceof java.lang.IllegalArgumentException; }
\> (self.x < 0 -> thrownIllegalArgumentException = TRUE)
```
———————————————————————————————————— KeY ——

Another example program of the same kind is shown in Figure 5.13 with the following dynamic logic proof obligation:

```
─── KeY ───
 ==>
\<  boolean thrownIllegalArgumentException = false;
    try { TrivialExc1()::self.positive (); }
    catch (java.lang.Throwable thrownExc) {
       thrownIllegalArgumentException=thrownExc
         instanceof java.lang.IllegalArgumentException; }
\> thrownIllegalArgumentException != TRUE
                                              ─── KeY ───
```

```
─── Java ───
/**
 * @postconditions not excThrown('IllegalArgumentException')
 */
public void positive()
  throws IllegalArgumentException {
    try{
      if (this.x >= 0) { this.b = true; }
      else { throw new IllegalArgumentException(); }
    } catch (java.lang.Throwable exc) {
      this.b = false;
    }
  }
                                              ─── Java ───
```

Fig. 5.13. Another postcondition referring to exceptions

Miscellaneous

In the OCL constraints throughout this chapter, we frequently made use of the constant **null**. We extended OCL by assuming that any UML class diagram implicitly contains a class Null that is a subclass of every existing class in the diagram and whose only element is **null**. For all attributes attr that Null inherits the value of **null.attr** is undefined. Since OCL2.0 there is now an OCL type OclVoid that is the only instance of the metaclass VoidType and in turn contains as only element the object **null**. So, you could identify the types Null and OclVoid if you wished. We think of our solution as a first step towards defining an OCL profile for Java specification. We stick to it for the moment till the discussion on what is in general regarded as a null object has reached a consensus.

The OCL standard uses a three-valued logic to treat undefinedness. We deviate from this. In our logic all functions are total and undefinedness is handled by underspecification as explained in the sidebar 3.3.1 on page 90. See also Sect. refsect11:partmod.

If `a` and `b` are inherited attributes in class `Null` then in all snapshots `null.a` and `null.b` are defined, but we have no information on what the values are. Thus neither `null.a = null.b` nor `null.a != null.b` are valid. For a comparison of the various logical approaches to formalise undefinedness we recommend [Hähnle, 2005]. We want to emphasise that our semantics, defined by the translation into first-order logic faithfully models OCL in that an expression is undefined according to the OCL standard if and only if it is undefined in our translation semantics. The first difference is that we do not have an equivalent of the instance `invalid` which is the only element of the OCL type `OclInvalid`. The use of `invalid` can be easily avoided by using the query `oclIsInvalid()` on the the type `OclAny`. The second difference lies in the logic employed to deal with undefined statements. OCL uses a three-valued logic while KeY uses classical two-valued logic with underspecification.

5.3 Java Modeling Language

An increasingly popular specification language for Java projects is the *Java Modeling Language*, JML. Unlike UML the language is not standardised by an organisation like the OMG, the development is more a community effort lead by Gary T. Leavens, Iowa State University.[4] The nature of such a project entails that language details change, sometimes rapidly, over time and there is no ultimate reference for JML. Fortunately, for the items that we address in this introduction, the syntax and semantics are for the greatest part already settled in [Leavens et al., 2006]. Basic design decisions and extensive examples are described in [Leavens et al., 2003].

As the major difference to UML/OCL, JML focuses solely on the phases of software development in which source code is written. Moreover the only supported programming language is Java. JML talks *directly* about Java classes represented in source files. There is no need for separate UML class diagrams. Since specifications may also serve the purpose of documenting a program, it is most natural that specifications are directly *annotated* to the entities to which they refer. So if we have a class invariant for a Java class C, the JML representation of the invariant is directly written as comment (somewhat in the style of a Javadoc comment) into the class declaration of C. If it is not desired to include specifications into source code, it is also possible to add JML specifications in extra files, which contain copies of the source file signatures.

---

[4] See www.jmlspecs.org

The close integration of JML with Java allows one to use Java expressions, for instance the side-effect free boolean expression `atm.wrongPINCounter==0`, directly in invariants and operation contracts. This possibility makes writing specifications easily accessible for developers acquainted with Java. Moreover, JML is more easily adapted to the Java specific issues, such as abrupt termination.

In this section we start, as in the previous section, with some illuminating examples from our ATM scenario, before a more thorough introduction to JML's syntax and semantics follows.

5.3.1 JML by Example

Consider now a design phase in which concrete Java code has been written for a realisation of the ATM scenario. We assume that a Java class `ATM` is part of it. Immediately preceding its method `enterPIN`, the JML representation of the operation contract described in Sect. 5.1.1 is annotated as a comment starting with the symbol @. It has become customary to also end a JML comment with @ though this is not mandatory.

This can be seen in the listing in Fig. 5.14. At a first glance, we see that the JML specification from lines 9 to 40 contains three blocks, each starting with **public normal_behavior**. These blocks represent three operation contracts as introduced in Sect. 5.1.1. In JML terminology operation contracts are called *specification cases* while *contract* refers to the collection of all specification cases; we continue to stick with the term operation contract. JML annotations come together with visibility modifiers subject to the same rules as in Java. These have no bearing on the semantics, the meaning of a **public** contract is the same as that of a **private** contract. On the other hand visibility modifiers are in many cases helpful to formulate sensible contracts. JML adopts the principle that a **public** invariant is not allowed to talk about **private** fields.

The JML keyword `normal_behavior` states that the contract implicitly includes the requirement that the method *must* not throw an exception.

Let us look more closely at the third operation contract (lines 30 to 39). There are three keywords starting clauses that are terminated by a semicolon:

`requires` The condition following this keyword describes a precondition of the contract. More precisely, the conjunction of all these conditions forms the precondition of the operation contract. The expression following the first `requires` clause on line 10 resembles a Java expression, and its meaning is in fact that of a boolean Java expression. So this part of the precondition says that before calling `enterPIN` the `insertedCard` field must not be **null**, in order to ensure the assertions formalised in this contract. Alternatively, instead of expressing the precondition of the operation contract in separate clauses, one could have equivalently used the and-operator `&&` and written (replacing lines 31 to 34):

```
                  ── JAVA + JML ──────────────────────────
1   public class ATM {
2
3      private /*@ spec_public @*/
4              BankCard insertedCard = null;
5      private /*@ spec_public @*/
6              boolean  customerAuthenticated = false;
7
8
9      /*@ public normal_behavior
10         requires  insertedCard != null;
11         requires  !customerAuthenticated;
12         requires  pin == insertedCard.correctPIN;
13         assignable customerAuthenticated;
14         ensures   customerAuthenticated;
15
16         also
17
18         public normal_behavior
19         requires  insertedCard != null;
20         requires  !customerAuthenticated;
21         requires  pin != insertedCard.correctPIN;
22         requires  wrongPINCounter < 2;
23         assignable wrongPINCounter;
24         ensures   wrongPINCounter
25                       == \old(wrongPINCounter) + 1;
26         ensures   !customerAuthenticated;
27
28         also
29
30         public normal_behavior
31         requires  insertedCard != null;
32         requires  !customerAuthenticated;
33         requires  pin != insertedCard.correctPIN;
34         requires  wrongPINCounter >= 2;
35         assignable insertedCard, wrongPINCounter,
36                    insertedCard.invalid;
37         ensures   insertedCard == null;
38         ensures   \old(insertedCard).invalid;
39         ensures   !customerAuthenticated;
40      @*/
41   public void enterPIN (int pin) {
42     // here the implementation follows
                  ──────────────────────────── JAVA + JML ──
```

Fig. 5.14. A JML specification for enterPIN

```
——— JML (5.5) ———————————————————————————————
requires  insertedCard != null
          && !customerAuthenticated
          && pin != insertedCard.correctPIN
          && wrongPINCounter >= 2;
                                                    —— JML ——
```

ensures All boolean expressions of an operation contract following this keyword form (again in the sense of a conjunction) the postcondition of the contract. Our first example of a JML expression which is no JAVA expression turns up in line 38; here \old occurs. Keywords special to JML within expressions, like \old, start with a backslash. This one serves the same purpose as the @pre construct. Unlike @pre it refers to a whole expression. So \old(insertedCard) refers to the value of insertedCard before executing enterPIN. There are some subtle problems with this way of referring to pre states which we discuss later in Sect. 5.4.

assignable This keyword is followed by a list (items separated with a comma) of what is allowed to change during the execution of the method. JML does not allow temporary modifications of the specified location during the call deviating from the definition of modifies clauses in Sect. 3.7.4.

The JML contracts in Fig. 5.14 though marked **public** refer to the *private* field insertedCard. This is not a legal JML expression and any correct checker would reject it. To override the default we may declare a private JAVA field to be treated by the specification as if it were public by the annotation /*@ spec_public @*/. For the insertedCard field this was done in line 3 in Fig. 5.14. We could have omitted the keywords **public normal_behavior** because JML would assume them by default.

Clearly something is wrong if enterPIN is called but insertedCard is still equal to **null**. In the contracts we have seen so far the caller of the method is responsible to establish this precondition. If he does not, then no commitment is made. We could however decide otherwise and require that if the precondition is not met an exception of a type ATMException is thrown and no customer is authenticated. This could be specified in JML with the help of exceptional_behavior, a signals_only clause and a signals clause:

```
——— JAVA + JML (5.6) ———————————————————————————
  /*@ (* the contracts as defined above *)
   @ also public exceptional_behavior
   @    requires insertedCard==null;
   @    signals_only ATMException;
   @    signals (ATMException) !customerAuthenticated;
   @*/
  public void enterPIN (int pin) {
  // here the implementation follows
                                               —— JAVA + JML ——
```

The `signals_only` clause says that only exceptions of type `ATMException` must be thrown and the `signals` clause specifies that in the case of a thrown `ATMException` the `customerAuthenticated` field is set to **false**.

Another detail worth mentioning already here is the use of side-effect free and terminating methods in JML expressions. This is, as was the case with OCL, perfectly legal. Such methods are called *pure* in JML terminology and must be annotated with the keyword `/*@ pure @*/`. We could, e.g., add the following method, which is clearly pure, in class `ATM`:

```
—— JAVA + JML ——————————————————
  public /*@ pure @*/ boolean cardIsInserted() {
    return insertedCard!=null;
  }
                                  —— JAVA + JML ——
```

Now `cardIsInserted()` could replace `insertedCard != null` in all the contracts above.

The next example shows how invariants are written in JML. Again we want to formalise the property that different cards have different card numbers, compare the OCL constraint (5.2) on page 253. Clearly, this requires means that go beyond JAVA expressions. Universal quantification, syntactically quite similar to first-order logic, is used. The range of the quantification must only include the objects which are created. This can be achieved with the help of the expression `\created(o)`, which says that *o* is a created object. Since the resulting expression does not depend on *one* particular instance of `BankCard` it is referred to as a *static invariant*. The whole annotation to the class `BankCard` now reads:

```
—— JAVA + JML (5.7) ——————————————
public class BankCard {
  /*@ public static invariant
    @  (\forall BankCard p1, p2;
    @    \created(p1) && \created(p2);
    @    p1!=p2 ==> p1.cardNumber!=p2.cardNumber)
    @*/
    private /*@ spec_public @*/ int cardNumber;
    // rest of class follows
}
                                  —— JAVA + JML ——
```

Opposed to static invariants are *instance invariants*. They formalise properties of a particular instance, referred to by **this**. The OCL invariant (5.3) from page 254 on the class `CentralHost` reads in JML as follows:

```
——— Java + JML (5.8) ————————————————————
public class CentralHost {
  /*@ public instance invariant this.validCardsCount
    @                    == (\num_of BankCard p; !p.invalid)
    @*/
}
                                    ——— Java + JML ———
```

As in Java we could have skipped **this** in **this.validCardsCount**. An instance invariant contains an implicit universal quantification in that it requires that the stated property must evaluate to true *for all* created objects of its class.

We could use this to rewrite the JML static invariant (5.7) into an equivalent instance invariant:

```
——— Java + JML (5.9) ————————————————————
public class BankCard {
  /*@ public instance invariant
    @   (\forall BankCard p; this != p ==>
    @           this.cardNumber != p.cardNumber)
    @*/
    private /*@ spec_public @*/ int cardNumber;
    // rest of class follows
}
                                    ——— Java + JML ———
```

5.3.2 JML Expressions

Every Java expression according to Gosling et al. [2000] that does *not* include operators with side-effect, like e++, e--, ++e, --e, non-pure method invocation expressions, and assignment operators, is a JML expression. Any such expression e has a natural representation in KeY's first-order logic, which we denote by [e]. The JML reference manual [Leavens et al., 2006] does not contain a formal semantics of JML. The paper [Jacobs and Poll, 2001] roughly sketches a semantics of JML expressions in a higher-order logic that is a common abstraction of PVS and Isabelle/HOL.

The translation to first-order logic serves us as a precise definition of the meaning of JML expression. In Table 5.6, the mapping $e \rightsquigarrow$ [e] is defined for JML expressions e_0, e_1, and e_2.

For example, the JML expression

```
        insertedCard != null && !customerAuthenticated;
```

is translated as follows to first-order logic:

Table 5.6. Mapping from JML and Java expressions to FOL (selected items)

JML Expression	first-order logic formula
!$e0$	![e0]
$e0$ && $e1$	[e0] & [e1]
$e0$ || $e1$	[e0] | [e1]
$e0?e1:e2$	if [e0] then [e1] else [e2]
$e0$!= $e1$	!([e0] $\doteq$ [e1])
$e0$ >= $e1$	[e0] >= [e1]

$$!(o.\texttt{insertedCard} \doteq \texttt{null}) \,\&\, !\,o.\texttt{customerAuthenticated} \doteq \text{TRUE} \quad .$$

Note that this formula contains free occurrences of a variable o of type ATM, which is the **this** type the JML expression refers to.

Moreover JML introduces operators to express implication (==>) and logical equivalence (<==>).

Finally JML extends Java by *quantified* expressions. We have already seen an example of universal quantification at work in the JML annotation (5.7). Existential quantification works analogously. Table 5.7 summarises the first-order logic translations of these expressions. Note that quantifiers bind two expressions, the range predicate and the body expression with the semantics shown in the first-order logic column. A missing range predicate is by default **true**. Quantifiers are meant to range over all objects including the not yet created ones. This is in accordance with our definition of quantification in Sect. 3.3. In contrast to that, JML excludes **null** from the range of quantification.

Table 5.7. Mapping from new JML expressions to first-order logic (selected items)

JML Expression	first-order logic formula
$e0$ ==> $e1$	[e0] -> [e1]
$e0$ <==> $e1$	[e0] <-> [e1]
(\forall T e;$e0$;$e1$)	\forall T e; (([e] !$\doteq$ **null** & [e0]) -> [e1])
(\exists T e;$e0$;$e1$)	\exists T e; ([e] !$\doteq$ **null** & [e0] & [e1])

In addition to these traditional quantifiers JML offers so called generalised and numerical quantifiers. We have already seen the \num_of quantifier which delivers the number of values of its quantified variable for which the expression in the second argument is true. Other such quantifiers are \sum, \product, \min, and \max. Translations of these expressions have to be done similarly as for OCL (see Sect. 5.2.3).

More on the translation of JML expressions can be found in [Engel, 2005].

5.3.3 Operation Contracts in JML

We now turn our attention to operation contracts in JML. We have already encountered operation contracts starting with `normal_behavior` and `exceptional_behavior` in Figure 5.14. These are, in fact, special cases of a general contract concept starting with the keyword `behavior` which we discuss now.

An operation contract consists of a number of *clauses* each starting with one of the keywords `requires`, `assignable`, `ensures`, `diverges`, `signals`, or `signals_only`.

The boolean expressions following the `requires` clauses specify (seen as a conjunction) the preconditions of the operation contract. All other clauses must be true only under the provision that all `requires` clauses hold.

The postcondition of an operation contract is spread over the `ensures`, `signals`, and `signals_only` clauses. `ensures` describes the postcondition in the case of *normal* termination of the operation. That is, *if* the operation terminates normally then all the boolean expressions following `ensures` must hold. The `signals` clause specifies what happens if the operation terminates *abruptly*. `signals` is not directly followed by a JML expression. Instead there is first a declaration of an exception type T, and then a boolean JML expression e. If abrupt termination is caused by an exception of type T then e must be true in the post-state. Note that e does *not* specify the condition which triggers the specified expression to be thrown; such conditions can be stated in the `requires` clause of an operation contract. Finally `signals_only` lists the types of exceptions that may at most be thrown by a method. As we have done for JML expressions, we can define the meaning of a JML postcondition by translating them into the first-order fragment of JAVA CARD DL. The postcondition of a contract

---- JML ----
```
ensures   E;
signals (ET₁) S₁;
⋮
signals (ETₙ) Sₙ;
signals_only OT₁,…,OTₘ;
```
                                                            — JML —

is translated into

$$
\begin{aligned}
&(\; e \doteq \texttt{null} \rightarrow [E]) \\
&\& \; (\; e \in \; [ET_1] \doteq \text{TRUE} \rightarrow [S_1]) \\
&\qquad \cdots \\
&\& \; (\; e \in \; [ET_n] \doteq \text{TRUE} \rightarrow [S_n]) \\
&\& \; (\; e \in [OE_1] \doteq \text{TRUE} \; | \\
&\qquad \cdots \\
&\qquad | \; e \in [OE_m] \doteq \text{TRUE})
\end{aligned}
$$

We assume in this translation that the operation stores a thrown exception causing abrupt termination in the variable `e`. If the operation terminates normally then `e` equals `null`.

`assignable` is followed by a list of expressions which specify locations of the program. When these expressions are translated into our first-order logic, the top-level operator must be a non-rigid function symbol representing a field symbol or an array access. As special symbols we allow the JML expressions `\nothing` (which is equivalent to the empty modifies set) and `\everything` (which means that every location is allowed to be modified). The semantics of assignable clauses follows Definition 3.62. The `diverges` clause consists again of a boolean JML expression. It specifies the condition which must hold before calling the operation if the operation does *not* terminate. This sounds complicated but fortunately in practice and also as a matter of normalisation this can be reduced to two cases. As one case, we specify `diverges` **false**, then, in case of non-termination, **false** must have been satisfied before the operation call. This is never the case. Thus, `diverges` **false** *requires* the operation to terminate. On the other hand one could specify `diverges` **true**, then non-termination is always allowed. It is quite easy to figure out, that we can use appropriate `requires` clauses and these two incarnations of `diverges` to express all termination behaviour we may desire.

We can summarise the requirements imposed by an operation contract for an operation *op* as follows: When *op* is called in any state that satisfies all the `requires` clauses then:

- If *op* terminates normally then all `ensures` clauses are satisfied.
- If *op* terminates abruptly with an exception of type ET then
  - all `signals(`ET'`)` clauses for exception types ET' where ET is a subtype of ET' are satisfied and
  - there is a `signals_only(`ET''`)` clause such that ET is a subtype of ET''.
- If *op* terminates (either normally or abruptly) then at most the locations specified by `assignable` are modified compared to the pre-state.
- If *op* does not terminate, then the `diverges` condition has been true before calling *op*.

Figure 5.15 depicts the meaning of the special contracts `normal_behavior` and `exceptional_behavior` in terms of `behavior` contracts. Abbreviations, like the use of `normal_behavior` instead of a more verbose `behavior`, occur quite often in JML, and the process of resolving them is referred to as *desugaring*. Extending this scheme to specification cases with more than one occurrence of the different clauses can naturally be done.

Some JML operation contracts even have no `behavior`, `normal_behavior`, or `exceptional_behavior` header at all. Instead they start with clauses (like `requires`, `ensures`, etc.) directly. Such operation contracts are called *light-weight* in JML jargon. All others are called *heavyweight*. There is only a

```
normal_behavior                behavior
 requires R;                    requires R;
 assignable A;        ⟹        assignable A;
 ensures E;                     ensures E;
 diverges D;                    diverges D;
                                signals (Exception) false;


exceptional_behavior           behavior
 requires R;                    requires R;
 assignable A;                  assignable A;
 diverges D;          ⟹        ensures false;
 signals (ET) S;                diverges D;
 signals_only (OT);             signals (E) S;
                                signals_only (OT);
```

Fig. 5.15. Desugaring of `normal_behavior` and `exceptional_behavior`

small semantical difference of lightweight specifications compared to heavy-weight specifications starting with `behavior`. In lightweight specifications most missing clauses default to `\not_specified`, which leaves different JML tools different options to treat the missing items. In KeY, always the same defaults as for heavyweight specifications are used. See Table 5.8 for lightweight and heavyweight defaults. The choices correspond to Table 5.1 in Sect. 5.1.

Table 5.8. Defaults for missing JML clauses

Clause	Lightweight default	Heavyweight default
requires	`\not_specified`	**true**
assignable	`\not_specified`	`\everything`
ensures	`\not_specified`	**true**
diverges	**false**	**false**
signals	`\not_specified`	(Exception)**true**
signals_only	All exception types declared in the JAVA method declaration	

We have already seen in the introductory examples that, when describing post-states, one needs to refer to the state before the method invocation. The `ensures` and `signals` clauses describe post-states so that the JML expressions used in these clauses may include the `\old` construct.

With `/*@ pure @*/` annotations, implicit additions to all operation contracts are implied. This can again be seen as a de-sugaring. The operation contracts for an operation annotated with `/*@ pure @*/` are equivalent to adding `assignable \nothing` and `diverges false` to all operation contracts which are available for the constrained operation.

JML dictates a stricter rule of inheritance of operation contracts than required in Sect. 5.1. Every contract for a method automatically applies to overridden methods, too. Syntactically this is signified by the fact that contracts for overridden methods must start with `also`, the keyword which conjoins several contracts for an operation. The contract inheritance policy has the effect that all subtypes of a type T are behavioural subtypes (see Sect. 8.1.3) of T [Leavens and Dhara, 2000].

5.3.4 Invariants in JML

JML distinguishes two types of class invariants: instance invariants and static invariants.

An *instance invariant* is a boolean JML expression containing explicitly or implicitly the variable `this`. An instance invariant is satisfied in a program state if it always evaluates to true when the value of `this` ranges over all instances of its class. Syntactically, instance invariants are comments (as usual, starting and ending with @) which are explicitly marked with `instance invariant` or, if the targeted type is a class, just as `invariant`.

As illustrated in Sect. 5.3.2, we can translate boolean JML expressions into first-order logic formulae. The characteristic property of instance invariants is that there is a free variable in the resulting formulae. Consider the JML invariant (5.9) in Sect. 5.3.1. It could be represented as follows as first-order logic formula containing a program variable o of type `BankCard`:

```
---- KeY ----
\forall BankCard p; p.<created> = TRUE ->
    o != p -> o.cardNumber != p.cardNumber
                                        ---- KeY ----
```

The variable o is, according to the semantics of invariants, implicitly universally quantified over all *created* objects of the respective type. For a uniform treatment of invariants, we make this quantification explicit. We obtain closed formulae. If ϕ is the "raw" translation of a boolean JML expression in an invariant and if o is the occurring free variable of type T, then

$$\backslash\text{forall } T\ o;\ (o.\texttt{<created>} \doteq \texttt{true} \Rightarrow \phi)$$

is defined to be the translation of the JML invariant. The translation of our example yields:

```
---- KeY ----
\forall Bankcard o; \forall Bankcard p;
  ( o.<created>=TRUE & p.<created>=TRUE & o != p
    -> o.cardNumber != p.cardNumber )
                                        ---- KeY ----
```

According to Leavens et al. [2006], instance invariants defined in a class C must hold at any *visible state* for any object o of C. Visible states for an object o are the states reached when a method of o (this includes non-static methods and static methods declared in C or a super class) is invoked or finished or when a constructor of o is finished. A further visible state is when no method or constructor of o is in progress. The latter means that invariants must be established, according to JML, when in a method of o another method is called. JML thus requires invariants to hold at intermediate states of an operation. In Chapter 8 we will deviate from the visible state semantics of JML, since it is overly strong to require invariants to hold at intermediate states.

The semantics of invariants is liberalised by the possibility of JML to declare methods with `/*@ helper @*/`. It is not required that invariants hold at the entry and exit states of such helper methods.

Static invariants do not refer to a special instance of the class they are defined in. This implies that static invariants can only refer to instance fields via quantification as in the example of the static invariant in Sect. 5.3.1. We have seen there that it was in that case possible to replace it with an equivalent instance invariant (5.9). So are static invariants necessary at all? Imagine we want to express that the static integer field `maxAccountNumber` in class `CentralHost` is always greater or equal to 0. Then we want to require this condition even in states in which no object of `CentralHost` is created at all. So it is of no use to add an instance invariant

———— JAVA + JML ————

```
public class CentralHost {
 /*@ public instance invariant maxAccountNumber >= 0 @*/
//...
```
———————————————————— JAVA + JML ————

which would need to hold only after the constructor call of the first instance of this class is finished. The following *static* invariant

———— JAVA + JML ————

```
public class CentralHost {
 /*@ public static invariant maxAccountNumber >= 0 @*/
//...
```
———————————————————— JAVA + JML ————

must however hold already after the static initialisation of `CentralHost` has finished, which is the desired property.

Static invariants must be explicitly declared as `static` (as above) or they are written into an interface declaration and just start with `invariant`.

5.3.5 Model Fields and Model Methods

The operation contracts and instance invariants we have seen so far may only talk about instance (and static) fields occurring in the Java program they annotate. Since instance fields may only occur in classes and not in interfaces, how would we write operation contracts and instance invariants for *interfaces*?

In our banking scenario we could extend the simple `BankCard` class into a card which allows one to collect bonus points as well. Whenever certain transactions are done with the card, a counter `bankCardPoints` on the card is increased. We also foresee the situation that the bonus point system will be used with other cards from other vendors than our bank. It may thus be a good idea to separate the interface of accessing bonus points from the `BankCard` class. We use a Java interface `IBonusCard`, which `BankCard` implements. A Java interface is definitely the best choice since we do not want to provide implementations, as for instance in an abstract class, for the other vendors, just the mere interface:

——— Java ———
```
public interface IBonusCard {
  public void addBonus(int newBonusPoints);
}
```
——————————— Java ———

As already mentioned, we may wonder how to add a suitable specification, since there are no fields to talk about in a Java interface. Here JML model fields are the solution. We simply *assume* that a field representing bonus points was available. Let us call it `bonusPoints` of type `int`. Since it is not a true field and just for specification purposes, we add it (as usual in JML) as comment and qualified with the key word `model`. In specifications, as in the operation contract for `addBonus` this field may then be referred to:

——— Java + JML ———
```
public interface IBonusCard {
   /*@ public instance model int bonusPoints; @*/

   /*@ ensures bonusPoints == \old(bonusPoints)+newBonusPoints;
     @ assignable bonusPoints;
     @ */
   public void addBonus(int newBonusPoints);
}
```
——————————— Java + JML ———

The specification says that the bonus points are increased by the number given as argument in the method `addBonus`.

You may wonder how we can relate concrete implementations like that of
BankCard with model fields. Let us consider the implementation of addBonus
in BankCard:

─── JAVA ───
```
public class BankCard implements IBonusCard{
  /*@ public instance model int bonusPoints; @*/
  /*@ also
    @   assignable bankCardPoints;
    @*/
  public void addBonus(int newBonusPoints) {
    bankCardPoints+=newBonusPoints;
  }
}
```
─── JAVA ───

Since JML operation contracts are inherited, the contract in IBonusCard
is implicitly present at this method, but it specifies the change of field
bonusPoints not that of bankCardPoints as the implementation does. We
thus need to specify the relation between the concrete field and the model
field. In our case the relation is simple: bonusPoints *exactly* corresponds to
bankCardPoints; whenever we refer to bonusPoints in a specification, we
mean bankCardPoints in the implementation. This is how we denote this in
JML, added directly after the header of the class declaration:

─── JML (5.10) ───
```
/*@ private represents bonusPoints <- bankCardPoints; @*/
```
─── JML ───

The expression on the right side could in fact also be a more complicated
expression. If for some reason the points stored on the bank card are 100
times the points credited by the addBonus method we could write:

─── JML ───
```
/*@ private represents bonusPoints <- bankCardPoints * 100;
  @*/
```
─── JML ───

In our translation to first-order logic, we can simply replace every occurrence
of the model field with the expression
    The represents clauses so far are called *functional abstractions* since the
relation between model field and concrete field(s) is a function. There are
also *relational abstractions*

─── JML ───
```
/*@ represents x \such_that A(x); @*/
```
─── JML ───

which relate concrete fields with the model field `x`; the relation must satisfy the axiom `A(x)`. The functional abstraction (5.10) can thus also be expressed as relational abstraction:

```
──── JML ────────────────────────────────────
 /*@ private represents bonusPoints
               \such_that bonusPoints==bankCardPoints;
    @*/
───────────────────────────────────── JML ───
```

As Breunesse and Poll [2003] point out, the translation of model fields into a logical representation is non-trivial if $A(x)$ is not a function of x or if it is not a total function. KeY roughly follows one of the solutions in that paper: All occurrences of the model field in an expression are replaced by occurrences of a reference to a pure ("model") method m with no arguments and the same result type as the type of the model method. Method m is specified with an operation contract which (a) requires in its precondition that there is an x such that $A(x)$ holds, and (b) ensures that the result r of m satisfies $A(r)$.

5.3.6 Supporting Verification with Annotations

All JML annotations considered so far are obligations for verification: We are aiming to prove that the program satisfies the given specification. There are also other kinds of annotations which can be considered more as helpers for the verification process, such as loop invariants. For the program (3.1) on page 155 a loop invariant could be specified with JML as follows.

```
──── Java + JML ─────────────────────────────
m = a[0]; i = 1;
while (i < a.length) {
  /*@  ensures \forall integer x;0 != x && x < i ;a[x] <= m;
    @  assignable m, i;
    @ */
  if (a[i] < m) then
    m = a[i];
  i++;
}
─────────────────────────────── Java + JML ───
```

This example also shows the use of the assignable clause for loop bodies. This is at the time of this writing not a part of the official JML syntax, but is expected to be included soon.

5.4 Comparing OCL and JML

Advantages of OCL over JML:

1. OCL lives on a higher level of abstraction. A UML diagram can be annotated with OCL constraints before code is developed. Automatic generation of constraints from patterns (described in Chapter 6) as well as editing constraints parallel to natural language phrases (as detailed in Chapter 7) would be much harder if not impossible on code level.
2. As a consequence of the previous item, OCL is not committed to a particular programming language and better suited for model driven system development.
3. OCL is an OMG standard, though one has to admit that at the time of this writing the official standard draft still contains serious inconsistencies and many unfinished items.

Advantages of JML over OCL

1. JML is closer to Java code, which encourages its use by programmers and developers. In fact, today JML specifications are much more widespread than OCL specifications.
2. JML offers a greater variety of concepts on the implementation level, like exceptional behaviour, modifies clauses, and loop invariants.

JML is not standardised and its specification document is still very incomplete.

Referring to the Pre-state

It is a detail, but nevertheless instructive to compare the differences in referring to values in pre-states in OCL and JML, i.e., to compare OCL's `@pre` construct with JML's `\old`. The former can be attached to individual symbols while the second can only be applied to whole expressions. So, `o@pre.b@pre.c@pre`, `o.b@pre.c@pre`, `o.b.c@pre`, `o@pre.b.c@pre` are all legal OCL expressions while only `\old(o.b.c)` is allowed in JML, and would correspond to the first of the OCL expressions. The JML proponents argue that the explicit scoping of the `\old` construct make it easier to read. A more substantial difference is the fact, that the `@pre` construct is hard to implement for run-time checking in full generality. A drawback of the `\old` construct comes to the surface in the following specification problem. Suppose you want to state in a postcondition to a method `m` manipulating an array `a[]` and a field `idx` that the value `a[0]` equals the old value of the array at position `idx`. Now, `\old(a[idx])` would not do, since the value of `idx` in the pre-state would be used. We resorted to

```
 ─── JML ───
(\forall int x; x==idx; \old(a[x])==a[0]);
                                        ─── JML ───
```

On the other hand, one has to admit that OCL does not offer a built-in construct to model Java arrays. The sequence data type does not fit since it does not take into account that Java arrays are objects and also it declares operations, e.g., union or append, that do not make sense for arrays. As an extension of OCL we introduced functions `a.get(i)` and `a.length(i)` for array object `a` and integer `i`. The above discussed expression can now easily be written as `a.get@pre(i)`.

Modifies Clauses

Our semantics of the assignable or modifies clause deviates slightly from the semantics in JML. In the JML semantics, only the locations listed in the assignable clause can be assigned to during method execution. In the KeY semantics the locations contained in the modifier terms may be assigned to, it is only important that in the end the terms have the same value as before. We found no clear statement on OCL's position on the frame problem. The unofficial position seem to be that it is assumed that locations not contained in the postcondition cannot change. In [Baar, 2006] explicit extension of OCL to deal with the frame problem are proposed.

Range of Quantification

In JML, quantification extends over all elements of a given type and not only over all created or allocated elements. Since our logic uses the same semantics the static JML invariant (5.7) translates in

$$\forall p1.\forall p2.(p1 \mathrel{!\dot{=}} p2 \mathrel{-\!\!>} p1.cardNumber \mathrel{!\dot{=}} p2.cardNumber)$$

where $p1, p2$ are variables of type `BankCard`. The instance JML invariant (5.9) on the other hand translates to

$$\forall this.(this.\texttt{<created>} \mathrel{-\!\!>} \forall p.(this \mathrel{!\dot{=}} p \mathrel{-\!\!>}$$
$$this.cardNumber \mathrel{!\dot{=}} p.cardNumber))$$

This discrepancy is attributable to the fact that implicit quantification of the variable `this` is treated differently from explicitly quantified variables; they are only meant to range over existing elements.

For a not created BankCard `o`, the value of `o.cardNumber` should be undefined. The validity of the two formulae above now depends on how undefinedness is modelled. In our logic we model undefinedness by underspecification which would make both formulae invalid.

In our logic we express the intended invariant by

$$\forall p1.\forall p2.(p1.\texttt{<created>} \mathrel{\&} p2.\texttt{<created>} \mathrel{-\!\!>} (p1 \mathrel{!\dot{=}} p2 \mathrel{-\!\!>}$$
$$p1.cardNumber \mathrel{!\dot{=}} p2.cardNumber))$$

The JML community is at the time of this writing considering the introduction of an attribute similar to `<created>`.

The shown first-order formula is also the correct translation of the OCL constraint (5.2). The OCL method `A::allInstances()` returns the set of all existing instances of `A`.

The semantics in Appendix A of the OCL standard draft also distinguishes between existing elements and reservoir elements waiting to be created. But there seems to be no possibility to talk about these element in the language.

Integers

The following JML specification for the integer square root method can be found in [Leavens et. al., 2003]

—— JAVA + JML (5.11) ——

```
/*@ requires y >= 0;
  @ ensures
  @  \result * \result <= y &&
  @ y < (abs(\result)+1) * (abs(\result)+1);
  @ */
  public static int isqrt(int y)
```

———————————————————————— JAVA + JML ——

In [Chalin, 2003], the following flaw has been pointed out. For $y = 1$ and $\result = 1073741821 = \frac{1}{2}(max\_int - 5)$ the above postcondition is true, though we do not want 1073741821 to be a square root of 1. The problem arises since JML uses the JAVA semantics of integers which yields

$$1073741821 * 1073741821 = -2147483639$$
$$1073741822 * 1073741822 = 4$$

The findings in [Chalin, 2003] seem to indicate that programmers tend to have the mathematical integers in their minds and frequently make mistakes in JML specification. Chalin proposes the extension JMLa that includes a new primitive type `\bigint` of arbitrary precision integers, i.e., the mathematical integers.

The KeY system offers the option to choose between the mathematical and the JAVA semantics of integers ($\Rightarrow$ Chap. 12).

In OCL quantification over all integers is not possible. Its semantics only allows finite sets. The expression `Integer::allInstances() -> forAll(e)` is thus undefined.

6

Pattern-Driven Formal Specification

by

Richard Bubel
Reiner Hähnle

6.1 Introduction

There are few examples of innovations in software engineering that caught on as quickly and as pervasively as software design patterns [Gamma et al., 1995] did. Offering *reusable solutions for recurring software design problems*, software design patterns (from now simply called "patterns") proved to be attractive not only for software designers and developers: as academic teachers we often observe that patterns belong to a small number of methods that are immediately perceived as useful by most students. The pedagogical advantages of patterns are at least as big as the productivity gain.

Originating in urban architecture [Alexander, 1977, 1999], patterns turned out not only to be readily applicable in software design, but it was *only* there that they became mainstream technology. One has to surmise that there are reasons beyond the fact that patterns give *named solutions* to *recurring design problems* and discuss their *consequences*, as this would make them applicable in many areas where recurring design problems figure. And, in fact, patterns were tried out in a variety of settings, but not anywhere nearly as successfully as in software design. We believe the key reason for this phenomenon is that, in contrast to other areas, software patterns contain a mixture of informal and formal elements. The latter are typically expressed as structural diagrams in design languages (the pioneering book [Gamma et al., 1995] employed OMT, nowadays it is UML) and as schematic code (most often C++ or JAVA). It is a characteristic of software construction that solutions for design and code can immediately be used in production, once their applicability is realised. It is no coincidence that patterns became popular at the same time as formal notations for (object-oriented) design. Implementations of libraries of design patterns in object-oriented analysis and design (OOAD) CASE tools quickly followed.

In our opinion, patterns are among the very few methods that render themselves naturally to connect informal and formal models of software (even

if this is clearly not the intention of some pattern theorists, see [Coplien, 1996]). This intermediate status makes them interesting for solving one of the major problems discussed in this book: how to create formal specifications that are well-written and catch the intention of the designer. The first attempts to use patterns in formal specification appear in the *Bandera* project [Dwyer et al., 1998, 1999], where patterns are qualified regular expressions.

Most similar to our approach is the usage of specification patterns in the *Minerva/Hydra* project. They combine UML diagrams (class-, sequence and state diagrams) and schematic linear temporal logic formulas (similar to [Dwyer et al., 1998, 1999]) to specification patterns. In [Konrad et al., 2003] they report on a successful application on the specification of safety requirements for a Diesel Filter System. The coherence of specification and modelled systems is verified using the SPIN model checker.

Within the KeY project we have equipped a number of established design patterns (mainly from [Gamma et al., 1995]) with *schematic formal constraints* expressed in OCL [Baar et al., 2000]. These constraints formalise a part of the properties of the solution given by each pattern. The KeY system connects the schematic OCL constraints to the pattern library of the underlying case tool[1] so that they are instantiated and simplified automatically when the pattern that contains them is instantiated [Giese and Larsson, 2005]. We describe this process in Sections 6.3 and 6.4. There is a simple interface that permits the end user to extend the mechanism with his or her own patterns and constraints [Andersson, 2005]. This is sketched in Section 6.5.

It is also possible to take a view on pattern-driven specification, where the pattern is centered around formal specification. More precisely, the solution proposed in such a *specification pattern* is a formal specification. Structural design and code are merely collaborators than primary constituents of the solution. The justification for such *specification-centric* patterns is that in many specification scenarios neither structural design diagrams nor source code are adequate primary means of specification. The former give not enough details, while the latter is not generic enough. A typical example is the specification of operations that are in principle relational database queries, but which are not realised by using an explicit database. Usually these operations access internal data structure directly and the queries are hardcoded. This is a rather frequent scenario [Bubel and Hähnle, 2005]. The structural relationships are trivial while concrete code is too specific and not reusable. In the following section we further analyze the difficulties in formal specification and we exemplify our approach by defining a *database query* pattern.

---

[1] In addition KeY provides its own pattern library and can generate code that partially implements the provided patterns.

6.2 The *Database Query* Specification Pattern

In this section we describe a *specification pattern*. The pattern solves the problem of how to embed relational database query expressions into formal specification languages. We base our considerations on the formal specification language OCL ($\Rightarrow$ Sect. 5.2). The following subsection describes the *Database Query* Specification Pattern in detail. It is similarly structured as descriptions of Design Patterns [Gamma et al., 1995].

6.2.1 *Relational Database Query*

Intent

Make it possible for developers with less or none experience in writing formal specifications to formally specify relational database queries. Embed relational database query expressions succinctly in a formal specification language while avoiding domain-specific extensions.

Motivation

Consider the scenario from the banking application used throughout this book. Assume that the bank's marketing policy includes an upgrade to gold status of credit/debit cards associated to accounts with a transfer volume of at least € 5000 in the most recent accounting period. The affected clients have to be informed regularly.

   In order to determine the clients that are in line for gold status the bank's account database needs to be queried for all accounts that have the afore described property. This involves to sum up all transactions performed in the recent accounting period on a per-account basis and to select those accounts, where the sum is greater or equal to € 5000. For data security reasons, account holders (clients) and accounts are solely related by the account number. Therefore, both the transactions table (all transactions together with date, account number, and transferred amount) and the clients table (name, address, and account number) are involved in order to determine the gold status candidates.

   As relational databases are very common in many application areas, most developers are familiar (at least to a certain extent) with the *Standard Querying Language* (SQL) for relational databases and may come up easily with an SQL statement such as the following in order to realize part of the use case described in the scenario:

—— SQL (6.1) ————————————————————————————

```
SELECT name FROM clients
WHERE  accountnumber IN
  (SELECT   accountnumber FROM transactions t
```

```
WHERE Date.today - t.date <= period
GROUP BY accountnumber
HAVING   sum(t.amount) >= 5000);
```
── SQL ──

For writing a formal specification of our use case it is necessary to render the content of such an SQL expression in a formal specification language. This, however, poses serious problems for a number of reasons:

- Developers are not familiar with formal specification languages.
- Formal specification languages are developed for general purposes and lack specific features as provided by domain-specific languages.
- There is no or insufficient tool support for formal specification, for example, formal specification languages are not integrated into software development environments.

Our approach aims to address all three points: the developer may express the properties to be specified in domain-specific terms from which suitable OCL constraints are automatically generated. Developers need not to be OCL experts to specify (for example) complex database queries in OCL.

Applicability

Use the database query specification pattern, when:

- The specification of a property resembles or requires a complex database query.
- An SQL-like statement implementing the query would be easy to write.
- The queried database is explicitly modeled as part of the system, i.e., there is a small set of classes from which the structure of the corresponding relational database can be reconstructed.

The database query design pattern allows the user to generate OCL constraints for queries from an SQL-like terminology.

Structure

The database query pattern supports a small subset of the SQL SELECT statement. The supported small fragment includes

- nested selects of all or specific columns,
- aggregation functions like max and sum as part of the SELECT condition,
- row and group selection conditions WHERE and HAVING,
- the grouping of entries.

The pattern provides a generic OCL translation of SQL-like statements, which adhere to the following syntactic restrictions:

— SQL —

SELECT (* | $T1.column$ | $agg(T1.column)$)
FROM $table$ $T1$
(**WHERE** $boolean\_expression$)?
(**GROUP BY** $T1.column\_name$)?
(**HAVING** $boolean\_expression$)? ;

— SQL —

The selection conditions filter out entries that do not satisfy the Boolean expressions. Thereby the WHERE condition is evaluated *before* grouping takes place or the aggregation functions are computed, whereas the Boolean expression of the HAVING section is evaluated *after* these computations as it filters out complete groups and may therefore use aggregation functions. Please notice the necessity to provide an explicit label for each table in the FROM section and that references to columns in the SELECT statement have to be fully qualified. Otherwise, implicit references will always be interpreted as self references relative to the class context where the instantiated OCL constraints are placed.
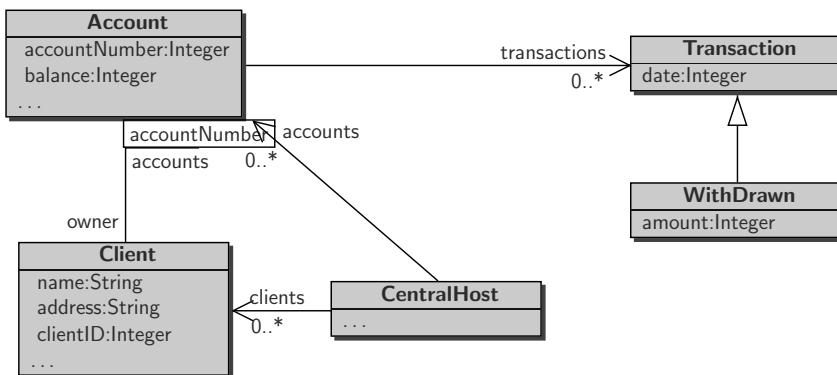
Fig. 6.1. Scenario gold card upgrade: UML class diagram (slightly simplified)

Participants

Database defines the context where to put table definitions and generated query constraints. Usually a type which allows for easy navigation to the required information (tables) is taken.

Table, **Entry** the context **Table** encapsulates a collection of **Entry** instances defining the content one is interested in (for example, **Entry**'s attribute values). Depending on the scenario one may identify the **Table** and **Database** contexts.

Implementation

Tables Database tables are represented as sequences of tuples. The type `TupleType` is new in OCL 2.0 and allows a natural representation of databases. The following schematic OCL constraint is a general and flexible template specifying a table's structure and content:

─── OCL ───
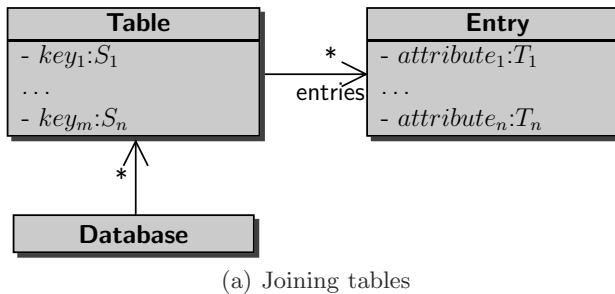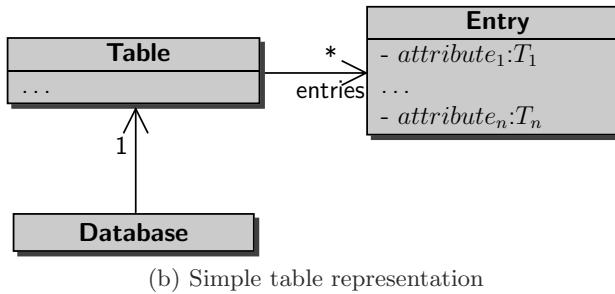
context *Database* **def**:
  tname: **Sequence(TupleType(**$col_1 : T_{col_1}$**,...,**$col_n : T_{col_n}$**))** =
      generator_expression

───────────────────────────── OCL ───

We are writing a pattern for formal specifications in OCL, so it is convenient to distinguish those parts of OCL expressions that can be modified during pattern instantiation from the ones that are fixed. Here and in the following we use italics as a typographic convention to denote OCL *schema variables*.

(a) Joining tables

(b) Simple table representation

Fig. 6.2. Standard idioms for database construction

For the schema variable *generator_expression*, a number of constructor templates are available that cover frequently encountered idioms in database construction, see Fig. 6.2. In Fig. 6.2(a) we depict the situation, where a database is constructed from a number of given *tables* only distinguished via

key attributes $k_1, \ldots, k_m$. The schematic constraint (6.2) defines a specific template for a generator expression that realises such a "join" in OCL:

—— OCL (6.2) ——

$tname$: **Sequence(TupleType**($k_1 : S_1, \ldots, k_m : S_m, a_1 : T_1, \ldots, a_n : T_n$)) =
   $tables$->**collect**(t:$Table$ |
       $entries$->**collect**(e:$Entry$ |
           Tuple{k_1 = t.k_1,...,k_m = t.k_m,
                   a_1 = e.a_1,...,a_n = e.a_n}))->**asSequence**()

———————————————————————— OCL ——

Example 6.1. In a video-controlled toll system the relationship between owners and cars is modeled as in Fig. 6.3. In order to send bills, the operating company needs to identify the owners of the observed cars. The problem is that the required information is fragmented between **Person** and **Car**, but with a defined relationship, namely the existence of an owner association between two instances. Their join yields a table that can be easily queried for the desired information:

—— OCL ——

carOwners: **Sequence(TupleType**(name:**String**, address:**String**,
                     licensePlate:**String**)) =
  registeredPersons->**collect**(p:Person |
    owns->**collect**(c:Car |
      Tuple{name = p.name,
             address = p.address,
             licensePlate = c.licensePlate}))->**asSequence**()

———————————————————————— OCL ——
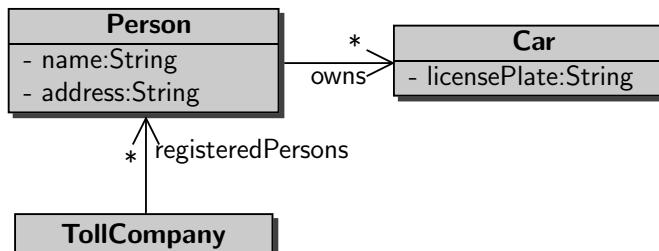
Fig. 6.3. Scenario: Video-controlled toll system

A frequent specialisation of this template is illustrated in Fig. 6.2(b). The schematic constraint (6.2) can then be simplified to:

—— OCL (6.3) ————————————————————————

$tname$: **Sequence(TupleType(**$attribute_1$:T_1,…,$attribute_n$:T_n**)) =**
   $entries$->**collect(**e:$Entry$ |
      Tuple{$attribute_1$ = e.$attribute_1$,
        …,
        $attribute_n$ = e.$attribute_n$**})**->**asSequence()**

——————————————————————————————— OCL ——

One difference between SQL rows and the OCL `TupleType` is that the latter permits identical names for different columns, whereas the former does not.

Simple Queries Our OCL templates so far enable us to define a database to be queried. It remains to model actual SQL queries as schematic OCL constraints. We do this in an incremental way. Whenever a schematic OCL constraint is instantiated it represents a concrete SQL query.

To get started let us look at how a simple SQL `SELECT` statement without aggregation functions or grouping can be expressed in schematic OCL:

—— OCL (6.4) ————————————————————————

`-- SELECT * FROM` $from$ `WHERE` $where$
$from$->**select(**$where$**)**
`-- SELECT` col_i `FROM` $from$ `WHERE` $where$
$from$->**select(**$where$**)**->**collect(**t | **Tuple{**col_i=$t.col_i$**})**

——————————————————————————————— OCL ——

The schema variable *from* denotes the queried table using variable t to iterate over the stored data.

Queries with Aggregation Now we add support for aggregation functions to our OCL template. Assume that *agg* is an SQL aggregate function such as `sum`, `max`, etc., and agg_{OCL} is its counterpart from the OCL standard collection library. The result is represented as a new sequence of tuples using the new aggregate function name agg_{new} as column identifier.

—— OCL (6.5) ————————————————————————

`-- SELECT` $agg(col_i)$ `FROM` $from$ `WHERE` $where$
Tuple{agg_{new} =
    $from$->**select(**$where$**)**->**collect(**t | $t.col_i$**)**->agg_{OCL}**()}**

——————————————————————————————— OCL ——

In general, an aggregate expression of kind '$agg(col_i)$' is translated as

—— OCL ————————————————————————————

Tuple{$agg_{newname}$ = $table$->**collect(**col_i**)**->agg_{OCL}**()}**

——————————————————————————————— OCL ——

Queries with Grouping Aggregate functions as described before are of limited use, because they apply to a specific column of the complete table. In our scenario we would not be able to calculate the transaction volume per account. The problem is that the table contains all transactions of all accounts, so if we sum up the withdrawal column we obtain the transferred volume of all accounts. In SQL, grouping is used to solve this problem. With grouping the user specifies a table column *col* to be used as a distinguishing aspect of a group. This means all rows with an identical entry in *col* are grouped together into one subtable. Applying an aggregate function on a grouped table then computes the result for each subtable rather than for the complete one.

First we define a family of helper functions $\texttt{groupBy\_}col_i\texttt{()}$ that take care of grouping a table *from* by its ith column:

──── OCL ────

```
context Database def:
groupBy_col_i(from:Sequence(TupleType(col_1:T_1,...,col_n:T_n))):
 Sequence(Sequence(TupleType(col_1:T_1,...,col_n:T_n))) =
    from->collectNested(e1 |
      from->iterate(e2; groupedRows:
        Sequence(TupleType(col_1:T_1,...,col_n:T_n)) = {e1} |
          if (e1.col_i = e2.col_i) then
            groupedRows->including(e2)
          else
            groupedRows
          endif))->asSequence()
```

──────────────────────────── OCL ────

The function $\texttt{groupBy\_}col_i\texttt{()}$ can now be applied to one of the select expressions of the previously given OCL templates. So let s_{OCL} be the OCL collection expression, $having_{OCL}$ the OCL translation of the HAVING expression, and e_{OCL} the OCL select expression of the first part of the query in the form "SELECT *s* FROM *from* WHERE *where*" as in (6.4) and (6.5). Then an SQL query with grouping can be translated to OCL according to the following template:

──── OCL (6.6) ────

```
-- SELECT s FROM from WHERE where GROUP BY col_i HAVING having
groupBy_col_i(e_OCL)->select(g| having_OCL)->collect(t| s_OCL)
```

──────────────────────────── OCL ────

Queries with Nested SELECT In order to increase readability we take a modular approach to nested inner SELECT expressions instead of inlining them. An inner SELECT is factored out as a standalone OCL definition. The definition declaration has one formal parameter for each variable visible in the scope of the inner SELECT. These are exactly the variables declared in FROM parts of outer SELECT statements.

6.2.2 Pattern Usage Example

We now demonstrate how to take advantage of the *Relational Database Query* pattern exemplified by the card upgrade scenario. Recall that the task was to compute those customers whose accumulated transactions in the recent accounting period exceeded a certain amount and who, therefore, are eligible for a credit card upgrade. Probably this information is used by several other operations, so it will be convenient to store it in an attribute `upgradeCandidates` with type `Sequence(TupleType(name:String))`. We want to specify the semantics of this attribute in OCL. The context of the OCL constraint will be the class `CentralHost` in Fig. 6.1, because (i) all information that is necessary to construct the tables can be retrieved by this class, and (ii) the attribute is implemented as a member of this or a closely connected class. The transactions and clients table result directly from instantiations of OCL template (6.2). The OCL constraints and the instantiation mappings needed to obtain them are shown in Fig. 6.4.

Based on the generated tables we continue with the translation of the innermost `SELECT` in the query (6.1). There is only one visible table declared in the outer `FROM` section, the `clientsTable` in Fig. 6.4. This table must become a parameter of the inner `SELECT` definition. In order to shorten the presentation we ignore the OCL translation defining the grouping operation for column `accountNumber`. The instantiation mapping is then as follows:

$$
map = \begin{cases}
e_{OCL} & \rightsquigarrow \texttt{transactionsTable->select(t | ...)} \\
s_{OCL} & \rightsquigarrow \texttt{g->collect(t | t.accountNumber)} \\
having_{OCL} & \rightsquigarrow \texttt{g->collect(t | t.amount)->sum() >= 5000} \\
& \cdots
\end{cases}
$$

When this mapping is applied to the template (6.6) one obtains the following OCL code (now including the grouping):

```
─── OCL ───────────────────────────────────────
context CentralHost def:
-- inner most select
computeVolume(c:TupleType(client:Client,accountNumber:Integer)):
 Sequence(TupleType(accountNumber:Integer)) =
  groupBy_accountNumber
   (transactionsTable->select(t | Date.today-t.date>=period))
      ->select(g | g->collect(t | t.amount)->sum()>=5000)
       ->collect(g |
        g->collect(t | Tuple{accountNumber=t.accountNumber}))
─────────────────────────────────────── OCL ───
```

It can be observed that the groups assume the role of tables in the translation of the select OCL expression s_{OCL}. The translation of the outer select is easier. It makes use of the inner select statement and is obtained from the

```
context CentralHost def:
  transactionsTable:Sequence(TupleType(
   accountNumber:Integer,date:Integer,amount:Integer)) =
    accounts->collect(acc:Account |
      acc.transactions->collect(trans:Withdrawn |
        Tuple{accountNumber = acc.accountNumber,
              date = trans.date, amount = trans.amount}))
```

$$
map = \begin{cases}
tname & \rightsquigarrow transactionsTable \\
tables & \rightsquigarrow accounts \\
Table & \rightsquigarrow Account \\
entries & \rightsquigarrow transactions \\
Entry & \rightsquigarrow Withdrawn \\
m & \rightsquigarrow 1 \\
k_1 & \rightsquigarrow accountNumber \\
S_1 & \rightsquigarrow Integer \\
n & \rightsquigarrow 2 \\
& \dots
\end{cases}
$$

(a) Transactions table generator with instantiation mapping

```
context CentralHost def:
  clientsTable:Sequence(TupleType(client:Client,
                                  accountNumber:Integer)) =
    clients->collect(owner:Client |
      owner.accounts->collect(accNr:Integer |
        Tuple{client = owner, accountNumber = accNr}))
```

(b) Clients table generator

Fig. 6.4. Instantiated table generators with the used instantiation mapping

template (6.4). The formal specification of attribute `upgradeCandiates` in terms of OCL is then:

─── OCL ───
```
context CentralHost def:
  upgradeCandiates:Sequence(TupleType(name:String)) =
    clientsTable->select(c |
      computeVolume(c)->includes(c.accountNumber))
        ->collect(t| Tuple{name=t.client.name})
```
─── OCL ───

The main drawback of this solution is that automatically generated OCL constraints are usually more complex than necessary. For this reason, the KeY tool features support for automatic simplification of OCL expressions. This is described in Section 6.4.

6.3 Specification Patterns

6.3.1 Format of Specification Patterns

As seen in Section 6.2.1, specification patterns follow a similar format as design patterns [Gamma et al., 1995, Coplien, 1996]. The most important section in a specification pattern are briefly explained in the following:

Intent Concise description of the pattern's objectives.

Motivation Explains where the need of this kind of pattern originates by means of a typical situation encountered when formally specifying a software system.

Applicability Clarifies the context conditions that have to be present for successful and useful pattern application. The *Relational Database Query* pattern, for example, is only useful when the database is explicitly modeled by classes in the system; it is likely to fail when the tables are actually stored in a relational database.

Structure Defines the minimal context required to specify the property of interest. As specification patterns are self-contained, all required types and symbols have to be declared in the pattern, except for predefined concepts of the specification language (here OCL). Consequently, a typical specification pattern for OCL/UML comes together with *template (class) diagrams* that declare all necessary classes, the members of these classes that occur in the pattern, as well as relations (associations) among these classes. Some properties of the pattern may be expressed on the UML level using multiplicities or stereotypes. If this is possible, it is preferable to OCL constraints, because it increases readability. In general, however, in order to specify more complex properties *schematic OCL constraints* must be provided such as (6.2)–(6.6). The following definitions make the notion of template diagram and schematic OCL constraint precise:

Definition 6.2 (Template Diagram). *A* template diagram *is a UML diagram describing the specification pattern's context. In the defining parts of the pattern only elements are allowed that either are declared in these diagrams (for example, types or attributes) or predefined in UML/OCL.*

Definition 6.3 (OCL Schema Variable). *An* OCL schema variable *is a placeholder within an OCL expression that may stand for (i) a local variable, (ii) a formal parameter, or (iii) the name of an operation within OCL definitions (*`def:` *clauses and* `let` *expressions). In addition, a schema variable may occur (iv) at any position within an OCL constraint where an instance of an* `OCLExpression` *is expected. The schema variable's type must conform to its usage, it must be the same throughout the constraint, and it must be either declared explicitly or it must be possible to infer it from the schema variable's occurrence.*

Definition 6.4 (Schematic OCL Constraint). *A* schematic OCL con-straint *is an OCL constraint whose context has to be given relative to a tem-plate class diagram. In contrast to standard OCL constraints, a schematic OCL constraint is allowed to contain OCL schema variables as replacement for all syntactical elements.*

Implementation This section of a specification pattern introduces and ex-plains all provided schematic OCL constraints. The context of a schematic OCL constraint must be a class or an operation declared in one of the tem-plate diagrams.

Example The example section of a pattern should illustrate the pattern's usage for a medium-sized example. In particular, common traps should be emphasised.

6.3.2 Application of Specification Patterns

Specification patterns are applied by mapping all schematic elements in the pattern to concrete counterparts within an existing UML/OCL model. There-fore, the modeler needs to provide an instantiation mapping defining the relationship between schematic and concrete elements.

Definition 6.5 (Pattern Application, Instantiation Mapping). *The process of embedding template diagrams and schematic OCL constraints into an existing UML model is called* pattern application *or* pattern instantia-tion. *The* instantiation mapping *map is the canonical continuation of two user-provided functions map$_{UML}$ and map$_{SV}$ that relate schematic diagram elements (types, attributes, role names, etc.), respectively, schema variables to their concrete counterparts.*

The KeY tool provides a graphical user interface that assists the modeler to enter the instantiation mapping to be used. Furthermore, concrete diagram elements that are the target of the instantiation mapping, but do not exist already, are created on-the-fly.

6.3.3 Other Pattern Usage Scenarios

The database query specification pattern is only one of several supported patterns. As in design pattern theory, we speak of an *idiom*, rather than of a pattern, when its complexity is low. Additionally, we distinguish be-tween proper specification patterns and conventional design patterns that were enhanced with schematic constraints capturing some of the forces and consequences of the pattern. A partial list of supported idioms and patterns is in Table 6.1.

6.4 Simplification of Pattern-Generated Constraints

OCL constraints obtained by translation from other domain-specific languages or by pattern instantiation of schematic templates (Def. 6.5) contain a certain amount of redundancy. The reason is that the original source has been written/designed for a more general scenario than the one it has been applied to. Once an OCL constraint has been instantiated for a concrete model, however, some redundancy can be removed using information from its specific model context such as the concrete classes including their inheritance relation and the concrete multiplicities of associations. Using the information in the concrete modelling context allows, for example, to evaluate guards of conditionals and to eliminate dead branches in OCL constraints. Additional possibilities for simplification include unwinding of iterator expressions over fixed finite collections as well as algebraic simplifications. These techniques are akin to those employed in optimising compilers. A detailed view on simplification and partial evaluation of OCL constraints and its realisation is in the paper [Giese and Larsson, 2005]. In the remainder of this section we demonstrate along an example, how OCL constraints can be simplified by partial evaluation.

The OCL constraints (6.4)–(6.6) specify the semantics of various SQL SELECT expressions, but it is left to the user (or to the CASE tool) to choose the appropriate template. It turns out that the reflexion capabilities of OCL are strong enough to write a generic template that works for all SQL expressions supported in the database pattern (Section 6.2.1). A generic template that can deal with plain *-arguments as well as with sum and column arguments of SELECT is as follows:

─── OCL ───

```
if (selectArgument = '*') then
    table->select(where)
else
    if (selectArgument = 'sum') then
      Tuple{sum=table->select(where)->
        collect(t | columnIdentifier)->sum()}
    else
      table->select(where)->
        collect(t | Tuple{columnIdentifier=t.columnIdentifier})
    endif
endif
```

─── OCL ───

This generic template is essentially a combination of (6.4) and (6.5). The first branch corresponds to the first constraint in (6.4), the second branch to (6.5), and the third branch to the second constraint in (6.4). A particular instantiation of schema variables could be *columnIdentifier* ⤳ price, *selectArgument* ⤳ 'sum' and *where* ⤳ Set{1,2,3}->forAll(s | s < 4):

Table 6.1. Additionally supported idioms and patterns

Idioms	*Specification enhanced design patterns*
Key Property	Abstract Factory
Field Value Restrictions	Observer
	Composite
	Singleton

———— OCL ————

```
if ('sum' = '*') then
  table->select(Set{1,2,3}->forAll(s | s < 4))
else
  if ('sum' = 'sum') then
    Tuple{sum=table->select(Set{1,2,3}->forAll(s | s < 4))->
      collect(t | t.price)->sum()}
  else
    table->select(Set{1,2,3}->forAll(s | s < 4))->
      collect(t | Tuple{price=t.price})
  endif
endif
```

———————————————————— OCL ——

This constraint offers several points for performing simplifications. The most obvious one is the evaluation of guards:

———— OCL ————

```
if (false) then ... else
  if (true) then
    Tuple{sum=table->select(Set{1,2,3}->forAll(s | s < 4))->
      collect(t| t.price)->sum()}
  else ... endif
endif
```

———————————————————— OCL ——

This simplification prepares the ground to eliminate dead branches of the `if`-conditionals. In addition, rules are applicable that match the body of an OCL select expression and rewrite the universal quantifier into an iterate expression [Giese and Larsson, 2005, Section 4.1]:

———— OCL ————

```
Tuple{sum = table->select
  (Set{1,2,3}->
    iterate(x; acc:Boolean = true | acc and x < 4))->
      collect(t | t.price)->sum()}
```

———————————————————— OCL ——

An `iterate` expression over a finite and concrete collection can be unwound by multiple applications of an `iterate` simplification rule, which executes the iterate expression on one element of the argument. The result of executing `iterate(x; acc:Boolean = true | acc and x < 4)` on `Set{1,2,3}` is:

—— OCL ——

```
true and 1<4 and 2<4 and 3<4
```

—————————————————————————————— OCL ——

After further simplification steps the original constraint is rewritten to a much more readable (and shorter) one that can be seen as a simplification of (6.5):

—— OCL ——

```
Tuple{sum = table->collect(t | t.price)->sum()}
```

—————————————————————————————— OCL ——

A prototypic implementation of OCL simplification is available as part of the KeY tool. The simplification of constraints is done by reusing the taclet mechanism (⇒ Chap. 4), where OCL constraints are represented as terms and simplification rules as taclets.

6.5 Support for Specification Patterns in KeY

In the previous sections we introduced and defined specification patterns, and we discussed the *relational database query* pattern in detail. In general, it is necessary to develop new domain-specific patterns. This leads to optimised division of work in a development process that employs formal methods: the (few) team members who have expertise in writing formal constraints implement patterns including OCL templates that the others merely need to instantiate. Empirical investigations indicate that it is realistic to assume that at least 25% of a formal specification on the design level can be obtain through pattern instantiation [Bubel and Hähnle, 2005].

In this section we describe how to implement new patterns using the pattern instantiation mechanism of the KeY tool. For more details, see [Andersson, 2005].

All interfaces and classes related to the pattern instantiation mechanism are part of package `de.uka.ilkd.key.casetool.patternimplementor`. Of particular interest are:

`AbstractPatternImplementor` This interface has to be implemented by all patterns. It declares methods to query for the specification pattern's name, the instantiable schema variables (therein called parameters), and the method that returns the instantiated template.

PIParameter and subclasses. A *pattern instantiation parameter* is uniquely identified by its internal name and typically is used to represent a schema variable as defined in Def. 6.3. As shown in Fig. 6.5 the **PIParameter** type hierarchy implements the composite pattern. This means parameters can be grouped together forming a **PIParameterGroup**.
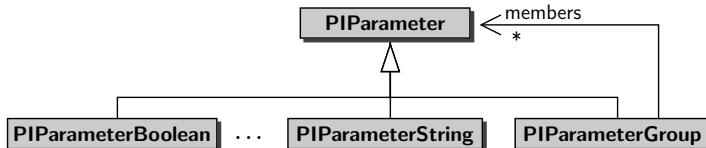
Fig. 6.5. PIParameter type hierarchy

ConstraintMechanism generates parameter groups from a textual template description and generates constraints defined as part of these descriptions with the entered parameter values.

The template language supported in this way allows to express schematic OCL constraints in a straightforward and natural manner. From a text file the template constraint as well as required parameters (schema variables) are read in and organised as a **PIParameterGroup**. The template constraint is automatically instantiated when applying the pattern. For more complex patterns the template language is not flexible enough, but for these cases the user can make use of the **PIParameter** directly.

We show how to implement a simplified version of the database query pattern. It suffices to create two new files. The first is the pattern's main class **SimpleDatabaseQueryPattern** (Fig. 6.8) that implements the interface **AbstractPatternImplementor**. The second contains the schematic OCL constraint to be instantiated (Fig. 6.6). The final result as laid out in the pattern instantiation panel for the user of the KeY tool can be seen in Fig. 6.7.

Some words on the concrete implementation of the pattern's main class (Fig. 6.8):

getName() returns the fully qualified name of the pattern. The qualifiers describe the category a pattern belongs to (here: Specification Pattern).

createParameters() creates the parameter group with all parameters for which an instantiation has to be provided by the user. In our example the parameter group is built up using

- the API directly, for example, for the database context and table definition generation;
- the template language mechanism: creation of an instance of class **ConstraintMechanism**, which is parameterised with the template description file name and the parameter group.

```
[Group] "databaseQuery", "Database Query"
 [String] "select", "SELECT", "t"
 [Boolean] "aggregate", "Aggregate", "false"
 [String]  "aggregateFunction", "Aggregate Function", "'sum'"
 ...
 [Context] Database
 [Definition]
  if '<:select:>' = '*' then
    <:from:>->select(<:fromIterator:>| <:where:>
  else
   if (<:aggregate:>) then
    if (<:aggregateFunction:> = 'sum') then
     Tuple{sum=<:from:>->select(<:fromIterator:>| <:where:>)->
       collect(<:fromIterator:>| <:fromIterator:>.<:select:>)->sum()}
     else ... endif
   else
     Tuple{<:select:>=<:from:>->select(<:fromIterator:>| <:where:>)->
       collect(<:fromIterator:>| <:fromIterator:>.<:select:>)}
     endif
   endif
[EndGroup]
```

Fig. 6.6. Template constraint description file

getParameters() (*not shown*) returns the created parameter group. The
    graphical user interface lays out the instantiation panel automatically
    as shown in Fig. 6.7.
getImplementation() returns the instantiated templates. In the first lines
    the template instantiation has been hard-coded by the pattern imple-
    mentor, while for the query instantiation itself the template language
    framework is used.

Almost self-explaining is the format of the template description file: A tem-
plate file is more or less a list of group definitions. Each group can contain a
number of parameter declarations, for example:

$$\text{[String]} \; internalName, \; displayName, \; defaultValue$$

The parameter declaration consists of an internal name used to refer unam-
biguously to the parameter (e.g. select), a display name (e.g. SELECT) and a
default value (e.g. *). The parameter declarations are followed by the context
definition ([Context]), where the instantiated schematic [Constraint] has
to be put. The schema variables occurring in the body of the constraint are
enclosed in <:...:>. Their identifiers must correspond to the internal names
of already declared parameters (either in the template or the pattern's main
class).
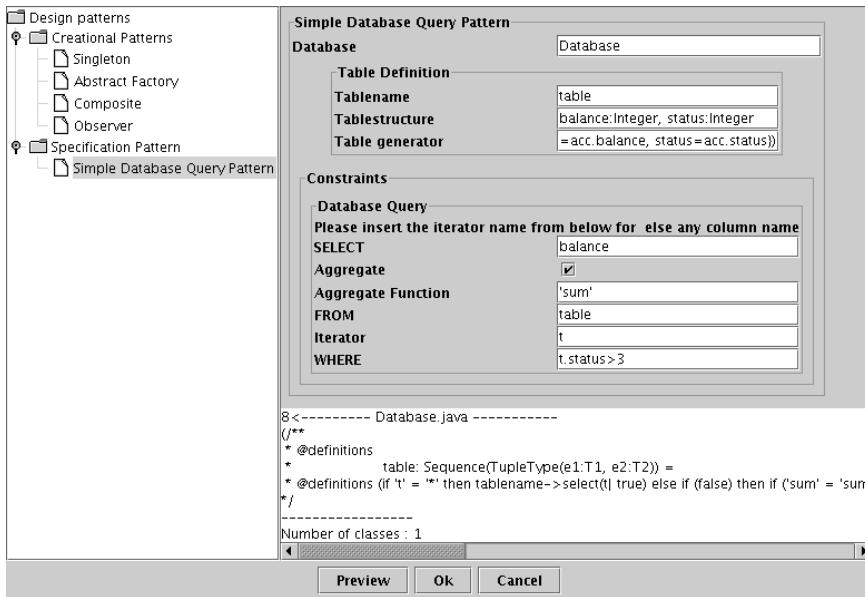
Fig. 6.7. Simple database query: instantiation window

6.6 Conclusion and Future Work

Patterns are a widely-used semi-formal concept for capturing good, reusable solutions for recurring design problems. There is little overhead involved in using them and the pedagogical gain is considerable. In this chapter we advocate the usage of patterns for the purpose of creating formal specifications. Methodologically, this is justified, because formal specifications are closely related to both design and code. Stressing the code aspect of formal specifications (precise contract-based semantics), it is natural to extend standard design patterns [Gamma et al., 1995] with constraint templates in addition to implementation templates that are already present. A library of such enriched patterns is part of the KeY tool [Andersson, 2005].

When viewing formal specification as a design problem (determining the right signature and relations among constituents), we write specification patterns in their own right. In their implementation part they contain again constraint templates. We exemplified the idea with a database query pattern. Completed by small-scale patterns, so-called specification idioms (Section 6.3.3), we obtain a whole range of pattern-driven mechanisms to arrive at well-written and sound formal specifications.

First experiences indicate that at least 25% of a formal specification on the design level can be obtain through pattern instantiation [Bubel and Hähnle, 2005]. As it is difficult to write good formal specifications, this is a considerable productivity gain. In addition, there is the advantage that users learn

─── JAVA ───────────────────────────────────────

```java
public String getName() {
    return "Specification␣Pattern:Simple␣Database␣Query␣Pattern";
}
private void createParameters() {
    paramGroup = new PIParameterGroup("simpleDatabaseQueryPattern",
                                "Simple␣Database␣Query␣Pattern");
    PIParameterString context =
        new PIParameterString("database", "Database", "Database");
    PIParameterGroup tableDefinitionGroup =
        new PIParameterGroup("tableDefinition", "Table␣Definition");
    ...
    /*
     * add parameter specifying the class used as database
     * to the pattern's parameter group
     */
    paramGroup.add(context);
    /* add table definition to the pattern's parameter group */
    paramGroup.add(tableDefinitionGroup);

    /* add query definition to the pattern's parameter group */
    oclTemplate = new ConstraintMechanism(oclTemplateFilename,
                                    paramGroup, this);
}
public SourceCode getImplementation() {
    ...
    // get table definition instantiations
    String tableName =
                paramGroup.get("tableIdentifier").getValue();
    ... // access further instantiations and
        // construct tblDef String

    // add definition of table
    src.add("␣*␣@definitions␣");
    src.add("␣*␣\t␣" + tblDef.toString());

    // add database query
    src.add(oclTemplate.getConstraints("␣*␣", "Database",
    database));
    ...
}
```

── JAVA ───

Fig. 6.8. Simple database query pattern implementation (excerpt)

how to specify from the solutions they see in the patterns. A further advantage of specification patterns is that they are a path to introduce domain-specific extensions into formal specification languages in a well-structured manner. For example, the database query pattern adds support for specifying relational database queries to OCL systematically and naturally.

Future Work

Throughout this chapter we focused on specification patterns tailored to UML/OCL. In the future, we intend to support JML specification patterns as well.

Making the template language more powerful is another direction to be aimed at in order to reduce the need to implement patterns in Java. One solution could be to incorporate Apache's velocity engine.

The KeY tool features an automatic translation module from OCL to natural language (described in Chapter 7). The quality of this translation depends on the availability of domain-specific hints (Section 7.5). Such hints could be attached to the OCL templates occurring in specification patterns.

Acknowledgements

7

Natural Language Specifications

by

Kristofer Johannisson

This chapter describes how to use the KeY tool to bridge the gap between formal and informal specifications. Specifications need to be understood, maintained and authored by people with varying levels of familiarity with a formal specification language such as OCL. While a user of the KeY theorem prover should know a formal specification language, we cannot expect the same from a typical software developer, manager or customer. Hence there is need for specifications of different levels of formality, and we need to keep these different versions synchronised.

The KeY tool addresses these problems by making it possible to automatically translate formal (OCL) specifications to natural language (English and German),[1] and by providing a multilingual editor in which specifications can be edited in OCL and natural language in parallel.

This chapter starts with an overview of the natural language features of KeY in Section 7.1. Sections 7.2 and 7.3 describe basic principles and components. The multilingual editor is described in Section 7.4. We outline how domain specific vocabulary is handled in Section 7.5, and conclude with pointers to further reading and a summary in Sections 7.6 and 7.7.

7.1 Feature Overview

This section gives an overview of the natural language features of the KeY tool. While the later sections give a more thorough description, this should give you an idea about what is possible to achieve, and what limitations there are.

7.1.1 Translating OCL to Natural Language

Using the KeY tool, it is possible to translate all OCL specifications in a Borland Together project to natural language. Fig. 7.1 shows an example

---

[1] As explained in Section 7.5, the support for German is limited.

English translation provided by KeY, based on the class diagram and OCL specifications in Fig. 7.2 and 7.3. To get an unbiased impression of what the KeY tool can do, the reader is encouraged to consider the English translation before reading the formal description provided by the class diagram and OCL specifications.

The translation in Fig. 7.1 is produced automatically, no user interaction is required (unless we want to customise the translation). The output is formatted, using either LaTeX (as shown here) or HTML.

Note that the structure of the natural language text is very similar to the structure of the OCL specification, and has the same level of abstraction. We get a direct translation of the OCL specification, not an informal explanation of what it means.

For translating the domain specific concepts from a class diagram (classes, attributes, operations and associations) we use some heuristics which often work well, but not always. For instance, translating juniorLimit as "junior limit" is probably fine, while for unsuccessfulOperations we may prefer "number of unsuccessful operations" rather than the default translation "unsuccessful operations". We therefore allow user customisation of the translation of domain specific concepts, as described in Section 7.5.

The OCL to natural language translation can be accomplished either from within the KeY tool, or by using stand-alone command line tools.

7.1.2 Multilingual Specification Editor

The KeY tool provides a multilingual, syntax-directed editor for editing of OCL and natural language specifications in parallel. The editor is started from the KeY submenu of the context menu of any class or operation in Borland Together. It allows the user to construct an abstract syntax tree of a specification (for instance an invariant of a class) by selecting alternatives from menus. The syntax tree is at all times presented to the user in both OCL and natural language.

Figure 7.4 shows an example editing session, where we have just started editing an invariant for the class PayCard. There are three main parts of the editor window: the syntax tree display (top left), the linearisation area (top right), and the refinements menu (bottom). The syntax tree display shows the abstract representation of the specification, while the linearisation area presents the specification in OCL and natural language (English and German). Unfinished parts of the specification—called goals, or metavariables—are shown as question marks. The refinements menu presents possible ways of filling in the goals. Basic editing proceeds by selecting a goal (by clicking in the text or in the tree) and a refinement (by choosing from the hierarchical refinements menu). Since the tree is presented in both OCL and natural language, knowledge of OCL is not required for using the editor.

Assume that we wish to complete the unfinished invariant in Fig. 7.4 into for instance `balance >= 0` (OCL) or "the balance is at least 0" (English).

For the operation **charge (amount : Integer)** of the class **PayCard** ,
given the following precondition :

- *amount* is greater than 0

then the following postcondition should hold :

- the balance is at least the previous value of the balance

For the operation **available () : Integer** of the class **PayCard** ,
the following postcondition should hold :

- the result is equal to the balance or the unsuccessful operations is greater than 3

For the class **PayCardJunior** the following invariant holds :

- the following conditions are true
    - the balance is at least 0
    - the balance is less than the junior limit
    - the junior limit is less than the limit

For the operation **createCard () : PayCardJunior** of the class **PayCardJunior** ,
the following postcondition should hold :

- the limit of the result is equal to 10

For the operation **charge (amount : Integer)** of the class **PayCardJunior** ,
given the following precondition :

- *amount* is greater than 0

then the following postcondition should hold :

- if the previous value of the balance plus *amount* is less than the junior limit then:
    - the balance is incremented by *amount*
    otherwise:
    - the balance does not change and the unsuccessful operations is incremented by 1

For the operation **checkSum (sum : Integer) : Integer** of the class **PayCardJunior**
,
the following postcondition should hold :

- if the result is equal to 1 then:
    - *sum* is less than the junior limit
    otherwise:
    - *sum* is at least the junior limit

For the operation **complexCharge (amount : Integer)** of the class **PayCardJunior**
,
given the following precondition :

- *amount* is greater than 0

then the following postcondition should hold:

- if the previous value of the balance plus *amount* is less than the limit then:
    - *amount* is equal to the balance minus the previous value of the balance
    otherwise:
    - the balance does not change and the unsuccessful operations is incremented by 1

Fig. 7.1. Example natural language translation of OCL constraints
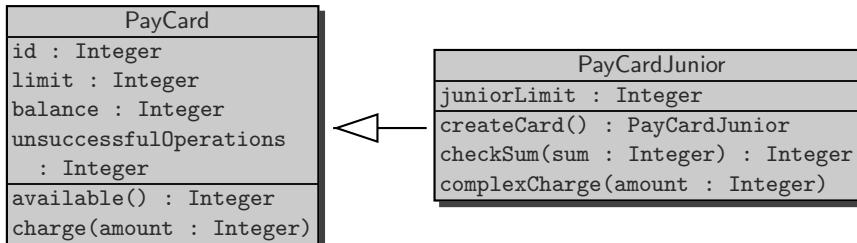
Fig. 7.2. Example class diagram

We would then proceed in a top-down fashion, first adding the comparison operator, and then the left and right argument to it. As shown in Fig. 7.4, the refinement "greater than or equal" is found in the submenu of "comparison operators". Figure 7.5 shows the editor after selecting this refinement.

We now have a specification `? >= ?` (OCL) or "? is at least ?" (English). Since the comparison operator takes two arguments, we have two new goals to fill in. In the figure, the leftmost goal has been selected. The refinements menu presents only type correct alternatives, which in this case means that we are only allowed to fill in instances of the OCL library type `Real` or any of its subtypes.

To complete the example, we have to fill in the left goal with `balance`, and the right with `0`, but we omit these steps here. The syntax editor is further explained in Section 7.4.

7.1.3 Suggested Use Cases

Translation of OCL to Natural Language

Being able to automatically translate OCL to natural language means that OCL specifications can be presented to people who do not know OCL. The translation can for instance be shown to a customer, who can then validate if it captures the desired behaviour of a system, or to a programmer who does not know OCL but needs to implement a system according to the specifications.

However, the provided natural language translations are on the same abstraction level as the original OCL specifications (as noted above). The intended reader of the translations must therefore be comfortable with this abstraction level. For instance, we cannot expect a translation of OCL specifications involving low-level implementation issues to be understandable to a customer.

The Multilingual Editor

The editor supports editing of OCL and natural language in parallel, and only allows the construction of specifications which are correct with respect

```
——— OCL ———————————————————————————————————————
context PayCard::charge(amount : Integer)
pre: amount > 0
post: balance >= balance@pre

context PayCard::available() : Integer
post: result = balance or unsuccessfulOperations > 3

context PayCardJunior
inv: self.balance >= 0 and self.balance < juniorLimit
     and juniorLimit < limit

context PayCardJunior::createCard() : PayCardJunior
post: result.limit = 10

context PayCardJunior::charge(amount : Integer)
pre: amount > 0
post: if balance@pre + amount < juniorLimit
      then balance = balance@pre + amount
      else balance = balance@pre and
        unsuccessfulOperations = unsuccessfulOperations@pre + 1
      endif

context PayCardJunior::checkSum(sum : Integer) : Integer
post: if result = 1 then sum < juniorLimit
          else sum >= juniorLimit endif

context PayCardJunior::complexCharge(amount : Integer)
pre: amount > 0
post: if balance@pre + amount < limit
      then amount = balance - balance@pre
      else balance = balance@pre and
        unsuccessfulOperations = unsuccessfulOperations@pre + 1
      endif
———————————————————————————————————————— OCL ———
```

Fig. 7.3. Example OCL constraints

to the OCL syntax and type system. It should therefore be useful for instance to a person who is not an OCL expert, but who needs to modify existing OCL specifications, as well as to people learning OCL.

For people who are already proficient in OCL, and who are not concerned with natural language translation, a traditional text editor is a more suitable tool for creating and modifying OCL specifications.
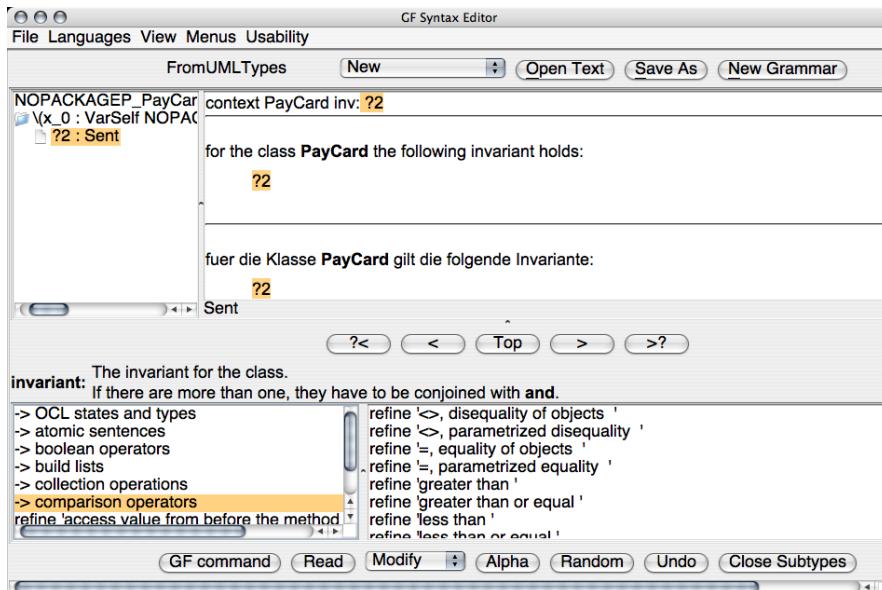
Fig. 7.4. Example editor session 1
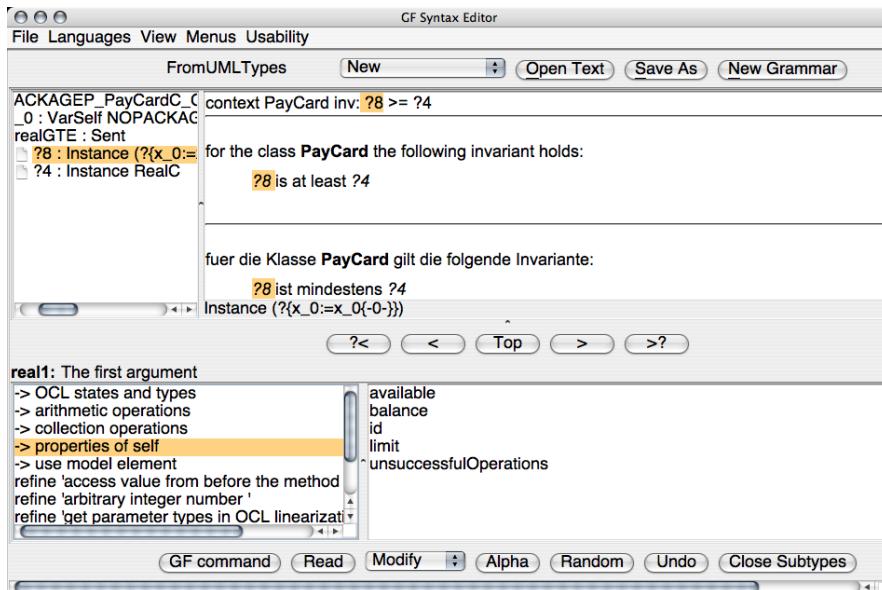
Fig. 7.5. Example editor session 2

OCL as Single Source

An important part of our approach is to use OCL as "single source": by creating and maintaining specifications in OCL (possibly using the multilingual editor), and then automatically translating them to natural language, we avoid the problem of having different versions of the same specification which need to be synchronised.

7.2 The Grammatical Framework

The natural language functionality in KeY is based on a multilingual grammar of specifications written in the Grammatical Framework (GF) formalism [Ranta, 2004].

A GF grammar defines abstract and concrete syntax. The abstract syntax gives rules for how to form abstract syntax trees. In a typical GF application grammar these trees are used as a non-linguistic, semantic representation of a restricted domain. In our case, we use abstract syntax trees to represent requirements specifications.

The concrete syntax defines how to present abstract syntax trees as expressions of a particular language, which can be a formal or a natural one. By having several concrete syntaxes for the same abstract syntax we get a multilingual grammar. We have defined concrete syntaxes for OCL, English, and German, which means that specifications represented in GF abstract syntax can be presented in these three languages.

The multilingual grammar for OCL, English and German specifications is written in the GF *formalism*. The GF *system* then provides functionality based on this grammar: it derives parsers and linearisers for the three languages as shown in Fig. 7.6. We can, for instance, parse an OCL specification (resulting in an abstract syntax tree) and then linearize it into English or German. Although we can also parse English or German specifications, the fragment of these languages described by our grammar is very small: we cannot expect to successfully parse arbitrary informal English or German specifications.

As noted above in Section 7.1.1, the structure of the natural language translation of an OCL specification provided by our tool is very similar to the structure of the original OCL specification. We can now explain the reason for this: the translation and the original specification both share the same abstract syntax, and the linearisation rules as defined by the concrete syntaxes for OCL, English and German cannot be arbitrarily complex. GF linearisation rules must be *compositional*, meaning that the linearisation of a tree is always expressed in terms of the *linearisation* of its subtrees, not the subtrees themselves.

An important aspect of our multilingual GF grammar is that it consists of a static as well as a dynamic part. The static part captures the OCL type system, basic OCL constructions such as invariants or if-then-else expressions,
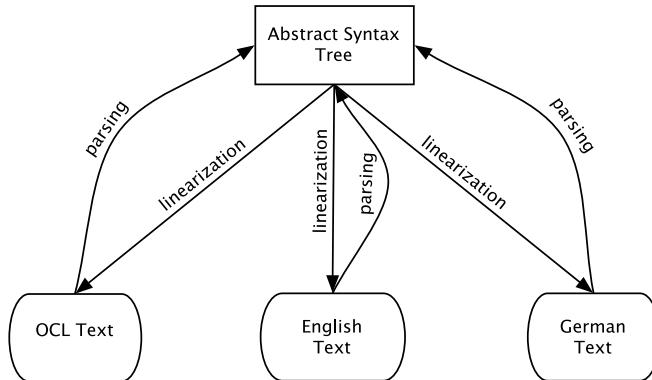
Fig. 7.6. GF parsing and linearisation

and the predefined types and operations of the OCL library. The dynamic part is a description of the domain specific concepts—classes, attributes, operations and associations—found in the class diagram of the current Borland Together project. This part of the GF grammar is generated from the current class diagram. Section 7.5 describes the basics of this generation, and how it can be customised.

7.2.1 GF Examples

To illustrate the general principles of GF we give some examples of abstract and concrete syntax rules, loosely based on our multilingual GF grammar (without explaining all the details of the GF formalism).

In the abstract syntax, we want to represent the domain of OCL specifications, for instance, we have to represent classes, expressions and queries. The following is one way to do this in GF abstract syntax:

——— GF ———

```
cat Class;
cat Expr (c:Class);
fun IntegerC : Class;
fun maxQ : (x,y : Expr IntegerC) -> Expr IntegerC;
fun intLit : Int -> Expr IntegerC;
```

——————————————————— GF ———

This defines two categories `Class` and `Expr`: If *c* is a `Class`, then `Expr` *c* represents expressions of type *c* (`Expr` is a *dependent type*, since it requires an argument). Using these two categories, we can then introduce the GF functions `IntegerC` and `maxQ` to represent the OCL library class `Integer`, and the query `max` (which returns the maximum of two integers). The function `intLit` allows us to use the built-in integer type of GF for integer literals.

The concrete syntax then gives rules for how to linearize abstract syntax trees in OCL, English, or German. Here we consider some examples for English. Writing GF concrete syntax is much like working in a functional programming language with record-types, strings, and finite algebraic data types. We must provide a record type for each category in the abstract syntax, and a function building records of the correct type for each abstract syntax function.

To express that a class in OCL corresponds to a noun in English we use the following concrete syntax:

```
── GF ──────────────────────────────────────────────────
param Number = Sg | Pl;
lincat Class = {s : Number => Str};
lin IntegerC = {s = table {Sg => "integer"; Pl => "integers"}};
                                                           GF ──
```

Here we introduce a parameter type for representing singular and plural number. The `lincat` judgement states that the category `Class` corresponds to records containing a field `s`, which is a string inflected in number (a finite function from `Number` to `Str`). Then, `IntegerC` is linearised as an inflection table with the singular and plural form of the noun "integer".

To linearize an abstract tree ($maxQ\ x\ y$), where x and y have type `Expr IntegerC`, as an English noun phrase "the maximum of x and y", we give the following rules to complete our small GF grammar (we also need to include a linearisation for the `intLit` function):

```
── GF ──────────────────────────────────────────────────
lincat Expr = {s : Str};
lin maxQ x y = {s = "the" ++ "maximum" ++ "of" ++ x.s ++
                    "and" ++ y.s};
lin intLit i = {s = i.s};
                                                           GF ──
```

Loading this grammar into the GF system, we can then parse for example the string "the maximum of 2 and 7", which gives us the abstract syntax tree ($maxQ\ (intLit\ 2)\ (intLit\ 7)$).

When writing larger GF application grammars, such as the one used to link OCL and natural language, you normally work on a higher level of abstraction than in these small examples. Instead of defining your own types for nouns and number (or for gender and case, as we would need for German), you make use of the resource grammar library which is supplied with the GF system. This library provides an interface of linguistically motivated types (e.g., types for number, gender, nouns, verbs and sentences) and functions (e.g., for building a sentence from a verb phrase and a noun phrase). Implementations of the interface are provided for several languages. By making

use of the resource library interface we can therefore share code between the English and German concrete syntax in our multilingual grammar.

7.3 System Overview

There are a number of components involved in linking OCL to natural language: a multilingual GF grammar, the GF system, a syntax-directed editor, a GF grammar generator taking class diagrams as input, and also a stand-alone OCL parser and typechecker. Fig. 7.7 shows how these components relate to each other in terms of input and output.
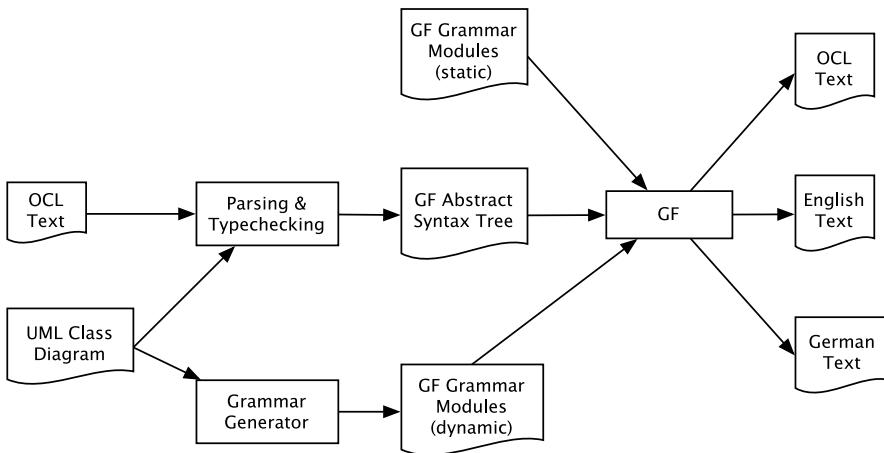
Fig. 7.7. System components

Grammar Generation

All functionality relies on the existence of the GF grammar for specifications, and as described in Section 7.2 above, parts of this grammar are dynamically generated from a class diagram. The class diagram is in turn extracted from Borland Together.

OCL Parsing and Typechecking

When translating an OCL specification to natural language, or when starting the multilingual editor for a given OCL specification, the first step is to turn the OCL text into a GF abstract syntax tree. To do this, we are not using the parser automatically derived by GF, but a custom parser and typechecker. Note that typechecking OCL requires also the class diagram as input.

There are a number of reasons for using a custom parser and typechecker: we need to work around a limitation in the parser derived by GF for our particular grammar, it makes it simpler to deal with all the various implicit forms in OCL concrete syntax, and it also makes it possible to give better error messages when encountering type errors. Finally, we expect the external parser, which is derived using a standard context-free parser generator, to be more efficient than the GF parser when parsing large specifications.

GF

The input to GF is the grammar (static and dynamic parts) and an abstract syntax tree. To translate OCL to natural language, the tree is then just linearised into English and German. In case of the editor, the user manipulates the syntax tree in the editor, while viewing the result in OCL, English and German in parallel.

7.4 The Multilingual Editor

The multilingual editor allows you to edit specifications in OCL, English and German in parallel. It is based on the generic GF syntax editor [Khegai et al., 2003] but has been customised for the domain of software specifications [Daniels, 2005]. The editor is started from the KeY submenu of the context menu of any class or operation in Borland Together. If the class or operation is already annotated with an OCL specification, it is parsed and shown in the editor, otherwise the editor starts up with an empty invariant (for classes) or with empty pre- and postconditions (for operations). The editor is intended for editing the OCL specification of one class or operation at a time.

7.4.1 Syntax-Directed Editing

The editor is syntax-directed: editing consists of manipulating the abstract syntax tree of a specification, rather than a string of characters as in a typical text editor. The tree is at all times presented in OCL, English and German, as defined by the GF grammar for specifications (the user can choose which languages to show). Since we are editing a syntax tree, we can only construct syntactically correct specifications. The editor also includes a type system and ensures that the syntax tree is always type-correct.

There are two basic ways of manipulating a syntax tree in the editor: *refinement* (top-down editing) and *wrapping* (bottom-up).

7.4.2 Top-Down Editing: Refinement

Refinement consist of selecting a goal—an unfinished part of the tree, displayed as a question mark—and filling in this goal by selecting a refinement

from a menu. The selected refinement may in turn contain new goals which need to be filled in.

Each goal has a type, and the refinements menu only lists refinements of this type. A type can for instance be "integer expressions", "sentences", or "attributes". The types and refinements available are given by the underlying GF grammar.

We consider the example from the beginning of this chapter again, as shown in Fig. 7.8. In the upper left part of the editor window we see the abstract syntax tree of a specification, which is presented in OCL and natural language in the upper right part of the window. There are two unfinished parts (goals), one for each argument to the comparison operator (>=).
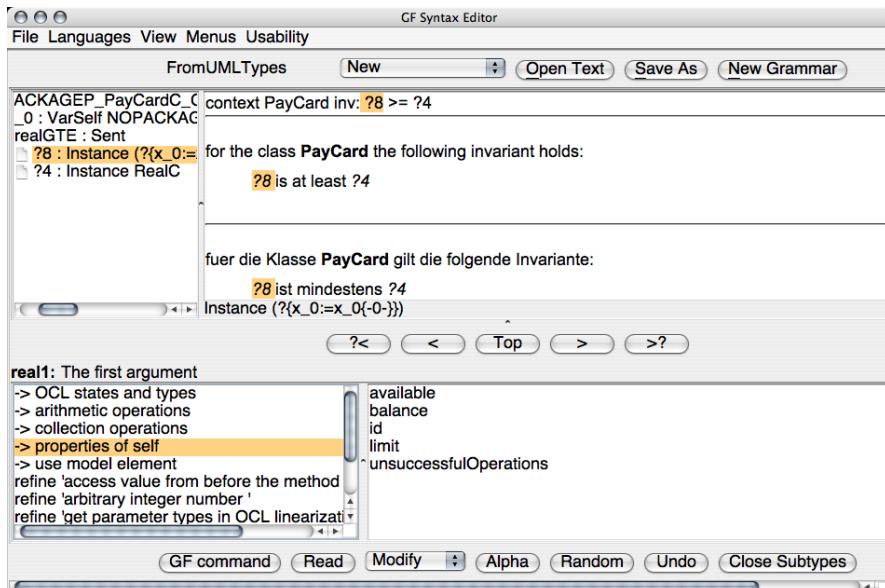
Fig. 7.8. Editing by refinement

7.4.3 Bottom-Up Editing: Wrapping

Wrapping consists of selecting any part of the syntax tree—with or without unfinished parts—and replacing it with a new construction, which contains the previously selected subtree as a part. For instance, if we have constructed the invariant `self.balance >= 0`, and would like to add that `balance` should also be smaller than `limit`, we do this by wrapping it using `and`. The first step is to select the subtree corresponding to `self.balance >= 0`, as shown in Fig. 7.9.
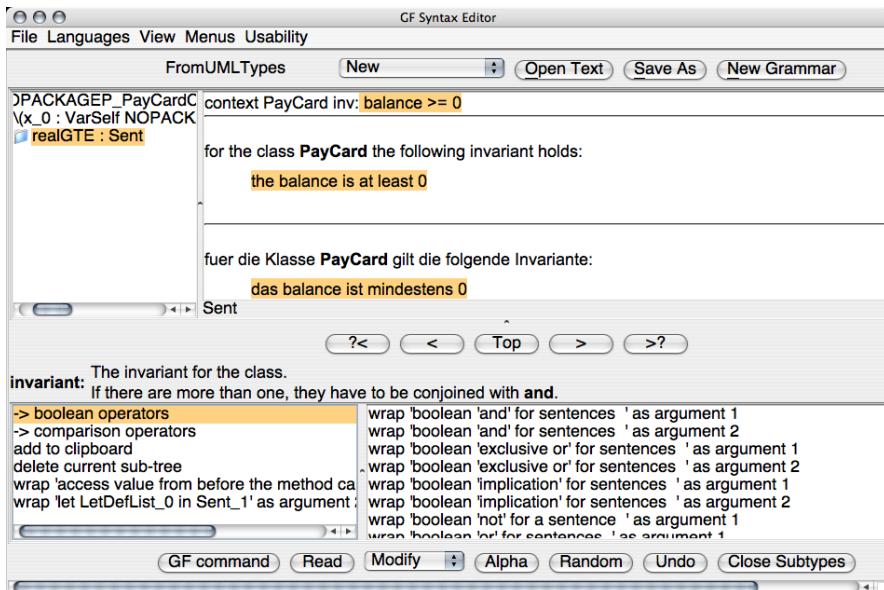
Fig. 7.9. Editing by wrapping, step 1

The current selection is now a sentence. Since **and** is a construction which takes two sentences into a new sentence, we can wrap the current selection using **and**. This is done by selecting "wrap boolean 'and' for sentences as argument 1" in the refinements menu. The result is shown in Fig. 7.10: the previously selected subtree `balance >= 0` has now been wrapped as the first argument to **and**, resulting in `balance >= 0 and ?`.

7.4.4 Other Editor Features

The editor also includes other features, for instance, as you would expect there is a clipboard for copying and pasting syntax trees, as well as an undo command. Another feature is refinement by parsing: instead of filling in a goal by selecting a refinement, one can enter a text string. The string is then parsed and (if parsing was successful) the goal is filled in with the resulting syntax tree. In this case, it is the parser derived by GF which is being used, not the custom OCL parser and typechecker.

7.4.5 Expressions and Sentences

The editor makes a distinction between expressions and sentences. Expressions are instances of any of the classes from the class diagram, or of the OCL library types such as Integer or Boolean. Sentences are used to express invariants, pre- and postconditions. An example expression is `self.balance`
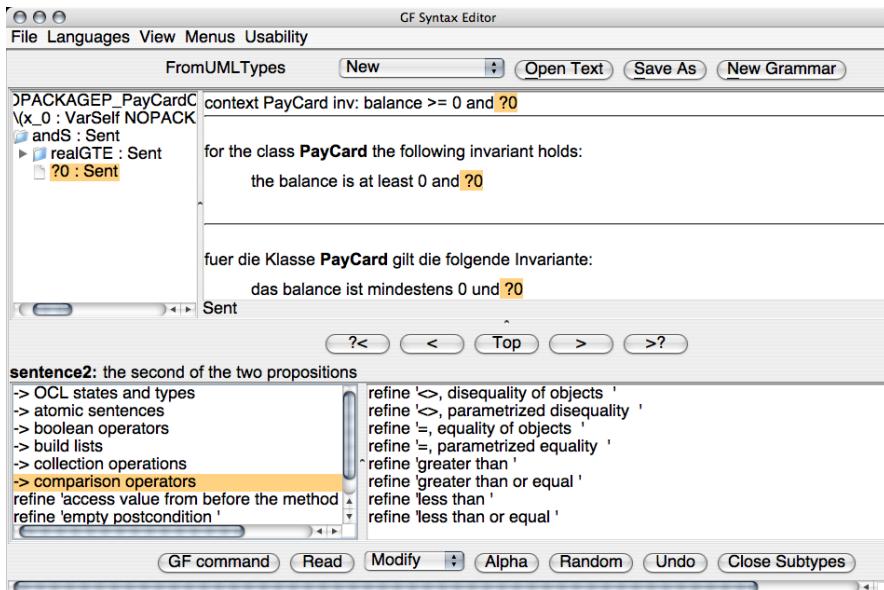
Fig. 7.10. Editing by wrapping, step 2

("the balance"), an example sentence is `self.balance >= 0` ("the balance is at least 0"). We mention this distinction since it is not present in OCL itself: there is no concept of sentences in the OCL language specification. Instead, expressions of type Boolean are used for invariants, pre- and postconditions. However, in the editor expressions and sentences are two different types: goals of expression type cannot be filled in with a sentence, and vice versa.

All OCL library operations as well as all domain specific attributes and operations which return Boolean from the point of view of OCL are considered as sentences in the editor. It is always possible to convert a sentence into a Boolean expression, but this has to be done explicitly.

7.4.6 Subtyping

The OCL type system includes subtyping: wherever an expression of a type T is expected, we can also use an expression of type T' as long as T' is a subtype of T. For instance, the OCL comparison operators <, >, <=, and >= are all defined for the class Real. However, since Integer is a subtype of Real, we can also use them to compare integers.

GF has no built-in notion of subtyping. In the GF grammars for specifications, this problem is solved by including explicit coercions (typecasts). These coercions are part of the abstract syntax tree, but are not visible in the OCL or natural language rendering of the tree. The editor usually creates these coercions automatically without requiring user interaction, but sometimes—

in particular when an existing specification is modified—the user has to be aware of the coercions.

7.5 Translation of Domain Specific Concepts

As previously mentioned, the translation of domain specific concepts is defined by GF grammar modules which are generated from the class diagram of the current project in Borland Together. This generation is based on some simple rules described below. If the automatically derived translation is not appropriate, it can be customised by hand.

The generation and customisation are both based on the assumption that the language used in class diagrams is English, and that OCL specifications are to be translated into English. The generated GF modules can be used with the German GF grammar anyway, but the resulting German contains fragments of the English used in the class diagram (as seen in the syntax editing examples in Section 7.4).

7.5.1 Grammar Generation

The grammar generation provides default translations for the concepts—classes, attributes, operations, and associations—in a class diagram. Currently, this generation is based on a few simple rules:

- Classes are treated as common nouns, or as common noun phrases. In case the name of the class is capitalised (as in PublicKey), it is split into separate words, where the last word is considered as a noun which is modified by the other words. For instance, a class Person is treated as a common noun "person", while a class PublicKey is treated as a common noun phrase "public key".
- Properties (attributes, operations and associations) are treated as noun phrases, except for Boolean properties, which are treated as sentences. Capitalization is used also for properties, e.g., an attribute juniorLimit is translated as the noun phrase "junior limit". Boolean properties which start with "is-", e.g., isEmpty or isValidated, are treated as adjectives (e.g., "... is empty", "... is validated").

7.5.2 Customising the Translation

If the translation provided by the generated grammar modules is not appropriate, it can be customised by hand. We plan to make it possible to perform such customisation by having the user add annotations to the Borland Together class diagram, but at present there is no such functionality. To

customise the translation, one must instead modify the generated GF grammar files directly. However, as described below, this can be done without requiring GF expertise.

Customisation is done on the level of concrete syntax. The generated concrete syntax makes use of a grammar-level API, which contains functions for common constructions. This API abstracts from the complexity of the rest of the grammar. To modify the generated concrete syntax it is therefore enough to have an understanding of the API, it is not necessary to be a GF expert.

This API is described in detail on the web site for the OCL-Natural Language tool ($\Rightarrow$ Sect. 7.6), here we just consider a small example. As mentioned in the previous example in Section 7.1.1, the default translation of the unsuccessfulOperations attribute of the PayCard class is "unsuccessful operations", although "number of unsuccessful operations" might be a more natural translation. The generated GF concrete syntax for unsuccessfulOperations is the following:

```
───── GF ─────
lin unsuccessfulOperations = mkSimpleProperty (adjCN
        ["unsuccessful"] ((strCN ["operations"])));
                                          ───── GF ─────
```

The left hand side of this linearisation judgement is simply the name of the construction in the abstract syntax which represents unsuccesfulOperations. The right hand side gives the linearisation of this construction, expressed using the functions mkSimpleProperty, adjCN and strCN of the grammar API.

This generated linearisation can be changed to produce "the number of unsuccessful operations" instead by using the ofCN and strCN functions:

```
───── GF ─────
lin unsuccessfulOperations = mkSimpleProperty (ofCN
    (strCN "number") (adjCN ["unsuccessful"] ((strCN
    ["operations"]))));
                                          ───── GF ─────
```

7.6 Further Reading

The basic motivations and design principles of a GF based tool to link OCL and natural language are described in a paper by Hähnle et al. [2002]. A later paper shows that the tool scales well enough to handle a case study: translating OCL specifications of the JAVA CARD API to natural language [Burke and Johannisson, 2005]. There is also a web site for the tool.[2]

---

[2] http://www.key-project.org/oclnl/

7.7 Summary

The KeY tool makes it possible for people who are not OCL experts to create and maintain OCL specifications, by providing a multilingual, syntax-directed editor in which specifications can be edited in OCL and natural language in parallel. OCL specifications can also be translated to natural language independently of the editor, which enables people who have no knowledge of OCL to make use of formal specifications.

A limitation is that the provided natural language translation has roughly the same structure and level of abstraction as the original OCL specification. In this sense, we do not provide informal explanations of formal specifications. Also, automatic formalisation of arbitrary informal specifications falls outside the scope of the KeY tool.

The natural language tools are built around a multilingual Grammatical Framework grammar for specifications in OCL, English and German. The translation of domain-specific concepts can be customised on the grammar level.

8

Proof Obligations

by
Andreas Roth

This chapter deals with the question how we can prove properties of specifications and of the relations between specifications and programs. The most important instance of such a property is the correctness of a program with respect to its specification.

In Chapter 5 we discussed specifications expressed with the languages UML/OCL and JML and their translations into the first-order fragment of JAVA CARD DL. We now present our answers to the questions we left open there. What is the role we want class invariants to play? In which states should they hold and how do we prove this? What is the relation between postconditions and invariants?

We formulate a series of *proof obligations templates*. These contain parameters that can be instantiated with a specification or parts of a specification to yield *proof obligations*. These are finite sets of JAVA CARD DL formulae that can be submitted to the KeY prover.

In the following we denote by P the program under investigation and by Spec a fixed specification of P. Furthermore, we refer to the common set of methods and constructors of a program as its *operations* ($\Rightarrow$ Sect. 5.3).

Building on the translations from Chapter 5 we assume that the elements of Spec are given by first-order JAVA CARD DL formulae. More specifically Spec consists of

- The set Inv_{Spec} of all formulae that result from the translation of *invariants* for P. These are closed formulae. Remember, that instance invariants ϕ with an implicitly universally quantified variable o for the object that was constrained (that is, the `self` or `this` object) are translated in the closed form

$$\backslash\mathrm{forall}\ T\ o;\ (o.\texttt{<created>} \doteq \mathrm{TRUE} \mathbin{-\!\!>} \phi)\ .$$

  In this chapter we often say, a bit sloppy, that this *formula* is an invariant.
- The set of *operation contracts* each consisting of
  - A method or constructor declaration op in a class or interface $C \in P$.

- A precondition *Pre* as first-order logic formula. This formula may contain program variables for the receiver object, this is the object which a caller invokes the method on, and for the parameters. If *op* refers to a static method or a constructor the receiver object variable is not allowed.
- A postcondition *Post* as first-order logic formula containing program variables for the receiver object, for the parameters, for the returned value, and for the thrown exception. The latter two are optional if the method's thrown exception or return type is irrelevant or non-existent. The receiver object variable is again not allowed for static methods.
- A modifier set

$$Mod = \{ \langle \phi^1, f^1(t_1^1 \ldots, t_{n_1}^1) \rangle, \ldots, \langle \phi^k, f^k(t_1^k \ldots, t_{n_k}^k) \rangle \}$$

  as the translation of the modifies or assignable clause, see Def. 3.61 in Sect. 3.7.4. Each $l_i = f^i(t_1^i \ldots, t_{n_i}^i)$ may contain program variables for the receiver object (if existent) and for the parameters,
- It is indicated whether the operation must terminate. Technically, a marker from $\{partial, total\}$ is provided. The marker *total* is set if and only if the operation contract requires the method or constructor to terminate, otherwise *partial* is set.

From time to time we explicitly list the program variables in *Pre*, *Post*, $l_1, \ldots, l_n$ by adding their names in parentheses in the following order: receiver object, parameters, return value, thrown exception object. For instance we could write

$$Post(\texttt{self}; \texttt{p}_1, \ldots, \texttt{p}_n; \texttt{result}; \texttt{exc}) \ .$$

We use *opct* to denote an operation contract comprising all the mentioned items.

Structure of this Chapter

We distinguish between *vertical proof obligations*, also called *program correctness proof obligations*, that relate programs and specifications and *horizontal proof obligations* or *design validation proof obligations* which concern properties of specifications only. We start with the latter in Sect. 8.1 on design validation. The rest of the chapter is devoted to vertical verification.

As we said above, we offer a series of proof obligation templates, that a user may chose from to suit his needs and fit into his development methodology. Yet, we adopt a particular notion of program correctness, that we have dubbed *observed state correctness*, as a basis for possible variations. This will be explained in Sect. 8.2 together with some other general assumptions.

We use the attribute *lightweight* to refer to proof obligations that address interesting or important parts of invariants and operation contracts without any claim to overall correctness. This is the topic of Sect. 8.3. In Sect. 8.4 we

present a first approach of how lightweight proof obligations may be combined to establish the correctness of whole programs.

The proof obligations considered so far apply to completed programs. Adding new classes to it or importing it into a new context would necessitate to redo all previous verifications. In Sect. 8.5 we investigate ways to overcome this deficiency and arrive at modular specification and verification. As it turns out these modularisation techniques may also be used to improve the verification of closed programs considered in Sect. 8.4.

Note on Notation

When we speak of states in this chapter we refer to states as defined in Def. 3.18 on page 88. Thus states are first-order models. We will frequently use S, S_i to denote states, i.e., $S = (\mathcal{D}, \delta, \mathcal{I})$. For function symbols f in the signature we use $S(f)$ as a short hand for the interpretation $\mathcal{I}(f)$ of f in S.

8.1 Design Validation

It is commonplace that the later in software development design flaws or bugs are uncovered, the more expensive they are to fix. It is thus desirable that formal verification starts as early as possible. In this section we propose proof obligation templates at the design phase where no code is yet written. Of course, we cannot expect statements on the correctness of the still to be written program, but we can get valuable hints on inconsistencies or deficiencies of the design.

8.1.1 Disjoint Preconditions

Consider the following JML specification:

```
──── JAVA + JML ──────────────────────────
 /*@ normal_behavior
   @    requires !customerAuthenticated
   @             && pin != insertedCard.correctPIN
   @             && wrongPINCounter >= 2;
   @    ensures \old(insertedCard).invalid;
   @ also exceptional_behavior
   @    requires insertedCard==null;
   @    signals (ATMException) !customerAuthenticated;
   @*/
 public void enterPIN (int pin) {
   // here the implementation follows
                                    ──── JAVA + JML ────
```

This example is a slight variation of our example in Sect. 5.3.1, but there can in fact be no correct implementation of this specification at all! Assume a state in which no costumer is authenticated, (`!customerAuthenticated`), the entered PIN is incorrect (`pin != insertedCard.correctPIN`) and there have been at least two earlier unsuccessful attempts to enter the PIN, (`wrongPINCounter >= 2`). Assume further that there is no card inserted in the ATM (`insertedCard==null`). Both preconditions are satisfied in this state, so both postconditions must hold after having executed `enterPIN` in that state. The first one says that the method must terminate normally, the second says that it must terminate abruptly! Clearly there is no implementation which can fulfil both operation contracts. In the original formulation (⇒ Sect. 5.3.1) we had `insertedCard==null` as an additional precondition in the first operation contract, which made the two preconditions not overlap.

   We would thus like to check whether contracts with overlapping preconditions have consistent postconditions. Unfortunately, this cannot be expressed in first-order logic in general. The best we can do is to check whether preconditions are disjoint.

Proof Obligation Template 1

DisjointPre(opct₁, opct₂):

for two operation contracts $opct_1$ and $opct_2$ with preconditions Pre_1 and Pre_2 using the same program variables for the receiver object and the arguments

$$! Pre_1 \mid ! Pre_2$$

If we can prove this formula to be universally valid, we can be sure that there is no state in which the preconditions of both operation contracts hold. The instantiation of this template with the preconditions from the above example is obviously not provable. The instantiation with the preconditions from the specifications 5.5 and 5.6 from Sect. 5.3.1 yields (for simplicity we skip some unimportant conjuncts):

$$!(\texttt{atm.insertedCard} \,!\dot{=}\, \texttt{null} \;\&\; \texttt{customerAuthenticated} \,!\dot{=}\, \text{TRUE})$$
$$\mid \texttt{atm.insertedCard} \,!\dot{=}\, \texttt{null}$$

This formula is universally valid, the preconditions are thus never valid in the same state.

8.1.2 Behavioural Subtyping of Invariants

According to Liskov and Wing [1994], a class D is a *subtype* of a class C if all properties on instances of C hold also for instances of D. This property is usually referred to as Liskov's principle or as *behavioural subtyping* (⇒ Sect. 1.4). Since we agreed that invariants be inherited by subclasses (⇒ Sect. 5.1.2) Liskov's principle for invariants is automatically satisfied. For instance invariants, which we view as universally quantified formulae of the shape

$$\forall\, C\ o;\ (o.\texttt{<created>} \doteq \text{TRUE} \rightarrow \phi(o))\ .$$

Liskov's principle is already an immediate consequence of our semantics of typed first-order logic ($\Rightarrow$ Chap. 2). In any first-order structure class D is interpreted as a subset of the interpretation of class C. So if $\phi(o)$ holds true for all instances o of D it is also trivially true for all instances o of C.

The following template makes thus only sense for users that do not subscribe to automatic inheritance of invariants. A possible argument in favour of this position might be that specifications should be as *local* as possible. To understand the invariant of a class it should not be required to retrieve the invariants for all its superclasses. In this case the invariant of a class D could syntactically look completely different from the invariant of its superclass C.

Proof Obligation Template 2
BehaviouralSubtypingInv(C, D):

$$\forall\, D\ x;\ \big(\,\mathrm{Conj}_D \rightarrow \mathrm{Conj}_C\,\big)$$

where the formulae Conj_C *and* Conj_D *are the conjunction of all formulae* ϕ *where* $\forall\, x\ C;\ \phi$ *and* $\forall\, x\ D;\ \phi$ *are invariants. If C or D contain no invariant then* Conj_C *(or* Conj_D*) equals* true.

Example 8.1. In Sect. 7.1.1 we already encountered the following invariant of class `PayCardJunior` (translated to first-order logic):

$$\texttt{self.balance} >= 0\ \&\ \texttt{self.balance} < \texttt{PayCardJunior.juniorLimit}$$
$$\&\ \texttt{PayCardJunior.juniorLimit} < \texttt{self.limit}$$

and the invariant of its superclass `PayCard`:

$$\texttt{self.balance} >= 0\ \&\ \texttt{self.balance} < \texttt{self.limit}\ .$$

Instantiating *BehaviouralSubtypingInv* with these invariants we we obtain:

$$\big(\texttt{self.balance} >= 0\ \&\ \texttt{self.balance} < \texttt{PayCardJunior.juniorLimit}$$
$$\&\ \texttt{PayCardJunior.juniorLimit} < \texttt{self.limit}\big)$$
$$\rightarrow \texttt{self.balance} >= 0\ \&\ \texttt{self.balance} < \texttt{self.limit}$$

The validity of this formula can easily be proved with KeY.

8.1.3 Behavioural Subtyping of Operations

Let us again consider a subclass D of a class C. This time we focus on an operation *op* and its operation contract *opct$_C$* with precondition *Pre$_C$* and postcondition *Post$_C$*. We assume that *op* is overridden in D now with

a contract $opct_D$ with precondition Pre_D and postcondition $Post_D$. Liskov's principle requires that any implementation of op in D satisfying its contract Pre_D also satisfies the contract $opct_C$. This requirement may be broken down into the following obligations:

- *Weaken preconditions in subclasses: Pre_D must be weaker than Pre_C.* Again we use an implication to express this property: $Pre_C \rightarrow Pre_D$. This condition ensures that D's implementation produces a well-defined result at least in those states C's implementation may be safely called.
- *Strengthen postconditions in subclasses: $Post_D$ must be stronger than $Post_C$.* In terms of formulae this can be expressed by a logical implication: $Post_D \rightarrow Post_C$. If this condition is established we can be sure that D's implementation produces at least the results C's implementation does.

These two formulae could be used to formulate a proof obligation template right away. But, let us think for a moment if we cannot do better. The second formula expressing that postconditions in subclasses are stronger than in superclasses ignores the fact that both *preconditions* are true before invoking the operations. Can we get extra milage out of this? To help shape our ideas consider the following specification of a method `m` which returns the result `result` of the division of two (static) attributes `a` and `b`:

$$Pre_C : \texttt{b} \mathbin{!\dot{=}} 0 \qquad\qquad Pre_D : \text{true}$$
$$Post_C : \texttt{result} \dot{=} \texttt{a/b} \qquad Post_D : (\texttt{b} \dot{=} 0 \rightarrow \texttt{result} \dot{=} 0)$$
$$\& \ (\texttt{b} \mathbin{!\dot{=}} 0 \rightarrow \texttt{result} \dot{=} \texttt{a/b})$$

In this example the formula expressing strengthening of postconditions instantiates to

$$(\texttt{b} \dot{=} 0 \rightarrow \texttt{result} \dot{=} 0) \ \& \ (\texttt{b} \mathbin{!\dot{=}} 0 \rightarrow \texttt{result} \dot{=} \texttt{a/b}) \rightarrow \texttt{result} \dot{=} \texttt{a/b} \ .$$

One can easily figure out that this proof obligation is not valid in states satisfying $\texttt{b} = 0$. Let us, as a first attempt, just add the precondition that `b` is not equal to 0:

$$\texttt{b} \mathbin{!\dot{=}} 0 \rightarrow \big((\texttt{b} \dot{=} 0 \rightarrow \texttt{result} \dot{=} 0) \ \& \ (\texttt{b} \mathbin{!\dot{=}} 0 \rightarrow \texttt{result} \dot{=} \texttt{a/b})$$
$$\rightarrow \texttt{result} \dot{=} \texttt{a/b}\big) \quad (8.1)$$

This formula is valid. But, does it guarantee Liskov's principle? Consider the following JAVA implementation in `D`:

──── JAVA (8.1) ────

```
public int m() {
  b--;
  return (b==0) ? 0 : a/b;
}
```
──────────────────── JAVA ────

It first decrements `b` by one and returns 0 in the case that `b` is 0 and the result of the division otherwise. Since the case distinction in $Post_D$ refers to the values of `a` and `b` in the *post-state*, `m` fulfils $opct_D$. It does however not fulfil $opct_C$, since, here, the case distinction refers to the pre-state, since it is written in the precondition. So in the case of `b` $= 1$ in the pre-state, our implementation delivers 0 but $opct_C$ promises `a`.

The example shows that our first attempt to improve the proof obligation for behavioural subtyping for operations was wrong. We failed to take into account that preconditions and postconditions are evaluated in different states. A vehicle to account for this difference are anonymising updates $\mathcal{V}$ (see Def. 3.59). $\mathcal{V}$ assigns arbitrary unknown values to all locations which can be possibly modified by an operation. We are thus assuming a worst-case approximation of the actually executed operation. This update is put "between" preconditions and postconditions in our proof obligation. In the example we get:

$$
\mathtt{b} \;!\!\doteq\; 0 \;\rightarrow\; \{\mathcal{V}\}\big((\mathtt{b} \doteq 0 \rightarrow \mathtt{result} \doteq 0) \tag{8.2}
$$
$$
\&\; (\mathtt{b} \;!\!\doteq\; 0 \rightarrow \mathtt{result} \doteq \mathtt{a/b}) \rightarrow \mathtt{result} \doteq \mathtt{a/b}\big)
$$

In order to know *which* locations are modifiable, we have already required that a modifies clause is present in an operation contract. In our example we could specify that the modifies clause is empty, i.e., we want *op* to be a query without side effects. In that case, $\mathcal{V}$ is the empty update, and can simply be omitted. Then the proof obligation (8.1) results, which holds as desired.

The empty modifies clause would not be correct for the JAVA method in (8.2). We had to specify $Mod = \{\mathtt{a}\}$. Now we get $\mathcal{V} = \mathtt{a} := a'$ where a' is a rigid function symbol. With it (8.2) is, as desired, not valid. Remember that $\mathcal{V}$ equals the completely anonymous update $*$ in case that *everything* is allowed to be modified. Then, since there is no relation at all between pre-state and post-state, (8.2) is not valid either.

In general the proof obligation template for behavioural subtyping with respect to two operation contracts $opct_C$ and $opct_D$ is as follows:

Proof Obligation Template 3

BehaviouralSubtypingOperationPair($opct_C$, $opct_D$):

$$
Pre_C \rightarrow Pre_D \tag{8.3a}
$$
$$
Pre_C \rightarrow \{\mathcal{V}\}(Post_D \rightarrow Post_C) \tag{8.3b}
$$

where

- *D is a subclass of C.*
- *Pre_C and Pre_D are the preconditions of the operation contracts $opct_C$ and $opct_D$ (resp.), $Post_C$ and $Post_D$ are their postconditions, and Mod is their modifies clause (which must be identical in both contracts). All of these formulae use the same program variables for the receiver object, the arguments, and the result.*
- *$\mathcal{V}$ is the anonymising update for Mod.*

We could relax the requirement of identical modifier sets to modifier sets that have the same effect, but the increased complexity of verifying this hardly pays off.

This is a proof obligation for a pair of operation contracts. Behavioural subtyping requires to check this obligation for *all* operation contracts:

Proof Obligation Template 4

BehaviouralSubtyping(C, D):

For all operation contracts for an operation op_C in C and all operations op_D which override op_C in the subclass D of C there is an operation contract $opct_D$ for op_D with:

$$BehaviouralSubtypingOperationPair(opct_C, opct_D)$$

For specifications that strictly follow the subtyping discipline described, e.g., in [Leavens and Dhara, 2000] where all operation contracts are inherited or copied to subclasses, behavioural subtyping is trivially satisfied. In this case the proof obligation templates (8.3a) and (8.3b) reduce to the tautologies $Pre_C \Rightarrow Pre_C$ and $Pre_C \Rightarrow \{\mathcal{V}\}(Post_C \Rightarrow Post_C)$.

In our opinion the behavioural subtyping discipline is not universally accepted at the moment and not without criticism. Even JML, that adopts inheritance of operation contracts as default, knows the construct of *code contracts* that are not inherited. The KeY system gives the designers the freedom to choose their preferred methodology on operation contracts.

8.1.4 Strong Operation Contract

We had previously introduced the concept of a *strong operation contract* ($\Rightarrow$ Sect. 1.5), i.e., a contract that is strong enough to ensure that all invariants of its class are satisfied after the return of the operation provided they were true before. The ultimate check of this property can only be done when an implementation of the operation is available. What can be done at the level of design verification? One could try to prove that the postcondition of an operation *op* implies all invariants. This will rarely be the case, since it does not make use of the fact that the invariants and the precondition may be assumed to be true at the start of the operation. As in the previous section we make use of anonymising updates to cope with this problem and arrive at the following proof obligation template:

Proof Obligation Template 5

StrongOperationContract(opct; Assumed; ϕ):

$$Pre \mathbin{\&} \phi \mathbin{\&} \mathrm{Conj}_{Assumed} \mathbin{\&} \{\mathcal{V}\}Post \Rightarrow \{\mathcal{V}\}\phi$$

where $\mathcal{V}$ is the anonymising update for the modifies clause Mod of opct (see Def. 3.59), Pre is the precondition and Post the postcondition of opct and ϕ

is an invariant. Moreover Pre, Post, and Mod use the same program variables for the receiver object, the arguments, and the result. The set Assumed consists of additional invariants which are assumed to be valid before the operation invocation; $\mathrm{Conj}_{Assumed}$ *is their conjunction. If op is a constructor operation neither* ϕ *nor Assumed are allowed.*

We have included in this template the extra parameter *Assumed* to allow for greater flexibility in its application. In many case it will be sufficient to instantiate *Assumed* as the empty set. In other cases the invariants might interact with each other.

And this is what this proof obligation does. Assume first, that we have verified that all operations satisfy their contracts. This involves, of course, looking at the methods implementing these operations. Further assume that all instances of the *StrongOperationContract* template for all operations *op* and all invariants ϕ can be proved, no implementations need to be considered for this second step, then we are sure that all invariants are true in all reachable system states. This is an ideal situation, which we will not take for granted in general. Templates to deal with less ideal situations will be discussed in the Sections 8.2 to 8.5.

Let us recall the example from Sect. 1.5 with the specification of the operation `charge` in class `PayCard`:

```
—— OCL ——
context     PayCard::charge(amount: Integer)
modifies : balance
pre :       balance + amount < limit and amount >=0
post :      balance = balance@pre + amount
                                            —— OCL ——
```

an the invariant:

```
—— OCL ——
context PayCard
inv: withinLimit : 0 <= balance and balance <= limit
                                            —— OCL ——
```

Instantiating *StrongOperationContract* to these specification we end up with the formula:

```
—— KeY ——
\forall PayCard pc; balance@pre(pc)=pc.balance
& self.balance + amount < self.limit & amount >= 0
& \forall PayCard pc; (pc.<created>=TRUE -> 0 <= pc.balance
                       & pc.balance <= pc.limit)
& {self.balance:=l_balance(self)}
           self.balance = balance@pre(self) + amount
-> {self.balance:=l_balance(self)}
```

```
\forall PayCard pc; (pc.<created>=TRUE -> 0 <= pc.balance
                     & pc.balance <= pc.limit)
```
--- KeY ------

It can be discharged with the KeY prover.

If we omitt the precondition, as has been done in the first specification attempt described in Sect. 1.5, the corresponding instantiation of *StrongOperationContract* would fail, even if we add the modifies clause containing the attribute `balance`.

8.2 Observed-State Correctness

At some point during software development programs come into play and the question arises whether the program is *correct* according to the specification. In this section we define what we consider to be a correct program.

Let us first recall the notations from the beginning of this chapter. We deal with a program P and its specification Spec. Specifications are for the purposes of this chapter JAVA CARD DL formulae. The specification consists of a set of invariants Inv_{Spec} and operation contracts *opct* for each operation *op*. Each operation contract in turn specifies a precondition *Pre*, a postcondition *Post*, a modifier set *Mod*, and the termination marker *total*.

The basic definition, that we elaborate on in the rest of this subsection, is

Definition 8.2 (Observed-State Correctness). *A program P is* observed-state correct *w.r.t. a specification* Spec*, if*

1. *all operations op fulfil all operation contracts of* Spec *for op,*
2. *all invariants Inv_{Spec} of* Spec *are preserved by all operations of P, and*
3. *all invariants are valid in the initial state of P.*

In the sequel, we make explicit what constitutes fulfilment of operation contracts and preservation of invariants. The mental model behind this notion of correctness is that of an *observer* of P. We think of an observer as a set of classes that may run all public methods of P. The observer *sees* all internal details of P, all values stored in all fields and arrays of all of P, but only in the pre-state and post-state. He does not see *intermediate* states. He has access to all public fields and can run a public method in a state satisfying its precondition and observe whether the final state satisfies its postcondition. He can see, whether the pre-state satisfies an additional property I and he can check whether I is again true in the post-state. When all observers only observe behaviour of P consistent with Spec, we call P observed-state correct. On the other hand, if there is an observer which observes behaviour which differs from Spec, then P is incorrect. In our opinion the observed-state paradigm fits very well with the *design-by-contract* methodology of Meyer [1992].

We point out though, that in the presence of a transaction concept in the programming language observed-stated correctness is not sufficient. In this case it is indispensable to take intermediate states into account ($\Rightarrow$ Sect. 9.4). Our observed-state semantics also differs from the JML visible state semantics ($\Rightarrow$ Sect. 5.3); we discuss this issue in the next subsection.

The observed-state model seems simple and obvious. Yet there are some fine points to be clarified: How is the call performed? Are there restrictions when an observer may call an operation, that is, which *assumptions* are made before invoking the operation? And most importantly: When is an observed behaviour judged to be correct, that is which conditions should be *ensured* in the post-state of an operation? What methods are available to the observer in the initial state? An observer could as well modify fields of a sufficient visibility, like public, *directly*. If invariants depend on these fields giving any guarantees on the validity of such invariants is almost impossible. We thus require that all fields are declared as private or protected. Note that classes using fields of other visibilities can easily be transformed into programs which satisfy this requirement by making the field protected and providing "setter" and "getter" methods. These new methods make it apparent that assertions on these fields are difficult to meet.

8.2.1 Observed States vs. Visible States

Another approach to entire program correctness employs the *visible state semantics* [Poetzsch-Heffter, 1997, Leavens et al., 2006].

A state is *visible* [Leavens et al., 2006] for an object o if it is reached at one of the following moments during the execution of a program; we leave out finalisers and JML's helper methods for simplicity:

- at the end of a constructor invocation which is initialising o,
- at the beginning and end of a non-static method invocation with o as receiver,
- at the beginning and end of a static method which is declared in the class of o or a superclass.
- when no constructor, non-static method invocation with o as receiver, or static method invocation for a method in o's class or a superclass is in progress.

A state is visible for a type T if it is reached after static initialisation for T is complete and it is a visible state for some object of type T.

The visible state semantics requires all instance invariants declared in type T to hold for every object o of type T in every visible state for o. All static invariants declared in T must hold in every visible state for T.

Clearly this semantics is stronger than observed-state correctness. The example program in Fig. 8.1 is observed-state correct but not visible-state correct:

```java
1      public class A {
2        private int i = 1;
3        /*@ instance invariant i>0 */
4
5        public int getI() { return i; }
6        /*@ requires p>0;
7          @ ensures i==p;
8          @*/
9        public void setI(int p) { i=p; }
10       public void m1() {
11         setI(0);
12         i=1;
13       }
14       public void m2() {
15         i=0;
16         setI(1);
17       }
18       public int m3() {
19         i=0;
20         i=(new B()).m5(this);
21       }
22       /*@ ensures \result>0 @*/
23       public int m4() {
24         return 42/i;
25       }
26     }
27
28     public class B {
29       /*@ ensures \result>0 @*/
30       public int m5(A a) {
31         if (a.getI()<=0) a.setI(1);
32         return a.m4();
33       }
34     }
```

Fig. 8.1. An example program demonstrating the differences between visible state semantics and observed-state semantics

With the visible-state semantics, the methods m1(), m2(), and m3() are not correct:

- According to visible-state semantics the invariant i>0 must be established in line 12 where the call to method setI(int) returns. This is not the case. With observed-state correctness, this state does not matter since it is not observed by an observer.

- For `m2()` the visible state semantics requires that the invariant holds in line 16 before the call to `setI(int)` starts. This is not the case but again an observer does not notice such a state and when `m2()` terminates the invariant is valid again.
- For the same reason `m3()` does not fulfill its contract in the visible state semantics, but it does in the observed-state semantics.

Our view is more liberal and we believe closer to a programmer's view of invariants.

An argument often used in favour of visible state semantics [Barnett and Naumann, 2004, Huizing and Kuiper, 2000] is the fact that invariants are required to be true at the start of *reentrant calls* (also referred to by the name *call-back*). To explain this phenomenon we look at the classes `A` and `B` from our example in Fig. 8.1.

When an `A` instance `o` executes `m3()` it calls at line 20 method `m5(o)` on a `B` instance. The `B` instance calls back to `o` namely the method `m4()` on line 32. The visible state semantics requires that the invariant in the visible state reached on line 32 is true, which in the example it in fact is. Nevertheless, instance `o` violates its visible state contract already on line 20. On the other hand, the program {`A`,`B`} is observed-state correct: Which ever method an observer invokes, in any state observable state in which the invariant holds true, after the method returns `A`'s invariant holds again.

This section is about the semantical definition of observable state correctness. But, we feel it will be helpful for the reader to include here some explanations of the calculus that is used in our verification system. The KeY verification system tries to prove that `m3()` preserves the invariant `i>0` by symbolically executing the program code of `m3()`. When symbolic execution reaches line 32 it knows that `i>0` is true and subsequent symbolic execution of `m4()` succeeds to establish the invariant after termination of `m3()`. The difference to the visible state semantics becomes apparent, if we decide in the KeY verification effort not to symbolically execute method `m4()`, but use its contract instead. Since the precondition of `m4()` is **true** the proof would again succeed. But, the system would record that the contract for `m4()` has not been proved yet. It turns out that it cannot be proved as it stands, we have to assume the invariant as a precondition. The proof that `m3()` preserves the invariant is still open at this point and one has to redo it. Let us consider another scenario, where we first prove the contract of `m4()` using the invariant as an additional precondition. Next the proof that `m3()` preserves the invariant is started again using the contract of `m4()` when symbolic execution reaches line 32. The system offers (or automatically selects) the proved contract. Since the state on reaching line 32 is not observable, the invariant cannot be assumed to hold and has to be discharged on the basis of what the system knows at this point of the symbolic execution. In the present example this proof succeeds.

Note, that with the help of our JAVA CARD DL calculus we *can* enforce visible state semantics by requiring all invariants to be true whenever a method contract is invoked. When we try, e.g., to prove that `m2()` preserves invariants and symbolic execution has reached line 16 we use the contract of `setI(int)` with the invariant as additional assumption. In order to use this contract we must prove that the precondition of `setI(int)` and the invariant holds. This fails as it should since `class A` is not visible state correct. The KeY calculus is more flexible and would allow us to use a contract which relies on weaker assumptions than that of all invariants: In our example, a contract which does not include the instance invariant would hold and could thus be used.

8.2.2 Assumptions Before Operation Calls

As already remarked in Sect. 5.1.1 an operation contract does not give guarantees for *all* pre-states. Our observer model reflects this by calling the observed program P only in certain states that satisfy properties which we define in this section.

In particular such a state must be reachable from the start state of the system by calling operations on P and performing other statements in the observer. Any intermediate state while a method in a class of the observer is in progress is such a reachable state. Instead of talking about an intermediate state we could also use any final state of a method execution (since our observer can be any program):

Definition 8.3. *A state S is* reachable *by a program P if there is a class Obs with one static method m such that at the end of the execution of m the state S is reached.*

Clearly, other states cannot be checked by an observer and according to our observer model we are only interested in what an observer can observe. We will extend the notion of reachable states in the context of components in Sect. 8.5.4.

Our first-order specification languages are unfortunately not capable of making statements about reachability of states. When formalising proof obligations, such a requirement can thus not be formalised. The way out is that we must *characterise* reachable states by properties which they must satisfy. This is exactly the purpose of class invariants. Nevertheless for the theoretical notion of legal pre-states we include the condition that these states must be reachable.

Though we cannot completely formalise reachability of pre-states we can give some necessary conditions of reachable states which are obvious in our mental observer model, but must now be made explicit:

- If the observer invokes an instance method, then the receiver object `self` is a *created* object; in our first-order logic we write

$$\texttt{self.<created>} \doteq \text{TRUE} \quad .$$

- All arguments $\texttt{p}_1, \ldots, \texttt{p}_n$ passed to the formal parameters of the method call are created objects, $\texttt{null}$, or a primitive value; so for every $i = 1, \ldots, n$ where p_i is of a reference type:

$$\texttt{p}_i\texttt{.<created>} \doteq \text{TRUE} \mid \texttt{p}_i \doteq \texttt{null} \quad .$$

We write $\textit{ValidCall}_{op}(\texttt{self}, \texttt{p}_1, \ldots, \texttt{p}_n)$ or simply $\textit{ValidCall}_{op}$ to refer to the conjunction of these formulae for an operation op.

According to what we said in Sect. 5.1.1 we are furthermore only interested in pre-states which satisfy a precondition of the considered operation. An operation may have more than one operation contract, so that we can assume that the disjunction of the preconditions of all contracts of the operation holds. We denote this formula by Disj_{Pre}.

From the point of view of an observer, invariants always hold. We may thus assume that all of them hold in a pre-state.

8.2.3 Operation Calls

To compare implemented and specified behaviour, the observer must invoke the method or constructor. The call is arbitrary in the sense that we have arbitrary receiver and arguments for the call. We are thus using unknown but fixed objects which we assume to be stored in the suitably typed program variables $\texttt{self}$ (for the receiver object, if one is needed) and as well suitably typed argument program variables $\texttt{p}_1, \ldots, \texttt{p}_n$. Depending on the signature, a variable (usually called $\texttt{r}$) that captures the returned value of a non-void method call is needed.

The observer uses the JAVA statement $\text{Prg}^0{}_{op}(\texttt{self}; \texttt{p}_1, \ldots, \texttt{p}_n; \texttt{result})$ listed in Table 8.1, depending on the program variables introduced above, as call to method or constructor op. Note, that we use *method body statements* as introduced in Chapter 3. This means, dynamic binding is not triggered by the observer call, instead a concrete method body is executed. Consider for instance the following method declared in class C:

--- JAVA (8.2) ---

```java
public void m(int i) {
   i=1;
}
```

JAVA

When a method body statement `self.m@`C`(p);` is executed in the post-state `p` equals 1. As a postcondition of `m` however, we expect $\mathtt{p} \doteq 1$ *not* to be satisfied, since assignments to parameters are hidden from the caller of a method.

The pure program $\mathrm{Prg}^0{}_{op}()$ is thus not what is desired. Our observer should naturally not see the assignments made to the arguments of the operation call. Thus we assign the arguments $\mathtt{p}_1, \ldots, \mathtt{p}_n$ to fresh variables $\mathtt{p}'_1, \ldots, \mathtt{p}'_n$ before the method body statement. So if $\mathtt{T}_1, \ldots, \mathtt{T}_n$ are the static types of $\mathtt{p}_1, \ldots, \mathtt{p}_n$ we use, instead of $\mathrm{Prg}^0{}_{op}(\mathtt{self}; \mathtt{p}'_1, \ldots, \mathtt{p}'_n; \mathtt{result})$, the following statements:

$\mathtt{T}_1$ `p`$'_1$ = `p`$_1$;
$\vdots$
$\mathtt{T}_n$ `p`$'_n$ = `p`$_n$;
$\mathrm{Prg}^0{}_{op}(\mathtt{self}; \mathtt{p}'_1, \ldots, \mathtt{p}'_n; \mathtt{result})$

When we reason about the example method `m` in (8.2) we would consider the program `int p'=p; self.m@`C`(p');`. In its post-state $\mathtt{p} \doteq 1$ is—as intended—*not* true.

Recall that JAVA CARD DL defined abrupt termination as non-termination. If the observer "executes" $\mathrm{Prg}^0{}_{op}(\mathtt{self}; \mathtt{p}_1, \ldots, \mathtt{p}_n; \mathtt{result})$ this implies that no statement about the post-state of an abruptly terminated *op* can be made, at least with the standard modalities $\langle \cdot \rangle$ and $[\cdot]$. Thus, this JAVA sequence is not sufficient to make statements about exceptional behaviour of methods. Not too much is missing, though, to get information on thrown exceptions: Potentially thrown exceptions are caught and assigned to an additional program variable `exc` which is assigned `null` before invoking the method or constructor (we abbreviate `result` by `res` here):

Table 8.1. Programs in proof obligations

$\mathrm{Prg}^0{}_{op}(\mathtt{self}; \mathtt{p}_1, \ldots, \mathtt{p}_n; \mathtt{result})$	if *op* is a...
`result=self.`m`(p`$_1$`,...,p`$_n$`)@`D`;`	method in type D declared as D_0 $m(D_1, \ldots, D_n)$
`self.`m`(p`$_1$`,...,p`$_n$`)@`D`;`	method in type D declared as `void` $m(D_1, \ldots, D_n)$
`result=`D`.`m`(p`$_1$`,...,p`$_n$`);`	method in type D declared as `static` D_0 $m(D_1, \ldots, D_n)$
D`.`m`(p`$_1$`,...,p`$_n$`);`	method in type D declared as `static void` $m(D_1, \ldots, D_n)$
`result=new` D`(p`$_1$`,...,p`$_n$`);`	constructor declared as $D(D_1, \ldots, D_n)$

$$\mathrm{Prg}_{op}(\mathtt{self}; \mathtt{p}_1, \ldots, \mathtt{p}_n; \mathtt{res}; \mathtt{exc}) := \begin{cases} \mathtt{T}_1 \ \mathtt{p}'_1 \ = \ \mathtt{p}_1; \\ \vdots \\ \mathtt{T}_n \ \mathtt{p}'_n \ = \ \mathtt{p}_n; \\ \mathtt{exc}\texttt{=}\mathbf{null}; \\ \mathbf{try}\{ \\ \quad \mathrm{Prg^0}_{op}(\mathtt{self}; \mathtt{p}'_1, \ldots, \mathtt{p}'_n; \mathtt{res}) \\ \} \ \mathbf{catch} \ (\texttt{Throwable e}) \ \{ \\ \quad \mathtt{exc} \ = \ \mathtt{e}; \\ \} \end{cases}$$

In post conditions the value of `exc` may now be referred to. It is either `null`, then *op* terminated normally, or is assigned an exception which is the reason for the abrupt termination.

From time to time however, we are not interested in the actual thrown exception but would only like to know that a postcondition holds independently of the question whether the method terminated normally or abruptly. Then, the following JAVA sequence suffices:

$$\mathrm{Prg}_{op}(\mathtt{self}; \mathtt{p}_1, \ldots, \mathtt{p}_n; \mathtt{result}) := \begin{cases} \mathtt{T}_1 \ \mathtt{p}'_1 \ = \ \mathtt{p}_1; \\ \vdots \\ \mathtt{T}_n \ \mathtt{p}'_n \ = \ \mathtt{p}_n; \\ \mathbf{try}\{ \\ \quad \mathrm{Prg^0}_{op}(\mathtt{self}; \mathtt{p}'_1, \ldots, \mathtt{p}'_n; \mathtt{result}) \\ \} \ \mathbf{catch} \ (\texttt{Throwable e}) \ \{\} \end{cases}$$

In the sequel, we are fixing the program variables $\mathtt{self}, \mathtt{p}_1, \ldots, \mathtt{p}_n, \mathtt{result}$, $\mathtt{exc}$ and thus most often simply write $\mathrm{Prg}_{op}()$ instead of the more verbose $\mathrm{Prg}_{op}(\mathtt{self}; \mathtt{p}_1, \ldots, \mathtt{p}_n; \mathtt{result}; \mathtt{exc})$ or $\mathrm{Prg}_{op}(\mathtt{self}; \mathtt{p}_1, \ldots, \mathtt{p}_n; \mathtt{result})$.

8.2.4 Assertions After Operation Calls

When an observer invokes an operation as described above in a pre-state which satisfies the assumptions stated in Sect. 8.2.2, then:

1. If the operation is required to terminate (that is, the *total* marker has been set in *opct*) the call must terminate in a post-state.
2. If it terminates, then in the post-state
   a) the postcondition *Post* of *opct* is valid,
   b) only the locations described by the modifies clause in *opct* are modified (comparing pre-state and post-state and ignoring local variables),
   c) all invariants Inv_{Spec} are satisfied.

If Conditions 2a and 2b are satisfied we say that *op fulfils the operation contract opct.* If the Condition 2c is satisfied we say that *invariants are preserved* by *op*. This will be more precisely defined as follows. Before that we are discussing the role of invariants as assumptions in each of these notions.

Invariants describe conditions that are, for an observer, always present in the observed program. Though invariants are (at least in the specification languages UML/OCL and JML) always defined in one class or interface, their effective scope is global. In particular, a method called by the observer can rely on *any* invariant defined in other types than those which the method is declared in. From the perspective of the overall proving process it may however be advantageous *not* to assume *all* invariants at that call: When we make use of contracts as lemma rules in a proof it is desirable to have as few assumptions as possible. Since all of them must be proven to hold when a method contract is applied. We are thus relativising both notions and define fulfilment of operation contracts and preservation of invariants of *under the assumption* of some subset of all invariants.

Definition 8.4 (Fulfilling Operation Contracts). *Let P be a program and* Spec *a specification for P. Let furthermore*

- *op be an operation declared in type T with parameter types $T_1, \ldots, T_n$ and return type R in a program P*
- `self` *be a program variable of type T,*
- $p_1, \ldots, p_n$ *be program variables of types $T_1, \ldots, T_n$,*
- `result` *be a program variable of type R,*
- `exc` *a program variable of type* `java.lang.Throwable`, *and*
- *opct = (op', Pre, Post, Mod, Termin) be an operation contract of* Spec

op fulfils the operation contract opct under the assumption of $I \subseteq Inv_{\mathrm{Spec}}$ if op' = op or op overrides op' and for all reachable states S_{pre} and all $e_{\texttt{self}} \in \mathcal{D}^T$, $e_i \in \mathcal{D}^{T_i}$ $(i = 1, \ldots, n)$ with $S_{\mathrm{pre}}(\texttt{self}) = e_{\texttt{self}}$ and $S_{\mathrm{pre}}(p_i) = e_i$ with

1. $S_{\mathrm{pre}} \models ValidCall_{op}$
2. $S_{\mathrm{pre}} \models Pre$
3. $S_{\mathrm{pre}} \models I$

the following conditions hold:

- *If the total marker is set there is a state S_{post} with*

$$(S_{\mathrm{pre}}, S_{\mathrm{post}}) \in \rho_{\mathrm{Prg}_{op'}(\texttt{self};p_1,\ldots,p_n;\texttt{result};\texttt{exc})} \ .$$

- *If there is such a state S_{post} then:*

$$S_{\mathrm{post}} \models Post(\texttt{self}; p_1, \ldots, p_n; \texttt{result}; \texttt{exc})$$
$$(S_{\mathrm{pre}}, S_{\mathrm{post}}) \models Mod(\texttt{self}; p_1, \ldots, p_n)$$

Definition 8.5 (Preservation of Invariants). *With the same abbrevia-tions as in the last definition, for some sets I and I' of invariants:*

op preserves a set of invariants I' *of a specification of P under the as-sumption of I if for all reachable states* $\mathrm{S_{pre}}$ *and all* $e_{\mathtt{self}} \in \mathcal{D}^T$, $e_i \in \mathcal{D}^{T_i}$ *($i = 1, \ldots, n$) with* $\mathrm{S_{pre}}(\mathtt{self}) = e_{\mathtt{self}}$ *and* $\mathrm{S_{pre}}(p_i) = e_i$ *with*

   1. $\mathrm{S_{pre}} \models \mathit{ValidCall}_{op}$
   2. $\mathrm{S_{pre}} \models \mathrm{Disj}_{Pre}$
   3. $\mathrm{S_{pre}} \models I$
   4. $(\mathrm{S_{pre}}, \mathrm{S_{post}}) \in \rho_{\mathrm{Prg}_{op}(\mathtt{self};p_1,\ldots,p_n;\mathtt{result})}$

the following condition holds for all $\phi \in I'$:

$$\mathrm{S_{post}} \models \phi \ .$$

Typically invariants impose constraints on all instances of a particular class. Usually one tends to think that constructors must *establish* such instance in-variants rather than *preserve* them. Remember again that instance invariants are formalised as

$$\backslash\mathrm{forall}\ C\ o;\ (o.\mathtt{<created>} \doteq \mathrm{TRUE} \mathbin{-\!\!>} \phi(o))\ .$$

In the case of a constructor we get:

- Before the invocation of a constructor, the instance invariant of the object to be created is not assumed to hold since we restrict our quantification to all created objects. This is as desired.
- When the constructor terminates normally an object is created and quan-tification includes it. Again this reflects our idea of what we expect from an invariant.
- When the constructor terminates abruptly the object is created but not initialised. We must however require that invariants hold for such objects, too. The reason is that during object initialisation references may have leaked and are then available to other objects.

Consider furthermore a state where no classes are initialised and no objects created. According to our mental model, instance invariants do not need to hold in such a state. And in fact the instance invariant quantified over all created objects evaluates trivially to true. By Def. 8.2 we, moreover, only permit static invariants which evaluate to true if all classes are not initialised.

8.2.5 Static Initialisation

In the initial state of a program P no classes are initialised and thus no instances created. Since instance invariants are universally quantified over all created instances, they are trivially true in the initial state.

Thus, our only concern is with static invariants. An invariant of the form `C.a` $\doteq 0$ would not be true in the initial state However, the following reformulation would:

$$C.\texttt{<classInitialized>} \doteq \text{TRUE} \rightarrow C.a \doteq 0 \ .$$

We then go on and try to prove that this formula is preserved by all operations.

Definition 8.6. *An initial state of a program P is an interpretation S_{init} for which for all classes $C \in P$:*

$$S_{init} \models C.\texttt{<classPrepared>} \doteq FALSE \ .$$

The values of all other implicit fields in S_{init} relevant for static initialisation can be deduced using the JAVA CARD DL calculus. If `<classPrepared>` is set to FALSE, all other variables are still set to their default values, for instance `<classInitialized>` is set to TRUE.

We make a restricting but realistic assumption on the state of static class initialisation in all states an observer can consider: All classes in P have either already *successfully* processed static class initialisation or have not started with their static initialisation. In pathological cases, classes can be in an *erroneous state* after having processed class initialisation, though such classes and their instances may behave according to their specifications. Although the JAVA CARD DL calculus correctly treats these cases ($\Rightarrow$ Sect. 3.6.6), we rule out that erroneous classes exist when a method or constructor is called, since in practice it is

- unlikely that a programmer ever *intends* to have erroneous classes during the execution of his program; if they occur something is likely to be wrong, and
- highly complicated to prove properties with more liberal assumptions than we are postulating.

These assumptions are encoded and treated as implicitly present invariants, that is we include for every non-abstract class C the following to the invariants Inv_{Spec}:

$$C.\texttt{<classErroneous>} \doteq \text{FALSE}$$
$$\& \ C.\texttt{<classInitialisationInProgress>} \doteq \text{FALSE} \quad (8.4)$$

The static initialisation routine of a class C is simulated in JAVA CARD DL by a static method `<clinit>()` implicitly declared in C ($\Rightarrow$ Sect. 3.6.6). As for ordinary methods and constructors, an observer may invoke `<clinit>()`, that is—in reality—it triggers static initialisation.

We require that assumptions and assertions for ordinary methods and constructors, which were postulated above, should hold for `<clinit>()`, too. There is only a minor adjustment to be made which we express by means of an implicit operation contract for `<clinit>()`.

Definition 8.7. *The* standard operation contract *of* C.`<clinit>()` *consists of the precondition*

C.`<classInitialised>` $\doteq$ *FALSE*

$$\& \ C.\texttt{<classPrepared>} \doteq \textit{FALSE} \quad (8.5)$$

and the termination marker total.

We assume that this standard operation contract is always present in our specifications.

As a summary: For static initialisation we *assume* the precondition from the standard operation contract to hold. As we included (8.4) to the invariants, we may assume in all other proof obligations that no class is erroneous and not being initialised as well as all invariants hold. It is *ensured* that static initialisation terminates normally, that is without top-level exception, again no class is erroneous, and all invariants hold. JAVA by itself ensures that no class is currently being initialised after a static initialisation terminates, so that this part of the implicitly given invariant is not needed to be shown.

8.3 Lightweight Program Correctness

Often while writing code one is not interested in the correctness of the whole program but only in certain *lightweight* properties, for instance that a method preserves a set of invariants.

Table 8.2 shows all such proof obligations. Table 8.3 lists the used notation. For a more concise display we assume that all templates operate on the same set of program variables; by convention this is `self` for the receiver object, `result` for the result variable, $p_1, \ldots, p_n$ for the arguments, and `exc` for the variable storing the thrown exception (or `null` if none is thrown).

By systematically combining these proof obligations, as shown in the next section, we can prove observed-state correctness of a program. The rest of this section explains the design and purpose of the proof obligations.

8.3.1 Invariants

According to Def. 8.2 we are interested in two properties of invariants:

1. They must be valid in all initial states.
2. (All) operations must preserve them.

The first item is covered by the template *InitInv*, the latter by *PreservesInv*.

InitInv This template is a direct translation of Def. 8.6. It requires that a formula ϕ holds in an initial state of a program.

As remarked earlier, instance invariants, that is, invariants of the shape $\forall\, C\ x;\ (x.\texttt{<created>} \doteq \text{TRUE} \rightarrow \phi')$ are true in the initial state

Table 8.2. Proof obligation templates for program correctness

Proof Obligation Template	Formula
$InitInv(I)$	$\phi_{\text{init}} \mathrel{-}{>} \text{Conj}_I$
$PreservesInv(op; Assumed; Ensured)$	$\text{Conj}_{Assumed}\ \&\ \text{Disj}_{Pre}\ \&\ ValidCall_{op}$ $\mathrel{-}{>} [\text{Prg}_{op}()]\,\text{Conj}_{Ensured}$
$PreservesOwnInv(op; Assumed)$	$\text{Conj}_{Assumed}\ \&\ \text{Disj}_{Pre}\ \&\ ValidCall_{op}$ $\mathrel{-}{>} [\text{Prg}_{op}()]\,\text{Conj}_I$
$EnsuresPost(opct; Assumed)$	$\text{Conj}_{Assumed}\ \&\ Pre\ \&\ ValidCall_{op}$ $\mathrel{-}{>} \langle\text{Prg}_{op}()\rangle Post$ (if $Termin = total$)
	$\text{Conj}_{Assumed}\ \&\ Pre\ \&\ ValidCall_{op}$ $\mathrel{-}{>} [\text{Prg}_{op}()]Post$ (if $Termin = partial$)
$RespectsModifies(opct; Assumed)$	$\text{Conj}_{Assumed}\ \&\ Pre\ \&\ ValidCall_{op}$ $\&\ \text{PreAxioms}\ \&\ \{\mathcal{V}\}\langle\texttt{anon();}\rangle\text{true}$ $\mathrel{-}{>} [\text{Prg}_{op}()]\{\mathcal{V}^{@pre}\}\langle\texttt{anon();}\rangle\text{true}$

Table 8.3. Abbreviations used in proof obligation templates

Abbreviation	Explanation
op	an operation of P
$opct$	an operation contract for op with $opct = (Pre, Post, Mod, Termin)$
$I, Assumed, Ensured$	subsets of Inv_{Spec}
Conj_F	conjunction of the formulae contained in set F
Disj_{Pre}	disjunction of all preconditions for op
ϕ_{init}	characterisation of the initial state; equivalent to the conjunction of $C.\texttt{<classPrepared>} \doteq \text{FALSE}$ over all $C \in P$
$\mathcal{V}$	anonymising update as defined in Def. 3.59

and thus also trivially pass this proof obligation. It is thus unnecessary to invoke this proof obligation if the formula already has this shape.

PreservesInv This template exactly corresponds to Def. 8.5.

For the special case that the parameter *Ensured* equals the set of all instance invariants of a given class we introduce the separate proof obligation *PreservesOwnInv*. It is depicted in the third line of Table 8.2.

8.3.2 Postconditions and Termination

For an operation *op* and an operation contract *opct* on *op*, we certainly want to prove that *op* establishes the postcondition of *opct* and complies to the termination behaviour specified in *opct*.

As in the case of *PreservesInv*, the generated proof obligation is of the shape: $\psi \mathrel{-}{>} [p]\phi$ but the kind of modal operator ($[\cdot]$ in that formula) varies

depending on the required termination behaviour. If termination is required in the operation contract we use $\psi \rightarrow \langle p \rangle \phi$ instead.

8.3.3 Modifies Clauses

A modifies clause for an operation indicates the locations that it can *at most* modify. Checking modifies clauses belongs amongst the most difficult tasks in program verification. The reason is that the focus is on those locations which are *not* explicitly mentioned in the specification. All of them must be proven to be unmodified. Due to the nature of the problem, the proof obligation is not as intuitive as for the others presented in this chapter. We thus proceed in small steps.

A modifies clause of an operation contract *opct* of an operation *op* consists of a set

$$Mod = \{ \langle \phi^1, f^1(t_1^1 \ldots, t_{n_1}^1) \rangle, \ldots, \langle \phi^k, f^k(t_1^k \ldots, t_{n_k}^k) \rangle \} \ .$$

Intuitively, *Mod* is correct if a call p to *op* (which satisfies the invariants and the precondition of *opct*) assigns at most to the locations L_{Mod} described by *Mod*. Temporary violations of this rule, not visible to an observer, are legal however ($\Rightarrow$ Sect. 3.7.4).

We consider two JAVA CARD code snippets:

1. p': This is a method reference to *op* plus subsequent assignments of fixed but arbitrary values $\{v_1, v_2, \ldots\}$ to the locations in $L_{Mod} = \{l_1, l_2, \ldots\}$, that is: v_1 is assigned to l_1, v_2 to l_2, etc. p' executes *op* and afterwards overrides exactly those locations allowed by *Mod*.
2. p: This consists *only* of the assignments of $\{v_1, v_2, \ldots\}$ to the respective locations in $L_{Mod} = \{l_1, l_2, \ldots\}$. Note, that these values must be the same as in the program p'. Thus, p overrides only exactly those locations allowed by *Mod*.

Assume now that our program is in a pre-state S_{pre} which satisfies the precondition of *opct* and the invariants. Our goal is to formalise that p' and p result in the *same* state when started in S_{pre}. If *op* modified (when called in S_{pre}) other locations than specified in *Mod*, say value v' was assigned to location l', then the state resulting from p' would be different from p, since location l' is assigned to a different value in the former than in the latter case. So if p' and p terminate in the same state we are done and *Mod* is a correct modifies clause for *op* (under the considered preconditions and invariants).

So we must formalise in JAVA CARD DL that two states (obtained by p' and p, resp.) are equivalent. Two states are equivalent if all locations are assigned the same value in both states. To capture this property we make use of an *anonymous* method `anon()`. The behaviour of this method may depend on all locations. In particular the question under which circumstances `anon()` *terminates* depends on all locations of the program. So we can say

that two states S_1 and S_2 are equivalent if the following condition holds: the anonymous method terminates started in S_2 if it terminates started in S_1.

In the following, we formalise these considerations as a JAVA CARD DL proof obligation. First of all, the pre-state S_{pre} is characterised as a state in which the precondition of *opct*, some subset *Assumed* of all invariants, and argument and receiver objects are created:

$$Pre \ \& \ \mathrm{Conj}_{Assumed} \ \& \ ValidCall_{op} \ .$$

Furthermore, we formalise the assignment of L_{Mod} to the fixed but unknown values as an update. In order to reflect that the values assigned to these locations are arbitrary, we use the anonymising update $\mathcal{V}$ introduced in Def. 3.59. The update has *rigid* symbols as top level operators of its right hand side terms. These match the signature of the top-level non-rigid symbols of the terms in the modifies clause. Termination of `anon()` is expressed, as usual, as $\langle$`anon();`$\rangle$true.

When formalising the update, we must take some care since the update occurs one time inside the scope of a modality; the locations that are to be updated refer to the pre-state of this modality however. In Sect. 5.2.3 we have seen a technique which allowed us to refer to values of non-rigid functions f in the pre-state. We introduce a *rigid* operator $f^{@pre}$ which reflects the signature of f and replace the occurrence of f which should refer to the pre-state by $f^{@pre}$. Moreover we must appropriately "set" $f^{@pre}$ in the pre-state by requiring

$$\backslash\text{forall } T_1 \ x_1; \ \cdots \backslash\text{forall } T_n \ x_n; \ f^{@pre}(x_1, \ldots, x_n) \doteq f(x_1, \ldots, x_n) \quad (8.6)$$

if $\alpha(f) = ((T_1, \ldots, T_n), T')$.

We want the complete update $\mathcal{V}$ to refer to the pre-state. But we must be careful: We must *not* replace the top-level operator f of the update's left-hand side by its counterpart $f^{@pre}$, simply because we want to refer to the *location* not to the *object*. Moreover replacing it would make the result not satisfy the definition of an update anymore.

We thus define $\mathcal{V}^{@pre}$ inductively as follows:

- $(f(t_1, \ldots, t_n) := t_0)^{@pre} := (f(t_1^{@pre}, \ldots, t_n^{@pre}) := t_0^{@pre})$,
- $(u_1 \, ; u_2)^{@pre} := (u_1^{@pre} \, ; u_2^{@pre})$,
- $(u_1 \,\|\, u_2)^{@pre} := (u_1^{@pre} \,\|\, u_2^{@pre})$, and
- $(\text{for } x; \varphi; u)^{@pre} := (\text{for } x; \varphi^{@pre}; u^{@pre})$
- $(f(t_1, \ldots, t_n))^{@pre} := (f^{@pre}(t_1^{@pre}, \ldots, t_n^{@pre}))$ for non-rigid functions f
- $t^{@pre} = t$ for all other kinds of terms and formulae t

We see that for all states S and all locations l

$$val_S(\mathcal{V}^{@pre})(l) = val_{S_{pre}}(\mathcal{V})(l)$$

if in S_{pre} all axioms (8.6) for the used $f^{@pre}$ function symbols hold.

Now we can assemble the proof obligation for the correctness of the modifies clause of an operation contract *opct* under the assumption of a set *Assumed* of invariants as in the last line of Table 8.2. PreAxioms denotes the conjunction of axioms (8.6) for all used $f^{@pre}$ function symbols.

This proof obligation template is correct:

Lemma 8.8. *If* $\models$ *RespectsModifies(opct; Assumed;* $\mathtt{self}; \mathtt{p}_1, \ldots, \mathtt{p}_n)$ *is true for some Assumed* $\subseteq Inv_{\mathrm{Spec}}$ *then in all states* $\mathrm{S}_{\mathrm{pre}}, \mathrm{S}_{\mathrm{post}}$ *with*

$$\mathrm{S}_{\mathrm{pre}} \models \mathrm{Conj}_{Assumed} \ \& \ Pre \ \& \ ValidCall_{op}$$
$$and \quad (\mathrm{S}_{\mathrm{pre}}, \mathrm{S}_{\mathrm{post}}) \in \rho_{\mathrm{Prg}_{op}(\mathtt{self};\mathtt{p}_1,\ldots,\mathtt{p}_n)}$$

the modifies clause Mod is satisfied (w.r.t. the pre-state $\mathrm{S}_{\mathrm{pre}}$ *and the post-state* $\mathrm{S}_{\mathrm{post}}$).

Example 8.9. In Sect. 5.3.1 we had an operation contract for the method `enterPIN (int pin)` describing the behaviour of an ATM when the correct PIN has been inserted. Translated in first order logic we get the following operation contract *opct*:

$$
\begin{aligned}
Pre: \quad & \mathtt{self.insertedCard} \,!\dot{=}\, \mathtt{null} \\
& \&\ \mathtt{self.customerAuthenticated} \,!\dot{=}\, \mathtt{TRUE} \\
& \&\ \mathtt{pin} \dot{=} \mathtt{self.insertedCard.correctPIN} \\
Post: \quad & \mathtt{self.customerAuthenticated} \dot{=} \mathtt{TRUE} \\
Mod: \quad & \mathtt{self.customerAuthenticated}
\end{aligned}
$$

The proof obligation *RespectsModifies(opct;* $\emptyset$*;* $\mathtt{self}$*;* $\mathtt{pin}$*)* that the modifies clause *Mod* is satisfied is as follows:

```
    self.insertedCard !≐ null
&   self.customerAuthenticated !≐ TRUE
&   pin ≐ self.insertedCard.correctPIN
&   self@pre ≐ self
&   {self.customerAuthenticated := f(self)}⟨anon();⟩true
−> [try{self.enterPIN@(ATM)(pin);}catch(Throwable e){}]
       {self@pre.customerAuthenticated := f(self@pre)}
                                           ⟨anon();⟩true
```

where f is a fresh rigid function.

8.4 Proving Entire Correctness

The properties presented so far are intended to help developers when questions concerning single correctness issues occur during development. At some point the program is finished being written and the question might arise

whether the complete program is entirely correct. Of course, the proofs originating from lightweight correctness proof obligations (with the unmodified underlying program context) remain useful in order to prove the entire correctness and might thus be re-used here.

Basic Proof Obligation System

The following proof obligation system consists of several elementary proof obligations and ensures observed-state correctness as defined in Def. 8.2. That definition is directly formalised:

Lemma 8.10. *Let S be a specification of P. If for all non-private operations op of P the following proof obligations are valid then P is observed-state correct w.r.t. S:*

- *for all operation contracts opct for op and some $Assumed \subseteq Inv_{\mathrm{Spec}}$:*

$$EnsuresPost(opct; Assumed) \quad and \quad RespectsModifies(opct; I) \ ;$$

- *for all invariants $\phi \in Inv_{\mathrm{Spec}}$ and for some $Assumed \subseteq Inv_{\mathrm{Spec}}$:*

$$PreservesInv(op, Assumed; \{\phi\}) \quad and \quad InitInv(\{\phi\}) \ .$$

Instead of proving $PreservesInv(op, I; \{\phi\})$ for all operations $\phi \in Inv_{\mathrm{Spec}}$ (and some $I \subseteq Inv_{\mathrm{Spec}}$) one could equivalently show the validity of the larger formula $PreservesInv(op, Inv_{\mathrm{Spec}})$.

One thing may look strange at a first glance: We require that *all* operations of the complete program preserve *all* invariants of the complete specification. This makes proving invariants really complicated! Were invariants not attached to a certain class, and is it not sufficient to let the operations of *just that* class preserve these invariants? A couple of things can be done to remedy the problem that proving invariants *is* complicated. These measures will be described in the rest of this chapter. But first let us explain why it is in fact necessary to consider all operations.

Consider the following invariant of the `ATM` class:

$$\texttt{insertedCard} \,!\!\doteq \texttt{null} \rightarrow \texttt{insertedCard.invalid} \doteq \text{FALSE}$$

which says that inserted cards are always not invalidated. There could be other references to the `BankCard` object stored in `insertedCard`. For instance, the `PermanentAccount` class could hold a reference to all cards associated with the account. If, for some reason, there is a method in `PermanentAccount` which sets the `invalid` bit in some `BankCard` instance to `true`, this could also affect the card inserted in the ATM. `ATM`'s invariant would thus be violated although no operation of `ATM` has been processed. If we checked however that the operations of `BankCard` preserve the operations of `ATM`, the `BankCard` instance cannot perform the invariant invalidating changes.

Problems

The preservation of invariants is with this definition quite a complex task, since the number of generated proof obligations *explodes*: with i the number of invariants in the program and j the number of operations, we get $i \cdot j$ proof obligations, whereas there are only $2 \cdot j$ proof obligations for ensuring that the operation contracts are fulfilled (assuming there is exactly one contract per operation).

   Moreover consider the situation in which we have verified a program P (w.r.t. invariants I) and then a class is added with additional invariants I'. What we would have to do now is to prove the preservation of I' for the operations of P *and* the preservation of I for the operations of the new class. This is a truly non-modular and undesired effect, which we tackle in the rest of this chapter.

Non-modular Improvements

Example 8.11. From our banking application as described in Chapter 5 we get the invariant:

$$\phi := \quad \forall c\!:\!\texttt{BankCard}.\ (c.\texttt{<created>} \doteq \text{TRUE}$$
$$\text{--> } 0 <= c.\texttt{wrongPINCounter}\ \&\ c.\texttt{wrongPINCounter} <= 2) \quad (8.7)$$

It is intuitively clear that this invariant cannot be violated by the method `addBonus` in `BankCard`. There should thus be no need to check that `addBonus` preserves this particular invariant, though in this case symbolic execution would be trivial. Clearly however there could be methods where a huge effort would be needed to get deductively rid of the code, although the method has no means to affect the evaluation of the considered invariant.

The intuition of a developer recognises quite immediately if a method cannot influence the evaluation of a certain invariant, since the locations that get changed by the method on the one side and those that an invariant depends on are distinct.

   We have already familiarised ourselves with the fact that those locations an operation may assign values to are specified in modifies clauses. If modifies clauses are checked as described above, we can make safe use of this kind of specifications to decide that an operation cannot be affected by an invariant.

   More precisely we make use of the fact that if $\mathcal{V}$ is the anonymising update relative to a modifies clause *Mod* and the formula $\{\mathcal{V}\}\phi$ is valid, then ϕ is valid after the execution of a program which *Mod* is an modifies clause for. So instead of symbolically executing the operation body we can apply the update $\mathcal{V}$. If this does not already affect a considered invariant, the actual body will neither.

The validity of

$$Pre \ \& \ \mathrm{Conj}_{Assumed} \ \& \ ValidCall_{op} \ \& \ \phi \Rightarrow \{\mathcal{V}\}\phi$$

for an operation op, $Assumed \subseteq Inv_{\mathrm{Spec}}$ thus implies that op preserves the invariant ϕ. We can however be more precise if we also take into consideration the precondition Pre and the postcondition $Post$ of an operation contract of op. We can assume that Pre holds in the pre-state and that after executing op, or its approximation $\mathcal{V}$, $Post$ holds.

If we pack all this into a formula we end up with a proof obligation we already know: $StrongOperationContract$:

$$Pre \ \& \ \phi \ \& \ \mathrm{Conj}_I \ \& \ \{\mathcal{V}\}Post \Rightarrow \{\mathcal{V}\}\phi$$

where in our case $I = Assumed \cup \{ValidCall_{op}\}$.

Lemma 8.12. *For an invariant ϕ and an operation op, if*

$$\models StrongOperationContract(opct; Assumed \cup \{ValidCall_{op}\}; \phi)$$

for some operation contract opct for op and if op fulfils opct under the assumption of Assumed then op preserves ϕ under the assumption of Assumed.

Example 8.13. Coming back to Ex. 8.11, `addBonus` could have the following operation contract:

$$Pre = \mathrm{true} \qquad Post = \mathrm{true} \qquad Mod = \{\texttt{self.bankCardPoints}\} \ .$$

Though this specification is quite weak it is in fact good enough to show that `addBonus` preserves (8.7):

$$\phi := \quad \forall c\text{:}\,\texttt{BankCard}. \ (c.\texttt{<created>} \doteq \mathrm{TRUE}$$
$$\Rightarrow 0 <= c.\texttt{wrongPINCounter} \ \& \ c.\texttt{wrongPINCounter} <= 2)$$

provided that the implementation of `addBonus` fulfils the contract. The latter has to be proven separately.

For the preservation of invariants the *StrongOperationContract* proof obligation is instantiated as follows:

$$\mathrm{true} \ \& \ \phi \ \& \ ValidCall_{op} \ \& \ \{\texttt{self.bankCardPoints} := f(\texttt{self})\} \, \mathrm{true}$$
$$\Rightarrow \{\texttt{self.bankCardPoints} := f(\texttt{self})\}\phi$$

where f is a rigid function. According to update application rules (and other simplification rules) this is equivalent to the tautology $\phi \Rightarrow \phi$. The lemma above ensures that `addBonus` preserves the considered invariant. Note again that we did not look at the implementation of `addBonus` to come to this conclusion; an inspection of the code is needed when the correctness w.r.t. *opct* is proven.

As a consequence, if the modifies clause of an operation is (provably) empty, there is no need to check *PreservesInv* for that operation, since this property is trivially true.

Lemma 8.14. *If all operation contracts opct applicable to an operation op have an empty modifies clause and* $\models$ *RespectsModifies(opct) then op preserves every invariant.*

Example 8.15. In our banking example, `cardIsInserted()` in class `ATM` does not modify anything (it is a *query* or *pure* method). Its modifies clause is thus empty in all applicable operation contracts. There is thus no need to check for invariant preservation for any invariant at this method.

8.5 Modular Verification

The just described basic proof obligation system (Lemma 8.10) requires to show the preservation of *all* invariants for *all* operations. In spite of the non-modular improvement described above, the basic problem persists: We have to take, for all invariants, *all* operations into consideration. The above improvement just saved us from examining the code of an operation. It does not help us with the problem of the explosion of proof obligations. We are now aiming at a more modular treatment: With a little effort, invariants themselves reveal that they are *relevant* to certain classes *only*. Then only operations in these classes must be checked to preserve these invariants. We describe two approaches which are commonly known as the visibility-based approach and the encapsulation-based approach.

8.5.1 Visibility-Based Approach

We reconsider the invariant (8.7) that the value stored in `wrongPINCounter` of `BankCard` is always greater or equal 0 and less or equal 2. We further assume that `wrongPINCounter` is declared as private. This means that only operations implemented in the class `BankCard` may modify that field. No other methods is able to set it to a different value; these methods thus trivially preserve the invariant. Thus, if all operations of `BankCard` take care that our invariant is preserved, we may conclude that *all* operations preserve it. The reason is that our invariant was *visible* to all classes which are relevant for it.

This example was about one of the simplest types of invariant we can imagine: A single private field was constrained to be in a certain range. We are now aiming at extending the visibility principle both to more complex invariants and to fields being declared as protected; remember that we are only dealing with private and protected fields.

For the former we can simply apply our argument of our example and come to the conclusion: If all used fields are private and all operations of the

classes which declare these fields preserve invariants, then all operations of the whole program preserve invariants. Implicit fields which cannot arbitrarily be assigned values do not need to be considered however. More precisely we can capture this with the notion of a *field depends clause* of an invariant.

Definition 8.16. *A field depends clause D of a closed formula ϕ is a set of (instance and static) non-implicit fields such that for all $f \in D$, for all $((d_1, \ldots, d_n), d') \in \alpha(f)$ and for all states S_1 and S_2 with $S_1(f)(d_1, \ldots, d_n) = S_2(f)(d_1, \ldots, d_n)$:*

$$S_1 \models \phi \quad \text{iff} \quad S_2 \models \phi \; .$$

Lemma 8.17. *Let ϕ be a formula and all occurring non-rigid function and predicate symbols are field symbols. Then the set of fields corresponding to these field symbols is a field depends clause.*

Lemma 8.18. *Let ϕ be a closed formula. Let D be a field depends clause of ϕ. If all fields in D are private and all classes declaring these fields preserve ϕ, then ϕ is preserved by all operations of the program.*

The argument for a proof of this lemma is as follows: During the execution of an operation of some class which does not declare some field of D all $f \in D$ are evaluated to the same value in every state because f cannot be assigned. In all other operations we assume the preservation of invariants however.

When we take *protected* fields into account, we must be more careful: Protected fields can only be assigned values in the class declaring it *or in a subclass*. We must thus check the preservation of invariants for *all subclasses* of classes declaring a protected field contained in the field depends clause. For an easier notation we introduce the notion of a *self-guard*:

Definition 8.19. *Let D be a set of not-implicit fields and G be a set of types. G is a self-guard of ϕ if it contains*

- *all classes declaring some $f \in D$ and*
- *all subclasses of all classes declaring some protected $f \in D$.*

With it the final condition for the visibility technique can be formulated:

Lemma 8.20. *Let ϕ be a closed formula. Let D be a field depends clause of ϕ. If there is a set G of types which is a self-guard of D and all operations of all classes in G preserve ϕ, then ϕ is preserved by all operations of the program.*

Example 8.21. For the introductorily mentioned invariant (8.7):

$$\phi := \quad \forall c{:}\texttt{BankCard.} \; (c.\texttt{<created>} \doteq \text{TRUE}$$
$$\rightarrow 0 \mathrel{<=} c.\texttt{wrongPINCounter} \;\&\; c.\texttt{wrongPINCounter} \mathrel{<=} 2)$$

we obtain, with the help of Lemma 8.17, the field depends clause

$$D := \{\texttt{wrongPINCounter}\} \ .$$

This field is declared private in `BankCard`. Thus {`BankCard`} is a self-guard of
D. If all operations of `BankCard` preserve ϕ then all operations of the whole
banking application will preserve it.

Assume now that `wrongPINCounter` is declared `protected` in `BankCard`
and there is a subclass `BankCardJunior` in our scenario. {`BankCard`} is then
no more a self-guard of D, we must extend it with the new subclass to

$$\{\texttt{BankCard}, \texttt{BankCardJunior}\} \ .$$

This entails that also all operations of `BankCardJunior` must preserve ϕ.

8.5.2 Encapsulation-Based Approach

Protecting invariants based on visibility is not always sufficient as the follow-
ing example extending our banking scenario shows, see Fig. 8.2.
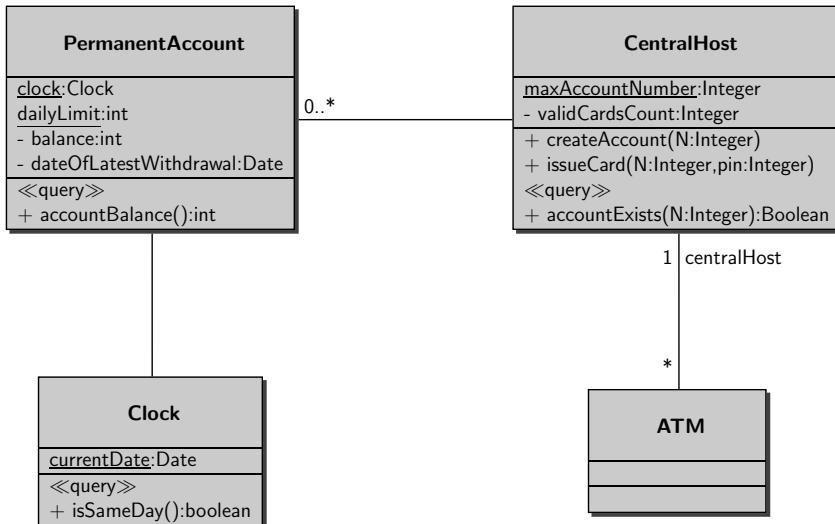
Fig. 8.2. Class diagram for extended ATM scenario

Example 8.22. The `PermanentAccount` class keeps track of the date at which
the latest withdrawal has been performed on this account. This is to ensure
that only a certain maximal amount is withdrawn each day. So that day is
stored in a field `dateOfLatestWithdrawal` of type `Date`. Let us assume for
simplicity that this class `Date` mainly consists of an integer counter `currDate`:

```
public class Date {
  private int currDate;
  public void setDate(int newDate) { currDate = newDate; }
  public int getDate() { return currDate; }
}
```

──── Java ────

(and maybe some convenience methods computing days, months, or years from it). Moreover there is a class `Clock` with a static field `currentDate` of type `Date`. A natural invariant of `PermanentAccount` is:

$$\forall \text{ Account } a; \; (a.\text{<created>} \doteq \text{TRUE}$$
$$\rightarrow \text{Clock.currentDate.value} \geq a.\text{dateOfLatestWithdrawal.value}) \tag{8.8}$$

With the visibility-based approach we would have to ensure that the classes `Account`, `Clock`, and `Date` preserve this invariant, since they are the classes in which all fields occurring in the invariant are declared in. While this task does not seem to be a problem for the former two[1], it is certainly impossible that `Date` preserves this invariant: The `setValue` method would permit the invariant to be violated. As soon as (more) general classes which are not only used in the context of the invariant (like `Date`) come into play the visibility strategy might fail. Note that also our "brute-force" method, that is, showing the preservation of all invariants for all operations, would have failed here.

   We can observe the following issue concerning our invariant: An object stored in the `dateOfLatestWithdrawal` field of a `PermanentAccount` instance a could be referenced (*aliased*) by another object o of some class T ($T \neq$ `Account`, $T \neq$ `Clock`). An operation m defined for o could violate our invariant, for instance by calling

```
... setValue(Clock.getCurrentDate().getValue()+1)
```

on it. If we do not want to check operation m as well for invariant preservation, we must control arbitrary leakage of certain object references. In our case it must be forbidden that a.`dateOfLatestWithdrawal` leaks to o. In other words, we must take care that there is a sufficient degree of *encapsulation*, which helps to protect invariants. Note, that an analogous requirement as for `dateOfLatestWithdrawal` in `PermanentAccount` must hold for `currentDate` in `Clock`, too.

---

[1] Note, that we might need additional invariants stating that the two `Date` objects are not the same. Moreover note that `Clock` will probably only increase the `value` counter.

In the following, we investigate *which* encapsulation properties must hold for which invariant and which consequences are implied for program verification; finally we discuss by which means encapsulation properties can be established.

Depends Clauses

First of all it is advantageous to have a more precise notion of the locations an invariant depends on than field depends clauses. We therefore extend them to full *depends clauses*. Instead of finding sets of *fields* for invariants, we are now aiming at *sets of terms*. As before, these sets of terms determine sets of locations. If these locations are not affected by a change from some state S_1 to some state S_2 then the invariant is evaluated to the same truth value both in s_1 and s_2.

Definition 8.23. *A depends clause D of a closed formula ϕ is a finite set of pairs $(\xi, f(t_1, \ldots, t_n))$ (where ξ is a formula, f a non-rigid function symbol, and $t_1, \ldots, t_n$ are terms[2] such that for all $(\xi, f(t_1, \ldots, t_n)) \in D$, for all states S_1 and S_2 and all variable assignments β with*

- $S_1(f)(val_{S_1,\beta}(t_1), \ldots, val_{S_1,\beta}(t_n)) = S_2(f)(val_{S_1,\beta}(t_1), \ldots, val_{S_1,\beta}(t_n))$ *and*
- $S_1, \beta \models \xi$

the following holds:
$$S_1 \models \phi \quad \textit{iff} \quad S_2 \models \phi \ .$$

The following lemma says that we can perform the visibility-based analysis also with depends clauses and not only with field depends clauses.

Lemma 8.24. *Let ϕ be a closed formula. A set DF of fields is a field depends clause iff*
$$\{(\text{true}, x.f) \mid f \in DF\}$$

is a depends clause of ϕ.

Depends clauses deliver us more detailed information on the dependencies of invariants. In particular we keep complete "access chains" like

.dateOfLatestWithdrawal.value

and not only the single and unrelated fields `dateOfLatestWithdrawal` and `value`.

Example 8.25. The invariant (8.8) in Ex. 8.22 has the following depends clause:

$$D := \left\{ \begin{array}{c} \texttt{Clock.currentDate.value}, \texttt{Clock.currentDate}, \\ x.\texttt{dateOfLatestWithdrawal.value}, \\ x.\texttt{dateOfLatestWithdrawal} \end{array} \right\} \qquad (8.9)$$

---

[2] Note that this is the same way to describe locations syntactically as for modifier sets in Sect. 3.7.4

Determining Depends Clauses

We have not yet said where depends clauses come from. Usually it is quite easy to extract them syntactically from the considered invariant.

Lemma 8.26. *Let ϕ be a formula which does not contain non-rigid predicates. Let $Dep(\phi)$ be the set of all subterms of ϕ which are of the shape $f(t_1, \ldots, t_n)$ with terms $t_1, \ldots, t_n$ where f is a non-rigid function symbol but not an implicit field. Then $Dep(\phi)$ is a depends clause of ϕ.*

Example 8.27. For the invariant (8.8) in Ex. 8.22 we obtain, with the procedure described in the above lemma, $Dep(\phi) = D$ where D is the set described in Ex. 8.25.

Example 8.28. The procedure does not always deliver an "optimal" depends clause. For the formula ϕ

$$\backslash\text{forall } T \ x; \ (x = \texttt{a.b} \mathbin{-\!\!>} x.\texttt{c} > 0)$$

where $\texttt{a}$ is a static field, we obtain $Dep(\phi) = \{\texttt{a}, \texttt{a.b}, \texttt{x.c}\}$ as depends clause. Since ϕ is however equivalent to $\texttt{a.b.c} > 0$ the set $D = \{\texttt{a}, \texttt{a.b}, \texttt{a.b.c}\}$ would as well be a depends clause. As can be seen after the next sections D is a better depends clause than $Dep(\phi)$.

An alternative to the above procedure is thus to let the developer find a depends clause which is then subject to be proven with the help of an appropriate proof obligation. Such a proof obligation can be found in [Roth, 2006].

Interior and Guards

Depends clauses D_ϕ of invariants ϕ deliver important information: If all locations described by them remain unchanged then the truth value of ϕ remains unchanged, too. These locations must thus only be changeable by operations from which we know that they preserve ϕ. We must take care that all other operations do not get a chance to directly modify the critical locations.

Let (f, o) be such a location, where f is an instance field symbol and o an object. Let us assume that o offers operations which could modify f. In that case the only possibility to prohibit that (f, o) is modified, is to avoid that o is referenced by a not trustable object. Given a state S, we call the objects o the *interior* of D_ϕ in S.

We thus need to distinguish between trustable and not trustable objects (w.r.t. ϕ). We trust in an object if and only if its operations preserve ϕ. Let G, the *guard*, be exactly that set of *classes* whose instances can be trusted. We will have to show (later) that these classes preserve ϕ, but we first want to establish the property that references to the interior of D_ϕ do not escape to not trusted objects. Not trusted objects are exactly those objects which

are not instances of some class in G. According to our discussion above they must not obtain a reference to the interior of the locations described by the depends clause of ϕ. It is a task of the guard classes to maintain this property.

Definition 8.29 (Interior, Guard). *Let L be the locations described by a depends clause D of the closed formula ϕ. Let G be a subset of the considered program context.*

- *The* interior *of D in a state S is the set In^S with*

$$\mathrm{In}^S := \{o'_i \mid (f, (o'_1, \ldots, o'_n)) \in L, 1 \le i \le n\} \ .$$

- *G is a* guard *for D in a state S if for all locations $(g, (o_1, \ldots, o_m))$ with $S(g)(o_1, \ldots, o_m) \in \mathrm{In}^S$ one of the following three alternatives holds:*
  - *If g is an instance field symbol with $\alpha(g) = T \times T''$ for some T and T'' then $T \in G$; if g is protected then for all T' with $T' \sqsubseteq T \colon T' \in G$.*
  - *If g is a static field symbol declared in type T then $T \in G$; if g is protected then for all T' with $T' \sqsubseteq T \colon T' \in G$.*
  - *$o_1, \ldots, o_m \in \mathrm{In}^S \cup \mathcal{D}^{PT}$ (where $\mathcal{D}^{PT}$ denotes the primitive values).*
- *G is a* guard *for D if it is a guard for D in all states.*

If we have found a guard for a depends clause of an invariant ϕ it is sufficient that just the operations of the guard classes preserve ϕ. Moreover we combine this with the self-guard approach.

Lemma 8.30. *Let P be a program. Let ϕ be a closed formula, $D = D_{sg} \uplus D_g$ a depends clause of ϕ, and $G \subseteq P$ a self-guard of D_{sg} and a guard of D_g. If all operations op of all classes in G preserve ϕ then all operations of all classes of P preserve ϕ.*

Example 8.31. The depends clause (8.9) of invariant (8.8) can be partitioned into the subsets D_{sg} and D_g with

$$D_{sg} = \{\texttt{Clock.currentDate}, x.\texttt{dateOfLatestWithdrawal}\}$$
$$D_g = \{\texttt{Clock.currentDate.value}, x.\texttt{dateOfLatestWithdrawal.value}\}$$

D_g has the guard

$$G := \{\texttt{PermanentAccount}, \texttt{Clock}\} \ .$$

Moreover G is a self-guard for D_{sg}. If all operations of G preserve (8.8) then *all* operations of our banking program preserve it.

Proving Guards with Encapsulation Predicates

We give a short glimpse at one method, using *encapsulation predicates* [Roth, 2005], how guardedness can be verified. For other ways to prove encapsulation see the following sidebar.

Verifying Encapsulation

Several possibilities exist to verify encapsulation properties, as needed to show guardedness. Techniques like *islands* [Hogg, 1991], *balloons* [Almeida, 1997], *uniqueness* [Boyland, 2001], and different types of *ownership* [Clarke et al., 1998, Müller, 2002, Boyapati et al., 2003] can be employed. These approaches are based on type systems and they all require that the programmer explicitly annotates the program to indicate the desired encapsulation. By type-checking the annotations one can conclude completely automatically that a certain kind of encapsulation is established. Weiß [2006] and Roth [2006] show that it is possible to make use of an ownership approach like *Universes* of Müller [2002] to check formulae containing encapsulation predicates. Other static analyses for checking encapsulation which do not require explicit extensive annotations are rare (e.g., [Burrows, 2005]).

The first-order fragment of JAVA CARD DL gives us almost all means to specify encapsulation. There is however at least one predicate missing to conveniently express the desired property: the binary Acc predicate. $Acc(o_1, o_2)$ formalises that there is at least one field a defined for o_1, an *access*, such that $o_1.a = o_2$. If all fields of a program are known then it is even possible to replace $Acc(o_1, o_2)$ with a big disjunction of formulae $o_1.a \doteq o_2$ over all fields a. This is however impossible in the context of open programs (see below).

With Acc we can define more specific predicates describing encapsulation properties. A useful property is that there are accesses to objects (or values) z satisfying a formula $p(z)$ exclusively from a set of objects y satisfying $g(y)$. This property is formalised by the 0-ary predicates $Enc_{y,z}[g(y), p(z)]$ for all formulae $g(y)$ and $p(z)$:

$$Enc_{y,z}[g(y), p(z)] \quad :\Leftrightarrow \quad \text{\forall Object } y; \text{ \forall Object } z;$$
$$(Acc(y, z) \& p(z) \rightarrow p(y) \mid g(y))$$

To simplify matters we assume that the only non-rigid functions occurring in our formulae are field accesses. Roth [2006] extends the approach to arrays. Guardedness of a set of types G w.r.t. a depends clause D of a formula ϕ can then be formalised as the conjunction of the m formulae (for $k = 1, \ldots, m$):

$$\phi_{enc} \quad :\Leftrightarrow \quad cl_\forall \Big(Enc_{y,z} \big(\text{Conj}_{instance}, \big(\phi_k \& z \doteq d'_k \big) \big) \Big)$$

where

- G is a set of types,
- $D = \{d_1, \ldots, d_m\}$ is a depends clause where $d^k = (\phi_k, d'_k.f)$ where f is a field,
- $cl_\forall$ universally quantifies the free occurrences of variables in d^k and ϕ_k for all $k = 1, \ldots, m$.

- $\text{Conj}_{instance}$ is the conjunction of the formulae $y \sqsubseteq C$ over all $C \in G$.

If we assume that no supertype of G declares fields, ϕ_{enc} is our desired property; if it holds in any state s: G is a guard for D in S. It remains to show that ϕ_{enc} is in fact an invariant of P. Roth [2006] shows that it is sufficient to show the preservation of ϕ_{enc} for the operations of G provided that the depends clause satisfies certain syntactic criteria.

8.5.3 Verification Strategies

This section summarises the results from above about the observed-state entire correctness of a closed program. We have seen that there are several ways to establish correctness, in particular the correctness of invariants. In the end, it is the choice of the individual developer which of the different ways or *verification strategies* should be taken.

For all operations *op* and all operation contracts *opct* in S for *op*:

- $EnsuresPost(opct; I)$,
- $RespectsModifies(opct; I)$,   for some $I \subseteq Inv_{\text{Spec}}$

and for all invariants $\phi \in Inv_{\text{Spec}}$:
$InitInv(\phi)$ *and* one of two following conditions holds:

- there is depends clause D for ϕ and
  there is set of types G and
  there are sets D_1 and D_2 with $D = D_1 \uplus D_2$ and
  – G is a self-guard for D_1 and
  – G is guard for D_2
  and
  for all operations *op* in G: $PreservesInv(op; I; \{\phi\})$ for some $I \subseteq Inv_{\text{Spec}}$
- for all operations *op* in P: $PreservesInv(op; I; \{\phi\})$ for some $I \subseteq Inv_{\text{Spec}}$

As alternative to $PreservesInv(op; I; \{\phi\})$ one can show:
$StrongOperationContract(opct; I; \phi)$ for some operation contract *opct* for *op*

Fig. 8.3. Verification strategies for entire observed-state correctness

Fig. 8.3 shows the proof obligations that must be proven for entire observed-state correctness. We must ensure that operation contracts are fulfilled and that invariants are preserved. With both tasks there are several options:

- For every proof obligation we can choose which invariants we assume before a call to an operation. For establishing observed-state correctness it is possible to assume all invariants. This however complicates the use of the proof as a lemma since the check for the applicability of the lemma becomes more complex. It is thus advisable to assume as few invariants as possible.

- There are two principal possibilities to establish the preservation of invariants. Either we restrict the number of operations which we check for the preservation with the help of guards and self-guards, or we check all operations of the whole program. If we decide for the former we have to choose between self-guards (*visibility* principle) and guards (*encapsulation* principle). The former works mainly for simple invariants which do not involve the state of general purpose classes. Whenever possible one should opt for this way, since proving encapsulation entails additional non-trivial proof obligations.

  As depicted in Fig. 8.3 it is possible to closely combine visibility and encapsulation principles: The depends clause of the considered invariant is, because of Lemma 8.24, basis for both. It is partitioned into two sets of terms. For the first we show that a self-guard is available, for the second a guard must be found.

- Finally, in order to prove that a concrete operation preserves a concrete invariant there are two alternatives:
  - Symbolic execution of the code with *PreservesInv*.
  - Analysis of the invariant with the help of *StrongOperationContract* using precondition, postcondition, and the modifies clause. If the operation contract is "sufficiently functionally complete" this proof obligation can conclude that the operation preserves the invariant without symbolically executing code. This way works also if the locations relevant for the invariant and the locations which are modified by the operation are distinct. Note, that symbolic execution of code takes place when we check that the operation fulfils its operation contracts.

8.5.4 Components and Modular Proofs

So far we have fixed a program context as the object of our verification effort. In a typical software engineering project this is however an unusual situation. Typically, *components* are programmed independently, with the goal to be composed with other components. For our purposes we use the term component for a set of classes which are developed and distributed as a unit to be composed with other components to form a complex application program.

Unfortunately, the notion of observed-state correctness as defined in Sect. 8.2 is not sufficient for these purposes as demonstrated in the following example:

Example 8.32. Consider again the invariant (8.7) which said that the wrong PIN counters of all BankCards should be between 0 and 2. Assume we have proven, as required by observed-state correctness that all methods of the considered component, including BankCard preserve this invariant. We may now use this component in a context which contains a class MyBankCard which is a subclass of BankCard and overrides a method m of BankCard. Let us

assume that the field `wrongPINCounter` was declared `protected` in `BankCard` such that it can be directly modified in the overridden version in `MyBankCard`. And in fact we override m in a way that we set the `wrongPINCounter` field to 5. Clearly, when running the composed program, we end up in a state, observable by an observer, which does not satisfy our invariant. Note, that the invariant also talks about subclasses of `BankCard`! So the mere preservation of invariants by the component methods was not sufficient, we have to be more strict.

Our definition of observed-state correctness must thus be adapted slightly. First we adapt the notion of reachable states. We must take into account that the pre-states we consider can be reached by an extended state space; for instance we could have to deal with subclasses of our components.

Definition 8.33 (refining Def. 8.3). *A state S is* reachable *by a program P if there is a set of classes P^{cl} such that at the end of the execution of a method of $Obs := P^{cl} \backslash P$ the state S is reached.*

This change triggers re-definitions of the fulfilment of operation contracts and the preservation of invariants.

This leads to the notion of *durable* observed-state correctness. It extends observed-state correctness. With that correctness, observers need to ensure invariants of the components before the call. Now we demand that invariants hold in any intermediate state (or equivalently at the end of an arbitrary method) in the observer. So for an observer, invariants of an observed component always hold.

Definition 8.34 (Durable Observed-State Correctness). *Let* Spec *be a specification of a component P. Suppose P^{cl} is an arbitrary super set of P consisting of types. Set further $Obs := P^{cl} \backslash P$. Furthermore we assume that every method or constructor op of P with an operation contract in* Spec *is called by Obs only in a state where the precondition of at least one fitting operation contract from* Spec *is satisfied.*

P is durable observed-state correct *w.r.t.* Spec *if it is observed-state correct and all invariants of P hold after an arbitrary method of Obs has terminated.*

Example 8.35. The component considered in the previous example is not durable observed correct since a method, namely the overridden one, of the observer $Obs := \{$`MyBankCard`$\}$ results in a state in which our invariant does not hold.

This notion imposes very strict conditions on the system to be verified. A system which is durable observed-state correct will thus not or only hardly be adaptable to new re-use contexts. For example it will not be possible to override methods which are critical to invariants of a specification. Yet overriding for the purpose of adapting behaviour is an integral part of object-oriented

development. The only way out of the dilemma verification vs. adaptability is to impose requirements on the context in which the component may safely be used. We obtain a notion of *relative* observed-state correctness: only in contexts which satisfy certain requirements, a component behaves as specified in its specification. This idea is refined in [Roth, 2006]. In our example we would have required that classes overriding `BankCard` in the context must preserve invariant (8.7).

The JAVA CARD DL calculus requires a fixed program context. By adding additional components, valid formulae can become invalid. In [Roth, 2006] a solution to this problem is provided. The idea is to introduce a new notion of validity and soundness. According to this notion, formulae are valid only if they are valid (in the original sense) in all type hierarchies extending the "core" type hierarchy. This classifies some rules of the JAVA CARD DL calculus as unsound in this sense. They may not be used when considering components. Only few rules are affected, the most relevant being the rule simulating dynamic method dispatching ($\Rightarrow$ Sect. 3.6.5). It is sound in a particular fixed context but not if classes (overriding the method) are added to the context.

The way out is that we do not offer the rule which replaces a method call by an if-cascade discriminating between all method implementations in the context. Instead a new rule inserts the specification attached to the method contract of the (static) type T of the receiver object ($\Rightarrow$ Sect. 3.8). Of course this does not guarantee that all subclasses of T fulfil this contract. Moreover, in component-based development, it is not possible to know about all subclasses and their implementation. We must thus impose constraints on contexts relative to which formulae are valid. Again generic extension contracts are an instrument to impose such constraints on unknown contexts [Roth, 2006].

Roth [2006] also defines a proof obligation system for durable observed-state correctness, which makes extensively use of the visibility and the encapsulation approach.

9

From Sequential JAVA to JAVA CARD

by
Wojciech Mostowski

9.1 Introduction

The JAVA CARD dialect of JAVA is often thought of as a subset of JAVA. JAVA
CARD is used to program smart cards, and due to the limited nature of smart
cards JAVA CARD is a much simpler programming language than JAVA; cur-
rently, there is no concurrency in JAVA CARD, floating point arithmetic, or
dynamic class loading. Because of these simplifications, verification of smart
card applications written in JAVA CARD is substantially easier than verifica-
tion of full-featured JAVA applications. However, there are some smart card
specific features in JAVA CARD that are not present in standard JAVA, namely
object persistency and an atomic transaction mechanism. Another way to
put it is that, technically, JAVA CARD is not a subset of JAVA, it is a *superset*
of a *subset* of JAVA. This chapter explains these JAVA CARD specific features
and describes extensions to the basic JAVA CARD DL to handle them. In
this chapter we assume that the reader is already familiar with basic JAVA
CARD DL presented in Chapter 3.

Object persistency is an important issue when a possible abrupt termina-
tion of a JAVA CARD applet running on a smart card is considered. Such an
abrupt termination can occur due to an unexpected power loss, caused by,
for example, ripping the smart card out of the card terminal—a *card tear*,
or, as it is also referred to, *card rip-out*. For such situations we would like to
be able to establish that the consistency of the persistent data stored on the
card is preserved. On the programming language side JAVA CARD technology
provides the atomic transaction mechanism—a program construct available
to the programmer to ensure that an arbitrary piece of JAVA CARD program is
executed in one atomic step. On the logic side, to be able to reason about con-
sistency properties of persistent data, it is necessary to introduce the notion
of a strong invariant and deal with the intricacies of the transaction mech-
anism. The manifestation of strong invariants in the logic is the throughout

modality ($\llbracket \cdot \rrbracket$), and conditional assignments inside transactions are modelled by updates of a new kind (*shadowed updates*).

In the next section we elaborate on the motivation for introducing strong invariants and support for transactions in Java Card DL and in Section 9.3 Java Card's object persistency and transaction mechanism are described. Section 9.4 discusses strong invariants and their treatment in the logic with throughout modality in detail. Section 9.5 presents the logic extension to deal with Java Card transaction statements and conditional assignments. In Section 9.6 sample proofs are given presenting the use of the new calculus rules. Section 9.7 presents further extensions to the logic to properly handle two specific Java Card library methods, `arrayCopyNonAtomic` and `arrayFillNonAtomic`, and Section 9.8 summarises the theoretical contents of this chapter. Finally, Section 9.9 discusses some details about the taclet implementation of the throughout and transaction rules.

9.2 Motivation

The main motivation to introduce strong invariants and support for transactions into Java Card DL resulted from the analysis of a Java Card case study described by Mostowski [2002]. The case study involves a Java Card applet that is used for user authentication in a Linux system instead of the password mechanism. After analysing the application and testing it, the following observation was made: the Java Card applet in question is not "tear safe". That is, it is possible to destroy the applet's functionality by removing (tearing) the Java Card device from the card reader (terminal) during the authentication process. The applet's memory is corrupted and it is left in an undefined state, causing all subsequent authentication attempts to be unsuccessful. Fortunately, this particular error causes the applet to become useless but does not allow unauthorised access. In a general case however, we should take the worst case scenario under consideration. Thus it is clear that, to avoid such errors, we have to be able to specify and verify the property that a certain invariant is maintained at all times during the applet's execution, and, in particular, in case of an abrupt termination. Standard OCL or JML invariants do not suffice for this purpose, because in principle their semantics is that if they hold before a method is executed then they hold after the execution of a method. Normally, it is not required for an invariant to hold in the intermediate states of a method's execution. To solve this problem, we introduce *strong invariants*, which allow to specify properties about all intermediate states of a program.

For example, the following strong invariant (expressed in pseudo OCL) says that we do not allow partially initialised PersonalData objects at any point in our program. In case the program is abruptly terminated, we should end up with either a fully initialised object or an uninitialised (empty) one:

```
─── OCL ──────────────────────────────────────────────
context PersonalData
stronginv: not self.empty implies self.firstName <> null
           and self.lastName <> null and self.age > 0
───────────────────────────────────── OCL ───
```

To be able to reason about such properties in the KeY system, Java Card DL needs to be extended to support strong invariants. What follows is that a proper notion of atomicity of Java Card programs needs to be established. Since Java Card transaction mechanism provides the programmer with the possibility to make arbitrary program blocks atomic, the transaction mechanism also has to be fully supported in the logic.

9.3 Java Card Memory, Atomicity, and Transactions

Here we describe the aspects of object persistency and transaction handling within the Java Card platform relevant to this chapter. A full description of the transaction mechanism can be found in [Chen, 2000, Sun, 2003b,c,d].

The memory model of Java Card differs slightly from Java's model. In smart cards there are two kinds of writable memory: persistent memory (EEPROM), which holds its contents between card sessions, and transient memory (RAM), whose contents disappear when a power loss occurs, in particular, when the card is removed from the card reader. Thus, every memory element in Java Card (a variable or an object field) is either persistent or transient. Based on the Java Card language specification the following simplified rules can be given:

- All objects (including the currently running applet, as well as this object, and arrays) are stored in persistent memory. Thus, in Java Card all assignments like "o.attr = 2;", "this.a = 3;", and "arr[i] = 4;" have permanent character; that is, the assigned values will be kept after the card loses power.
- A programmer can create an array with transient elements by calling a certain method from the Java Card API (for example, JCSystem.make-TransientByteArray), but currently there is no possibility to make objects (fields) other than array elements transient. Moreover, some Java Card system owned arrays (like the APDU buffer) are transient.
- All local variables are transient.

The distinction between persistent and transient objects is very important since these two types of objects are treated in a different way by Java Card's transaction mechanism. The following are the Java Card system calls for transactions with their description:

- `JCSystem.beginTransaction()` begins an atomic transaction. From this point on, all assignments to fields of persistent objects are executed conditionally, while assignments to transient variables or array elements are executed unconditionally.
- `JCSystem.commitTransaction()` commits the transaction. All conditional assignments are committed (in one atomic step).
- `JCSystem.abortTransaction()` aborts the transaction. All conditional assignments are rolled back to the state in which the transaction started.[1] Assignments to transient variables and array elements remain unchanged (as if performed outside a transaction). Also, references to all objects created inside the transaction are reset to `null`.

As an example to illustrate how transactions work in practice, consider the following fragment of a Java Card program:

—— Java Card ——————————————————————————

```
this.a = 1;
int i = 0;
JCSystem.beginTransaction();
  this.a = 2;
  i = this.a;
  SomeClass c = new SomeClass();
JCSystem.abortTransaction();
```

—————————————————————————————— Java Card ——

After the execution of this program, the value of persistent `this.a` is still 1 (value before the transaction), while the value of local (and thus transient) `i` is now 2 (the value it was assigned during the transaction). The value of `c` is `null`, even though `c` is a local variable and was updated unconditionally inside the transaction.

A transaction can be aborted explicitly by the programmer, like in the example above, but also implicitly by the Java Card Runtime Environment (JCRE), when a transaction cannot be completed due to lack of resources or an unexpected program termination (for example, card tear). In the first case, the Java Card program continues its execution with the assignments performed inside the transaction rolled back, while in the second case the program is terminated immediately and the updates are rolled back during the transaction recovery process next time the Java Card applet is initialised. The possibility of an explicit transaction abort has important consequences for the design of the logic to handle transactions, as we will see later in the chapter.

Transactions do not have to be interleaved properly with other program constructs, for example, a transaction can be started within one method and

---

[1] This definition is not entirely accurate, we will refine it later in Section 9.7, where we discuss *non-atomic* Java Card API methods.

committed within another method. However, transactions must be nested properly with each other. In the current version (2.2.1) of JAVA CARD [Sun, 2003c] the nesting depth of transactions is restricted to 1—only one transaction can be active at a time.

Considering the persistent objects, the whole program block inside the transaction is seen by the outside world as if it were executed in one atomic step. This also has a consequence for the design of the logic—multiple assignment statements inside a transaction will have to be regarded as a single atomic step by the logic, while outside of the transaction each single primitive assignment will be treated atomically.

9.4 Strong Invariants: The "Throughout" Modality

In some regard the basic JAVA CARD DL (and other versions of Dynamic Logic, as well as all sorts of Hoare logics) lacks expressiveness—the semantics of a program is a relation between states; formulas can only describe the input/output behaviour of programs. Basic JAVA CARD DL cannot be used to reason about program behaviour not manifested in the input/output relation. Therefore, it is inadequate for verifying strong invariants that must be valid throughout program execution.

Following Beckert and Schlager [2001], to overcome this deficiency and increase the expressiveness of JAVA CARD DL we added a new modality $[\![\cdot]\!]$ ("throughout") to the logic. In the extended logic, the semantics of a program is the sequence of all states its execution passes through when started in the current state (its *trace*). Using $[\![\cdot]\!]$, it is possible to specify properties of the intermediate states of terminating and non-terminating programs. And such properties (typically strong invariants and safety constraints) can be verified using the JAVA CARD DL calculus extended with additional rules for $[\![\cdot]\!]$ shortly to be presented in Section 9.4.1.

A "throughout" property (formula) has to be checked after every single field or variable assignment, that is, the rules for the throughout modality will have more premises and branch more frequently. According to the JAVA CARD runtime environment specification [Sun, 2003c], each single field or variable assignment is atomic. This matches exactly JAVA CARD DL's notion of a single update. Thus, a "throughout" property has to hold after every single JAVA CARD DL update.

However, since the transaction mechanism can be used to ensure atomicity of arbitrary program blocks, such additional checks have to be suspended for the assignments that appear inside a transaction. This is described in detail in Section 9.5.

In practice, only formulas of the form $\psi \ \& \ \phi \implies [\![p]\!]\phi$ will be considered, ϕ being a strong invariant for p and ψ being the necessary preconditions for proper execution of p. If transient arrays are involved in ϕ (explicitly or implicitly), one also has to prove $\phi \implies \langle \texttt{initAllTransientArrays();} \rangle \phi$,

that is, that after a card tear the re-initialisation of transient arrays preserves the strong invariant.

9.4.1 Additional Calculus Rules for the Throughout Modality

As far as the basic JAVA CARD DL calculus is considered, the new throughout modality is almost exactly the same as the box modality. What is required of the throughout formula $[\![p]\!]\phi$ in addition, is that ϕ holds after every single primitive assignment of p. This means that the assignment rule for the throughout modality has to be extended compared to the box rule. The only other rules that are affected by the new semantics of the throughout modality are the invariant loop rules. Otherwise, when transactions are not considered, the rules for the throughout modality are exactly the same as for the box modality $[\cdot]$. Thus, here we discuss the assignment rule and the while invariant rule for the throughout modality, while the rules to support transactions are presented in the next section.

   Note, that the rules we are about to present in this chapter are in a simplified form. In particular, each complex assignment in JAVA CARD DL is treated by a sequence of rules, which, among other things, unfold complex expressions and take care of possible abrupt termination. Here, we omit these details as they are irrelevant for our purposes. All the rules have been implemented in the KeY system in their full form though.

The Assignment Rule for $[\![\cdot]\!]$

Overall, the assignment rules for the throughout modality follow the same 4-step process, which has been described for regular assignments in Section 3.6.2. Important differences occur during Step 4 (generating an update from the assignment), on which we will concentrate here. We will simplify the case even further and disregard the possibility of exceptions at this stage.

   In JAVA CARD an assignment $v = se$; (where v is a variable and se is a simple expression) is an atomic program and can be executed in a single step. Its semantics is a trace consisting of the initial state s (before the assignment) and the final state s' (after the assignment). Therefore, the meaning of $[\![v = se;]\!]\phi$ is that ϕ is true in both s and s', which is what the two premisses of the following assignment rule express:

$$\mathsf{assignTout} \quad \frac{\Longrightarrow \phi \qquad \Longrightarrow \{v := se\}[\![\pi\ \omega]\!]\phi}{\Longrightarrow [\![\pi\ v = se;\ \omega]\!]\phi}$$

The left premiss states that the formula ϕ has to hold in the state s before the assignment takes place. The right premiss says that ϕ has to hold in the state s' after the assignment—and in all states thereafter during the execution of the rest of the program ω.

It is easy to see that using this rule causes some extra branching of the proofs involving the $[\![\cdot]\!]$ modality. This branching is unavoidable due to the fact that the strong invariant has to be checked for each intermediate state of the program execution. However, many of those branches, which do not involve JAVA CARD programs any more, are very simple and can be discharged quickly. An equivalent rule to assign Tout is the following:

$$\text{assign ToutOpt} \quad \frac{\Longrightarrow \phi \mathbin{\&} (\phi \rightarrow \{v := se\} [\![\pi \; \omega]\!]\phi)}{\Longrightarrow [\![\pi \; v \; = \; se; \; \omega]\!]\phi}$$

In this form, the rule is more efficient when used in practice in the KeY prover, because unnecessary branching can be avoided.

The While Rule for $[\![\cdot]\!]$

Another essential programming construct, where the rule for the $[\![\cdot]\!]$ modality differs from the corresponding rule for the $[\cdot]$ modality, is the `while` loop. A detailed description was already given of how the `while` invariant rule for JAVA CARD DL works, including the proper treatment of possible abrupt termination inside the loop (with the `break` and `continue` statements among others), and loop modifier sets ($\Rightarrow$ Chap. 3, Sect. 3.7.1). Here we only give a simplified version of the `while` rule for the throughout modality referring the reader to Section 3.7.1 for the technical details, which are essentially the same as for the box invariant rule. As in the case of the `while` rule for the $[\cdot]$ modality the user has to supply a loop invariant *Inv*. Intuitively, the rule establishes the following:

1. In the state before the loop is executed, some invariant *Inv* holds (*invariant is initially valid*).
2. If the loop condition is true, at the end of a single execution of the loop body the invariant *Inv* has to hold again (*loop body preserves the invariant*).
3. Provided *Inv* holds, the formula ϕ has to hold during and continuously after loop body execution in all of the following cases: (i) when the loop body is executed once and terminates normally, (ii) when the loop body terminates abruptly (by `break`, `continue`, or throwing an exception) resulting in a termination of the whole loop, and (iii) when the loop body is not executed (the loop condition is not satisfied).

Formally, the `while` rule for $[\![\cdot]\!]$ is the following:

$$\text{whileTout} \quad \frac{\begin{array}{l} \Longrightarrow \mathcal{U}\,Inv \\ Inv, \; a \doteq \text{TRUE} \Longrightarrow [p]\,Inv \\ Inv, \; a \doteq \text{TRUE} \Longrightarrow [\![\pi \; p \; \omega]\!]\phi \\ Inv, \; a \doteq \text{FALSE} \Longrightarrow [\![\pi \; \omega]\!] \end{array}}{\Longrightarrow \mathcal{U}[\![\pi \; \texttt{while}(a)\{p\} \; \omega]\!]\phi}$$

The four premisses of this rule establish the conditions listed above: The first premiss establishes the first condition, the second premiss establishes the second condition, the third premiss establishes sub-conditions (i) and (ii) of the third condition, and finally, the fourth premiss establishes sub-condition (iii) of the third condition.

9.5 Handling Transactions in the Logic

Up till now we only considered one level of atomicity when dealing with the semantics of the throughout modality, namely, that only the primitive assignments are atomic. But as we said already, the Java Card transaction mechanism can be used to ensure atomicity of larger blocks of the program. This is one issue that the logic has to deal with. The second and equally important issue is the fact that a transaction can abort and roll back the assignments to persistent data that appeared inside the transaction. When the transaction is aborted explicitly by the programmer, the program continues its execution after the abort. It is very important to note that an aborted transaction influences the final state of the program. What follows, is that the semantics of the diamond and box modality (total and partial correctness) has to account for the transaction mechanism as well. One other way of looking at it is to consider the transaction abort statement as an *undoing assignment*, and, being (a special kind of) assignment it has to be included in the semantics of all the modal operators: diamond, box, and throughout.

The basis of the solution for transactions consists of two parts. First, whenever a transaction is encountered in the Java Card program, the proof is split into two branches. One of the branches is responsible for analysing the program under the assumption that the transaction will commit (*commit branch*) and the second branch assumes that the transaction will abort (*abort branch*).[2] The second part of the solution is to mark the modality (respectively, the program in the modality) with a tag indicating that a transaction is in progress so that different rules (depending on the *commit* or *abort* assumption) for assignment can be applied. In particular a special assignment rule to deal with assignment roll-back is used on the abort branch, which selectively performs assignments to mimic assignment *undoing*. We explain this in the following sections.

9.5.1 Rules for Beginning and Ending a Transaction

Transactions in Java Card do not have to be interleaved properly with other program constructs (although a good programming practice would suggest they should). Thus we cannot assume that we will be able to "look ahead"

---

[2] This proof splitting is in some sense a very special kind of a cut/case split rule, which has to account for the two possible transaction termination scenarios.

in the verified program to see where a transaction is finished. In particular, a transaction terminating statement can appear in more than one place, for example, in two branches of the if statement. Thus, enclosing a transaction in a block with a separate set of rules for that kind of block, like it is done for method calls with the method-frame ($\Rightarrow$ Sect. 3.6.5), is not possible.

Following this motivation, our rules have to be aware of the Java Card transaction statements and the current state of the transaction mechanism (that is, whether there is an active transaction at a given program point or not). We now explain how this is achieved.

Additional Constructs

First, we introduce some new programming constructs (*distinguished* method calls) to Java Card DL. The three distinguished method calls we need are the following:

- jvmBeginTransaction—low-level Java Card call to begin a transaction,
- jvmCommitTransaction—low-level Java Card call to end a transaction with a commit,
- jvmAbortTransaction—low-level Java Card call to end a transaction with an abort.

These methods are used in the proof when the transaction is started, respectively, finished in the Java Card program. They are only part of the rules and not of the Java Card programming language. Thus, for example, when a transaction is started in a Java Card program by a call to JCSystem.beginTransaction() the following implementation of begin-Transaction is assumed in Java Card DL:

—— Java Card ——————————————————
```java
public class JCSystem {
  private static short _transactionDepth = 0;

  public static void beginTransaction()
                      throws TransactionException {
    if(_transactionDepth > 0)
      TransactionException.throwIt(
        TransactionException.IN_PROGRESS);
    _transactionDepth++;
    de.uka.ilkd.key.javacard.KeYJCSystem.
      jvmBeginTransaction();
  }
  ...
}
```
————————————————————— Java Card ——

And similarly for `commitTransaction` and `abortTransaction`. Thus, when we encounter any of `jvmBeginTransaction`, `jvmCommitTransaction`, or `jvmAbortTransaction` in our proof we can assume they are properly nested.

The Main Idea

When a call to `jvmBeginTransaction` is encountered during the symbolic execution, the proof is split into two branches. In the first branch the program is analysed with the assumption that the transaction will commit, in the second branch it is assumed that the transaction will be aborted. Later, when a `jvmAbortTransaction` statement is encountered on the commit branch, the branch is simply discarded. The same exact thing happens in the opposite situation, i.e., when a `jvmCommitTransaction` is encountered on the abort branch.

   On the calculus level, a rule for `jvmBeginTransaction` splits the proof into two branches, and each branch (more precisely, the modality containing the program) is marked with a tag indicating what kind of transaction finish is assumed. The two tags are:

- TRC—a transaction is in progress and is assumed to commit,
- TRA—a transaction is in progress and is assumed to abort.

Depending on the tag different rules for assignment are applied. Making the distinction between the commit and the abort case is very helpful in handling the assignments inside the transaction. Since we assume that the transaction is going to commit on the first branch, we do not have to worry about keeping the backup copies of the modified data and can commit all the changes as we encounter them. Conversely, on the abort branch, we know that the assignments eventually (upon encountering `jvmAbortTransaction`) will have to be rolled back, so we can choose not to perform them in the first place.

   There is, however, a complication: in JAVA CARD only the assignments to the persistent data are rolled back, the assignments to transient data are always performed unconditionally. Moreover, conditionally updated persistent values may be used to update transient variables. Thus, we cannot simply ignore the assignments inside the abort branch. Instead, we operate on backup (also called shadow) copies of persistent data, keeping the original persistent data unmodified, while the updates to transient objects are always performed on the original data. This is explained in the upcoming section on conditional assignment ($\Rightarrow$ Sect. 9.5.2).

Rules for Beginning a Transaction

We have already stated that the transaction mechanism affects the semantics of all modal operators in our logic. Thus, for each of the three modal operators currently used in the logic ($\langle \cdot \rangle$, $[\cdot]$, $[\![\cdot]\!]$) there is one `jvmBeginTransaction` rule:

$$\text{beginTransTout } \frac{\Longrightarrow \phi \qquad \Longrightarrow [\![\text{TRC:}\pi\ \omega]\!]\phi \qquad \Longrightarrow [\![\text{TRA:}\pi\ \omega]\!]\phi}{\Longrightarrow [\![\pi \text{ jvmBeginTransaction(); } \omega]\!]\phi}$$

$$\text{beginTransDia } \frac{\Longrightarrow \langle \text{TRC:}\pi\ \omega \rangle\phi \qquad \Longrightarrow \langle \text{TRA:}\pi\ \omega \rangle\phi}{\Longrightarrow \langle \pi \text{ jvmBeginTransaction(); } \omega \rangle\phi}$$

$$\text{beginTransBox } \frac{\Longrightarrow [\text{TRC:}\pi\ \omega]\phi \qquad \Longrightarrow [\text{TRA:}\pi\ \omega]\phi}{\Longrightarrow [\pi \text{ jvmBeginTransaction(); } \omega]\phi}$$

In case of the $[\![\cdot]\!]$ operator the following things have to be established. First of all, ϕ has to hold before the transaction is started. Then we split the proof in two cases: the transaction will be terminated by a commit, or the transaction will be terminated by an abort. In both cases the formulas are marked with the proper tag, so that corresponding rules can be applied later, depending on the case. The $\langle \cdot \rangle$ and $[\cdot]$ rules for `jvmBeginTransaction` are very similar to $[\![\cdot]\!]$ except that ϕ does not have to hold before the transaction is started.

Rules for Committing and Aborting Transactions

Note that, apart from the modality, the `jvmBeginTransaction` rules for diamond and box are exactly the same. For exiting the transaction (by commit or abort) the rules are the same for all three operators, so we only quote the $[\![\cdot]\!]$ rules.

The first two rules apply when the expected type of termination is encountered ("TRC" for commit, respectively, "TRA" for abort). In that case, the corresponding transaction tag is simply removed, which means that the transaction is no longer in progress. These are the rules:

$$\text{commitTransExp } \frac{\Longrightarrow [\![\pi\ \omega]\!]\phi}{\Longrightarrow [\![\text{TRC:}\pi \text{ jvmCommitTransaction(); } \omega]\!]\phi}$$

$$\text{abortTransExp } \frac{\Longrightarrow [\![\pi\ \omega]\!]\phi}{\Longrightarrow [\![\text{TRA:}\pi \text{ jvmAbortTransaction(); } \omega]\!]\phi}$$

We also have to deal with the case where the transaction is terminated in an unexpected way, that is, a commit is encountered when the transaction was expected to abort and vice versa. In this case we simply use an axiom rule, which immediately closes the proof branch. One of the proof branches produced by the `jvmBeginTransaction` rule will, eventually, always become obsolete since each transaction can only terminate by either commit or abort. The rules are the following:

$$\text{commitTransUnexp } \frac{}{\Longrightarrow [\![\text{TRA:}\pi \text{ jvmCommitTransaction(); } \omega]\!]\phi}$$

$$\text{abortTransUnexp } \frac{}{\Longrightarrow [\![\text{TRC:}\pi \text{ jvmAbortTransaction(); } \omega]\!]\phi}$$

9.5.2 Rules for Conditional Assignment

Finally, we come to the essence of conditional assignment handling in our rules. When the transaction is expected to commit, no special handling is required—all the assignments are executed immediately. Thus, the rule for an assignment in the scope of $[\![\mathsf{TRC}\colon\ldots]\!]$ is the same as the rule for an assignment within $[\cdot]$. Note that, even using the $[\![\mathsf{TRC}\colon\ldots]\!]$ modality, ϕ only has to hold at the end of the transaction, which is considered to be atomic:

$$\mathsf{assignTRC}\ \dfrac{\Longrightarrow \{v := se\}[\![\mathsf{TRC}\colon\!\pi\ \omega]\!]\phi}{\Longrightarrow [\![\mathsf{TRC}\colon\!\pi\ v\ \texttt{=}\ se\texttt{;}\ \omega]\!]\phi}$$

When a transaction is terminated by an abort, all the conditional assignments are rolled back as if they were not performed. Since it is assumed that the transaction is going to abort (because of a TRA tag), we can deliberately choose not to perform the updates to persistent objects as we encounter them. However, we cannot simply skip them since the new values assigned to (fields of) persistent objects during a transaction may be referred to later in the same transaction (before the abort). The idea to handle this, is to assign the new value to a copy of the object field (*shadow* object field) or array element (*shadow* array element) while leaving the original unchanged, and to replace—until the transaction is aborted—references to persistent fields and array elements by references to their shadow copies holding the new value. Note that if an object field to which no new value has been assigned is referenced (and for which therefore no shadow copy has been initialised), the original reference is used.

Making this work in practice requires altering the assignment rule for the cases where a transaction is in progress and is expected to abort (when the TRA tag is present). Also, the semantics of an update and the rules for update simplification need to be adjusted, as described later on. We now present the assignment rule for the $[\![\cdot]\!]$ modality with the TRA tag present. Since persistence is a property of object fields, this is a rule for assigning values to object fields, with the `null` check omitted for clarity.

$$\mathsf{assignTRAObject}\ \dfrac{\Longrightarrow \{o\texttt{.}a' := se'\}[\![\mathsf{TRA}\colon\!\pi\ \omega]\!]\phi}{\Longrightarrow [\![\mathsf{TRA}\colon\!\pi\ o\texttt{.}a\ \texttt{=}\ se\texttt{;}\ \omega]\!]\phi}$$

The shadowing is denoted with the prime symbol $'$. The corresponding rules for $\langle\cdot\rangle$ and $[\cdot]$ are the same.

In general, the rules of this class work by shadowing all appearing object fields and array elements, i.e., replacing all occurrences of `o.a` with `o.a'` and all occurrences of or `a[e]` with `a[e']'` in se when generating an update. For the array access function `[]` the prime symbol denotes a shadow access function that operates on copies of elements of a given array. The reference `obj` in `obj.a` as well as `a` in `a[i]` is not primed, since it is either a local variable, which is not persistent, or the `this` reference, which is not assignable, or a

static class reference, like `SomeClass`, which also should be viewed as not assignable. All occurrences of local variables are also left unchanged.

As mentioned, the semantics of an update has to be changed to take care of the cases when the shadow copy of an object's field has not been initialised. In the new semantics, if the value of `o.a`$'$ or `a[e]`$'$ is referred to in an update but is not known (that is, there was no such value assigned in the preceding updates) then it is considered to be equal to `o.a` or `a[e]`, respectively.

The assignments to the shadow copies are not visible outside the transaction, where the original values are used again—the effect of a roll-back is accomplished. Each separate transaction has to have its own set of shadow copies of fields or array elements, so that shadow copies from different transactions do not collide with each other. Thus, the second encountered transaction uses $''$, the third one $'''$, etc.

The rule assignTRAObject takes care of assignments to object fields, which are always persistent in JAVA CARD. For array elements however, the situation is a bit more complex—the programmer can also explicitly define an array to be transient, in which case the assignments to elements of such an array are executed unconditionally. For persistent arrays, as for object fields, assignments inside a transaction are conditional. Thus, the rule for handling assignments to array elements in the scope of the TRA tag has to account for the persistency type of the array. It is not possible to statically decide which arrays are transient and which are not, since they are defined to be transient by reference and not by name. This problem can be treated by adding an implicit field `<transient>` to each array ($\Rightarrow$ Sect. 3.6.6) indicating whether the given array is transient or persistent (rules for initialising arrays set this field). Compared to rule assignTRAObject, the conditional assignment rule for arrays has two premises to distinguish between persistent and transient arrays by checking the value of `<transient>` field:

assignTRAArray
$$\frac{a.\texttt{<transient>} \doteq \text{TRUE} \Longrightarrow \{\, a\,[e']\ :=\ se'\,\}[\![\text{TRA:}\pi\ \omega]\!]\phi \qquad a.\texttt{<transient>} \doteq \text{FALSE} \Longrightarrow \{\, a\,[e']'\ :=\ se'\,\}[\![\text{TRA:}\pi\ \omega]\!]\phi}{\Longrightarrow [\![\text{TRA:}\pi\ a\,[e]\ =\ se\,;\ \omega]\!]\phi}$$

The remaining rules for $[\![\text{TRA:}\cdot]\!]$ (that is, for other programming constructs) are the same as for $[\![\cdot]\!]$, and the remaining rules for $\langle\text{TRA:}\cdot\rangle$ and $[\text{TRA:}\cdot]$ are as if there were no transaction tag.

9.6 Examples

In the following, we show two examples of proofs using the above rules. The first example shows how the $[\![\cdot]\!]$ assignment and `while` rules are used, the second example shows the transaction rules in action. The formula we are trying to prove in the second example is deliberately not provable and shows

the importance of the transaction mechanism when it comes to *card tear* properties.

Corresponding proofs in the KeY system are performed almost automatically, the only place where user interaction is required is providing the loop invariant.

Example 9.1. Consider the following Java program p:

```java
x = 3;
while (x < 10) {
  if(x == 2)
    x = 1;
  else
    x++;
}
```

We show that throughout the execution of this program, the strong invariant $\phi \doteq x \geq 2$ holds, that is, we prove the formula $x \geq 2 \rightarrow [\![p]\!]x \geq 2$. Proof steps are numbered locally on the right side of formulae.

Proof. We start the proof with the sequent

$$x \geq 2 \Longrightarrow [\![x = 3; \ldots]\!]x \geq 2 \tag{P1}$$

Applying the assignment rule assignTout to (P1) produces two proof obligations:

$$x \geq 2 \Longrightarrow x \geq 2 \tag{P2}$$

$$x \geq 2 \Longrightarrow \{x := 3\}[\![\text{while} \ldots]\!]x \geq 2 \tag{P3}$$

Sequent (P2) is obviously valid. Applying the rule whileTout to (P3) with $x \geq 3$ as the loop invariant *Inv* gives us the four proof obligations below. Note that here it is necessary to use $x \geq 3$ as the invariant. Using $\phi \doteq x \geq 2$ would not be enough, because the assignment $x = 1$ inside the then-branch of the if statement could not be discarded and x would be assigned 1, which would break the $x \geq 2$ property.

$$x \geq 2 \Longrightarrow \{x := 3\}x \geq 3 \tag{P4}$$

$$x \geq 3, \ x < 10 \Longrightarrow [\beta]x \geq 3 \tag{P5}$$

$$x \geq 3, \ x < 10 \Longrightarrow [\![\beta]\!]x \geq 2 \tag{P6}$$

$$x \geq 3, \ x \geq 10 \Longrightarrow [\![\ ]\!]x \geq 2 \tag{P7}$$

where:

$$\beta \doteq \text{if}(x == 2) \ x = 1; \text{ else } x++;$$

Applying the update in (P4) results in $\text{x} >= 2 \Longrightarrow 3 >= 3$ which is valid, and simplifying (P7) results in $\text{x} >= 3,\ \text{x} >= 10 \Longrightarrow \text{x} >= 2$, which is also valid. We are left with (P5) and (P6) to prove. Applying the if rule to (P5) gives two proof obligations:

$$\text{x} >= 3,\ \text{x} < 10,\ \text{x} \doteq 2 \Longrightarrow [\text{x = 1;}]\text{x} >= 3 \qquad (\text{P8})$$

$$\text{x} >= 3,\ \text{x} < 10,\ \text{x} \,!\!\doteq 2 \Longrightarrow [\text{x = x + 1;}]\text{x} >= 3 \qquad (\text{P9})$$

(P8) is valid by contradiction in the antecedent. Applying the assignment rule to (P9) gives:

$$\text{x} >= 3,\ \text{x} < 10,\ \text{x} \,!\!\doteq 2 \Longrightarrow \{\text{x} := \text{x} + 1\}[\ ]\text{x} >= 3 \qquad (\text{P10})$$

which is reduced to:

$$\text{x} >= 3,\ \text{x} < 10,\ \text{x} \,!\!\doteq 2 \Longrightarrow \text{x} + 1 >= 3 \qquad (\text{P11})$$

Sequent (P11) is valid. We can continue with (P6) and apply the if rule yielding two proof obligations:

$$\text{x} >= 3,\ \text{x} < 10,\ \text{x} \doteq 2 \Longrightarrow [\![\text{x = 1;}]\!]\text{x} >= 2 \qquad (\text{P12})$$

$$\text{x} >= 3,\ \text{x} < 10,\ \text{x} \,!\!\doteq 2 \Longrightarrow [\![\text{x = x + 1;}]\!]\text{x} >= 2 \qquad (\text{P13})$$

Sequent (P12) is valid by contradiction in the antecedent. Applying rule assignTout to (P13) gives two proof obligations:

$$\text{x} >= 3,\ \text{x} < 10,\ \text{x} \,!\!\doteq 2 \Longrightarrow \text{x} >= 2 \qquad (\text{P14})$$

$$\text{x} >= 3,\ \text{x} < 10,\ \text{x} \,!\!\doteq 2 \Longrightarrow \{\text{x} := \text{x} + 1\}[\![\ ]\!]\text{x} >= 2 \qquad (\text{P15})$$

Sequent (P14) is valid. Sequent (P15) is reduced to:

$$\text{x} >= 3,\ \text{x} < 10,\ \text{x} \,!\!\doteq 2 \Longrightarrow \text{x} + 1 >= 2 \qquad (\text{P16})$$

Sequent (P16) is valid and thus we have proved the initial formula. Figure 9.1 shows the proof tree for this example.

Example 9.2. Now consider the following JAVA CARD program p (fields of o are persistent):

─── JAVA CARD ───────────────────────────────────
```
jvmBeginTransaction();
  o.x = 60;
  o.y = 40;
jvmCommitTransaction();
t = o.x;
o.x = o.y;
o.y = t;
```
──────────────────────────────── JAVA CARD ───

aT = assignTout
wT = whileTout
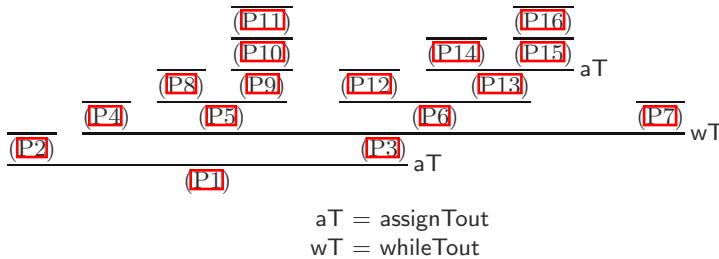
Fig. 9.1. The proof tree from Example 9.1

We try to prove that the strong invariant $\mathtt{o.x + o.y} \doteq 100$ holds throughout the execution of this program. The formula to prove is the following:

$$\mathtt{o.x + o.y} \doteq 100 \Longrightarrow [\![\mathtt{jvmBeginTransaction();...}]\!]\mathtt{o.x + o.y} \doteq 100 \quad \text{(P1)}$$

Note that this is not provable.

Proof. We start our proof by applying rule beginTransTout to (P1) yielding three proof obligations:

$$\mathtt{o.x + o.y} \doteq 100 \Longrightarrow \mathtt{o.x + o.y} \doteq 100 \quad\quad\quad\quad \text{(P2)}$$

$$\mathtt{o.x + o.y} \doteq 100 \Longrightarrow [\![\mathtt{TRC:o.x = 60;...}]\!]\mathtt{o.x + o.y} \doteq 100 \quad \text{(P3)}$$

$$\mathtt{o.x + o.y} \doteq 100 \Longrightarrow [\![\mathtt{TRA:o.x = 60;...}]\!]\mathtt{o.x + o.y} \doteq 100 \quad \text{(P4)}$$

Sequent (P2) is obviously valid. Applying rule assignTRAObject to (P4) gives:

$$\begin{array}{c} \mathtt{o.x + o.y} \doteq 100 \Longrightarrow \\ \{\mathtt{o.x'} := 60\} \\ [\![\mathtt{TRA:o.y = 40;...}]\!]\mathtt{o.x + o.y} \doteq 100 \end{array} \quad \text{(P5)}$$

Notice that since we are inside a transaction the assignment rule does not branch. Again, applying assignTRAObject to (P5) gives:

$$\begin{array}{c} \mathtt{o.x + o.y} \doteq 100 \Longrightarrow \\ \{\mathtt{o.x'} := 60, \mathtt{o.y'} := 40\} \\ [\![\mathtt{TRA:jvmCommitTransaction();...}]\!]\mathtt{o.x + o.y} \doteq 100 \end{array} \quad \text{(P6)}$$

Applying rule commitTransUnexp to (P6) proves (P6) to be valid. Applying rule assignTRC to (P3) gives:

$$\begin{array}{c} \mathtt{o.x + o.y} \doteq 100 \Longrightarrow \\ \{\mathtt{o.x} := 60\}[\![\mathtt{TRC:o.y = 40;...}]\!]\mathtt{o.x + o.y} \doteq 100 \end{array} \quad \text{(P7)}$$

Again, applying assignTRC to (P7) gives:

$$o.x + o.y \doteq 100 \Longrightarrow$$
$$\{o.x := 60, o.y := 40\} \tag{P8}$$
$$[\![\texttt{TRC:jvmCommitTransaction();} \ldots]\!] o.x + o.y \doteq 100$$

Applying rule commitTransExp to (P8) gives:

$$o.x + o.y \doteq 100 \Longrightarrow$$
$$\{o.x := 60, o.y := 40\} [\![\texttt{t = o.x;} \ldots]\!] o.x + o.y \doteq 100 \tag{P9}$$

Applying rule assignTout to (P9) gives two proof obligations:

$$o.x + o.y \doteq 100 \Longrightarrow \{o.x := 60, o.y := 40\} o.x + o.y \doteq 100 \tag{P10}$$

$$o.x + o.y \doteq 100 \Longrightarrow$$
$$\{o.x := 60, o.y := 40, t := o.x\} \tag{P11}$$
$$[\![\texttt{o.x = o.y;} \ldots]\!] o.x + o.y \doteq 100$$

Sequent (P10) is simplified to:

$$o.x + o.y \doteq 100 \Longrightarrow 60 + 40 \doteq 100 \tag{P12}$$

which is valid. Applying rule assignTout to (P11) gives two proof obligations:

$$o.x + o.y \doteq 100 \Longrightarrow$$
$$\{o.x := 60, o.y := 40, t := o.x\} o.x + o.y \doteq 100 \tag{P13}$$

$$o.x + o.y \doteq 100 \Longrightarrow$$
$$\{o.x := 60, o.y := 40, t := o.x, o.x := o.y\} \tag{P14}$$
$$[\![\texttt{o.y = t;} \ldots]\!] o.x + o.y \doteq 100$$

Sequent (P13) is reduced to:

$$o.x + o.y \doteq 100 \Longrightarrow 60 + 40 \doteq 100 \tag{P15}$$

which is valid. Applying rule assignTout to (P14) gives again two proof obligations:

$$o.x + o.y \doteq 100 \Longrightarrow$$
$$\{o.x := 60, o.y := 40, t := o.x, o.x := o.y\} o.x + o.y \doteq 100 \tag{P16}$$

$$o.x + o.y \doteq 100 \Longrightarrow$$
$$\{o.x := 60, o.y := 40, t := o.x, o.x := o.y, o.y := t\} \tag{P17}$$
$$[\![\ ]\!] o.x + o.y \doteq 100$$

Sequents (P16) and (P17) are reduced to, respectively:

$$o.x + o.y \doteq 100 \Longrightarrow 40 + 40 \doteq 100 \tag{P18}$$

$$o.x + o.y \doteq 100 \Longrightarrow 60 + 40 \doteq 100 \tag{P19}$$

Sequent (P19) is obviously valid. Sequent (P18) is not provable. Inspecting our program closely shows that indeed both o.x and o.y are equal to 40 at some point (after line 6 is executed) and their sum is 80, which violates the property we wanted to prove. Figure 9.2 shows the proof tree for this example with an open proof goal (P18).

aT = assignTout
bT = beginTransTout
cTE = commitTransExp
cTU = commitTransUnexp
aTRA = assignTRAObject
aTRC = assignTRC
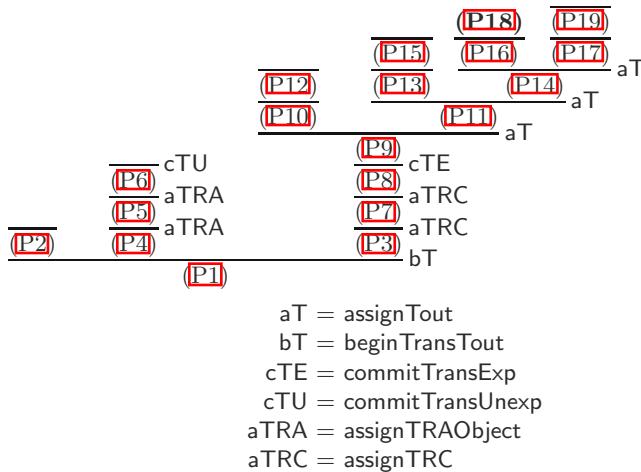
Fig. 9.2. The proof tree from Example 9.2

9.7 Non-atomic Java Card API Methods

There is one more aspect of Java Card transaction mechanism that we have to cover in our logic to make the support for strong invariants and transactions complete. There are two static methods in the Java Card API that exhibit an additional feature of the Java Card transaction mechanism. The two methods in question are `arrayCopyNonAtomic` and `arrayFillNonAtomic` from the `Util` class. Hubbers and Poll [2004b] thoroughly analyse the behaviour of the two methods based on extensive experiments performed with Java Card devices. Here we only present the highlights that motivate further extension to Java Card DL, for further details we refer the reader to [Hubbers and Poll, 2004b].

Methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` are responsible for copying, respectively, resetting to a given value, an array in a non-atomic fashion even when a transaction is in progress, that is, they *bypass* the transaction mechanism. The motivation behind the need for such methods is the following. Imagine a variable in your Java Card applet, which despite being persistent, needs to be updated unconditionally during a transaction. One example of such a variable would be the number of tries left to present a correct PIN code in the `OwnerPIN` class. Such a variable, call it `triesLeft`, has to be updated unconditionally during a transaction to prevent a security breach. If it would be updated conditionally, an aborted transaction would reset `triesLeft` to the value before the transaction was started, giving the prospective attacker an infinite number of tries to present the correct PIN code. Thus, the need for exclusion of certain persistent memory locations from the transaction mechanism. In the current version, Java Card only

allows such non-atomic updates for elements of `byte` arrays, that is, one cannot update an object attribute unconditionally inside a transaction, and hence there are only two API methods to take care of non-atomic updates, `arrayCopyNonAtomic` and `arrayFillNonAtomic`.[3]

The consequence for our logic is that, apart from committing or aborting, a transaction can also be suspended to perform unconditional updates to persistent `byte` array elements and later resumed to continue updating persistent data atomically.

On top of that, the two non-atomic methods introduce one more complication to the semantics of Java Card transaction mechanism. Extensive experiments with real Java Card devices presented by Hubbers and Poll [2004b] show that the notion of transaction roll-back is under-specified in the official Java Card specification [Sun, 2003d]. Consider the following two short (abbreviated) pieces of Java Card code. Persistent array `a` stores elements of type `byte` and the last argument of `arrayFillNonAtomic` is the value that is assigned to all elements of an array (in this case only one element, `a[0]`):

```
——— Java Card ———————            ——— Java Card ———————
a[0] = 0;                        a[0] = 0;
beginTransaction();              beginTransaction();
 a[0] = 1;                        arrayFillNonAtomic(a,0,1,2);
 arrayFillNonAtomic(a,0,1,2);     a[0] = 1;
abortTransaction();             abortTransaction();
——————————— Java Card ———        ——————————— Java Card ———
```

The question is what is the value of `a[0]` at the end of each of these programs. If we turn to official Java Card specification, it says that upon transaction abort JCRE will restore to their previous values all the memory locations conditionally updated since the previous call to `beginTransaction`. The problem is how to interpret *their previous values*. Naturally one would assume it means *values they had just before the transaction was started*. However, experiments show that this is not the case—when the first program above is executed the value of `a[0]` is 0, the execution of the second program results in `a[0]` equal to 2. So, looking at the result of the second program, we can see that the value of `a[0]` is not rolled back to the value it had before the transaction was started, but to the value it was updated to by a non-atomic method *after* the transaction started and before the conditional assignment `a[0] = 1` took place. Following Hubbers and Poll [2004b] this suggests that

---

[3] Another motivation, which is not relevant here, for introducing such methods is efficiency. Since they are non atomic they can be implemented more efficiently in hardware and used to update arrays which do not need to preserve atomicity related properties.

the official Java Card specification is ambiguous and should be clarified as follows:

> "If power is lost (tear) or the card is reset or some other system failure occurs while a transaction is in progress, then the JCRE shall restore to the values they had directly prior to the first conditional update after the previous call to `beginTransaction` all fields and array components conditionally updated since the previous call to `beginTransaction`."

This definition of a transaction roll-back will require us to slightly modify the conditional assignment rule for arrays assignTRAArray to account for possible non-atomic method calls or, more generally, transaction suspension.

Before we continue with explaining how all these issues are handled in the logic, we quote one more sentence from the JCRE specification [Sun, 2003d]:

> "**Note** – The contents of an array component which is updated using the `arrayCopyNonAtomic` method or the `arrayFillNonAtomic` method while a transaction is in progress, is not predictable, following the abortion of the transaction. *[explicit, by the programmer, or implicit, by the JCRE]*"

What follows is that in practice any formal, deterministic reasoning about non-atomic method calls in Java Card is not possible. It also renders the two non-atomic methods practically useless and, more importantly, dangerous.[4] Experiments on real Java Card devices show that such indeterministic behaviour is indeed exhibited by some devices, but they also show there is a handful of Java Card devices that behave deterministically, that is, the assignments performed by non-atomic methods to persistent locations are kept after the transaction is aborted. Hence we need to have support in our logic for such well behaved devices. Still, it is not advisable to make calls to `arrayCopyNonAtomic` and `arrayFillNonAtomic` methods from within a transaction at all.

9.7.1 Transaction Suspending and Resuming

Similarly to what we did for the basic transaction support, we need to make Java Card DL aware of possible transaction suspension in a program. Thus, we introduce two more distinguished methods to Java Card that are going to be treated by the logic rules:

- `jvmSuspendTransaction`: suspend an active transaction,
- `jvmResumeTransaction`: resume a suspended transaction.

---

[4] When a Java Card device indeed exhibits such an indeterministic behaviour suggested by the JCRE documentation, very simple *card tear* attacks are possible.

Similarly as for `jvmBeginTransaction`, etc., these statements are only used in the calculus, they are not part of the official JAVA CARD API. Our JAVA CARD model used in the logic can then assume the following implementation for `arrayFillNonAtomic`:[5]

```
—— JAVA CARD ——
public class Util {
  public static short arrayFillNonAtomic(
     byte[] dest, short offset, short len, byte val)
  {
    de.uka.ilkd.key.javacard.KeYJCSystem.
      jvmSuspendTransaction();
    for(short i = 0; i < len; i++)
      dest[(short)(offset + i)] = val;
    de.uka.ilkd.key.javacard.KeYJCSystem.
      jvmResumeTransaction();
    return (short)(offset + len);
  }
}
                                              —— JAVA CARD ——
```

And similarly for `arrayCopyNonAtomic`.

We also need to mark the modality, respectively, the program it contains, with a tag that says that the current transaction is suspended. The tag we are going to use is TRS. Similarly as for the TRA and TRC tags this indicates that different rules should be applied in the scope of a suspended transaction, in particular for the array element assignment.

Rules for Suspending a Transaction

It is important to note that, given the way we reason about transactions in our logic, the transaction suspension is only relevant when modalities with an active TRA tag are considered. When there is no transaction in progress the transaction suspension can be safely ignored. In the scope of the commit branch, transaction suspension also does not introduce any additional semantics—all assignments are going to be committed, no matter if triggered by a non-atomic method or any other program statement. This gives raise to the following `jvmSuspendTransaction` rules:

$$\text{suspendTrans} \ \frac{\implies \langle \pi \ \omega \rangle \phi}{\implies \langle \pi \ \texttt{jvmSuspendTransaction();} \ \omega \rangle \phi}$$

---

[5] The actual implementation would be more complex—it should take care of some extra checks that the actual JAVA CARD API method would do, here we used a simplified version for clarity.

$$\text{suspendTransTRC} \quad \frac{\Longrightarrow \langle \mathsf{TRC}{:}\pi\ \omega\rangle\phi}{\Longrightarrow \langle \mathsf{TRC}{:}\pi\ \texttt{jvmSuspendTransaction();}\ \omega\rangle\phi}$$

And similarly for box and throughout modalities. In the above two rules, statements related to transaction suspension are simply ignored. When the modality is marked with the TRA tag however, we need to mark the modality with the TRS tag to indicate that a transaction is momentarily suspended:

$$\text{suspendTransTRA} \quad \frac{\Longrightarrow \langle \mathsf{TRS}{:}\pi\ \omega\rangle\phi}{\Longrightarrow \langle \mathsf{TRA}{:}\pi\ \texttt{jvmSuspendTransaction();}\ \omega\rangle\phi}$$

Again, the rules for box and throughout modalities are the same. Rules for resuming a transaction will simply change the TRS tag back to the TRA tag.

Rules for Resuming a Transaction

As we pointed out in the previous section, transaction suspension can be ignored for modalities without any transaction tags or for modalities marked with the TRC tag. Thus, for these cases, the rules for resuming the transaction will also ignore the transaction resume statement jvmResumeTransaction (again, the rules for box and throughout are the same):

$$\text{resumeTrans} \quad \frac{\Longrightarrow \langle \pi\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{jvmResumeTransaction();}\ \omega\rangle\phi}$$

$$\text{resumeTransTRC} \quad \frac{\Longrightarrow \langle \mathsf{TRC}{:}\pi\ \omega\rangle\phi}{\Longrightarrow \langle \mathsf{TRC}{:}\pi\ \texttt{jvmResumeTransaction();}\ \omega\rangle\phi}$$

And for the modalities with TRS tag we simply change the tag back to TRA to indicate that a transaction is again in progress:

$$\text{resumeTransTRA} \quad \frac{\Longrightarrow \langle \mathsf{TRA}{:}\pi\ \omega\rangle\phi}{\Longrightarrow \langle \mathsf{TRS}{:}\pi\ \texttt{jvmResumeTransaction();}\ \omega\rangle\phi}$$

9.7.2 Conditional Assignments Revised

The new semantics of transaction roll-back affects the conditional assignment rule for arrays assignTRAArray in the following way. The value of an array element is rolled back to the state it had just before the first conditional assignment took place. This means that the assignment rules need to keep track and need to be aware if an array location has been conditionally updated or not. For this we need to introduce another implicit field to arrays on the logic level, like the `<transient>` field indicating the object's persistency type. The new field is called `<traInitialized>` and is of type boolean array. Having a field like this allows us to put formulas like `a.<traInitialized>[i]`

into the sequent which will evaluate to true or false depending if the array element `a[i]` has been conditionally updated (initialised) or not. So, first our conditional assignment rule needs to update information about conditional initialisation when a conditional assignment happens (the default value for `a.<traInitialized>[i]` is always false):

assignTRAArrayRevised
$$a.\text{<transient>} \doteq \text{TRUE} \Longrightarrow \{a[e'] := se'\}\langle\text{TRA}{:}\pi\ \omega\rangle\phi$$

$$a.\text{<transient>} \doteq \text{FALSE} \Longrightarrow$$
$$\{a.\text{<traInitialized>}[e']' := \text{TRUE}\}\{a[e']' := se'\}\langle\text{TRA}{:}\pi\ \omega\rangle\phi$$
$$\overline{\qquad \Longrightarrow \langle\text{TRA}{:}\pi\ a[e]\ =\ se;\ \omega\rangle\phi}$$

As before, the rule for box and throughout are the same. Note that we only need to keep track of conditional assignments for persistent arrays, because transient arrays are always updated unconditionally.

The second part of treating the new semantics of transaction roll-back is to make the rules that update an array element when a transaction is suspended check if an array element has been already updated conditionally, and depending on that do a conditional or unconditional update. This way, if an array element has not yet been initialised inside a transaction the update is going to be unconditional, so that when the transaction is aborted, the array element will have this unconditionally assigned value. So we need a rule for array assignment for suspended transactions. Here only the rules for diamond and box are the same, because for throughout extra checks need to be done in the scope of the TRA tag, which we will explain shortly:

assignDiaSuspArray
$$a.\text{<traInitialized>}[e']' \doteq \text{FALSE} \mid a.\text{<transient>} \doteq \text{TRUE} \Longrightarrow$$
$$\{a[e'] := se'\}\langle\text{TRS}{:}\pi\ \omega\rangle\phi$$

$$a.\text{<traInitialized>}[e']' \doteq \text{TRUE}) \Longrightarrow \{a[e']' := se'\}\langle\text{TRS}{:}\pi\ \omega\rangle\phi$$
$$\overline{\qquad \Longrightarrow \langle\text{TRS}{:}\pi\ a[e]\ =\ se;\ \omega\rangle\phi}$$

Finally, we need to properly handle assignments for a suspended transaction in the scope of the throughout modality. Since the transaction is suspended and some of the assignments are executed unconditionally, it means we need to check the strong invariant after these unconditional assignments. Thus, the rule for a suspended transaction for the throughout modality:

assignToutSuspArray
$$arr.\text{<traInitialized>}[e']' \doteq \text{FALSE}\ \&$$
$$arr.\text{<transient>} \doteq \text{FALSE} \Longrightarrow \{a[e'] := se'\}(\phi\ \&\ [\![\text{TRS}{:}\pi\ \omega]\!]\phi)$$

$$a.\text{<traInitialized>}[e']' \doteq \text{TRUE} \Longrightarrow \{a[e']' := se'\}[\![\text{TRS}{:}\pi\ \omega]\!]\phi$$

$$a.\text{<traInitialized>}[e']' \doteq \text{FALSE}\ \&\ a.\text{<transient>} \doteq \text{TRUE} \Longrightarrow$$
$$\{a[e'] := se'\}[\![\text{TRS}{:}\pi\ \omega]\!]\phi$$
$$\overline{\qquad \Longrightarrow [\![\text{TRS}{:}\pi\ a[e]\ =\ se;\ \omega]\!]\phi}$$

In this rule, in the first premiss, the formula ϕ is also evaluated right after the unconditional assignment to the array element happens. In the second premiss (an element is already conditionally assigned and hence is going to be rolled back after the abort) and in the third premiss (the array element is transient and is going to be reset to a default value upon unexpected termination) the extra check is not necessary.

9.8 Summary

In this chapter we presented numerous extensions to Java Card DL necessary to support strong invariants and Java Card's transaction mechanism. Strong invariants are crucial to express *card tear* properties for Java Card applets—properties that have be maintained in case of an unexpected (premature) termination of the program. The support for the transaction mechanism is necessary to reason properly about (i) different atomicity levels introduced by transactions in the context of strong invariants, and (ii) assignment roll-back caused by an aborted transaction in the context of all the modal operators. We also presented some "on paper" examples showing how the extended calculus works in practice. In the closing Section 9.9 we briefly discuss some of the implementation issues and point to relevant transaction examples included in the KeY distribution.

9.8.1 Related Work

To the best of our knowledge, the KeY system is the first Java Card verification tool that treats strong invariants and transactions thoroughly, including the implementation of the logic rules [Beckert and Mostowski, 2003, Mostowski, 2006]. The only other work that gives a basic formal framework to reason about card tears and Java Card transaction mechanism is presented by Hubbers and Poll [2004a], however the framework has not been implemented. Hubbers and Poll also wrote a technical report [Hubbers and Poll, 2004b], which we already cited on many occasions, that describes numerous experiments with real Java Card devices with respect to transactions, Java Card non-atomic methods and card tears and tries to set in order the intricacies of the transaction mechanism. The report was of enormous help to us when developing the support for transaction suspending.

Recently, the Krakatoa tool[6] has been extended to support Java Card transaction mechanism [Marché and Rousset, 2006]. The extension treats the transaction mechanism thoroughly and provides a JML interface to transactions related properties (like strong invariants).

The idea to introduce trace modalities "throughout" and "at least once" to pure Dynamic Logic was first presented by Beckert and Schlager [2001]. In our work we adapt these ideas to the more complex Java Card DL setting,

---

[6] http://krakatoa.lri.fr

where we also have to deal with the transaction mechanism, which obviously is not present in pure Dynamic Logic. We have also chosen not to introduce the "at least once" modality to JAVA CARD DL because so far we did not find practical applications for it. But given the KeY system's flexible logic framework it should be quite straightforward task and future versions of the KeY system may contain the support for "at least once" modality.

9.9 Implementation of the Rules

Here we briefly discuss some taclet implementation issues related to throughout and transaction rules.

9.9.1 New Modalities

The transaction markers `TRC`, `TRA`, and `TRS` are implemented as separate modalities. That is, instead of marking a modality, say diamond, with a `TRC` tag, a new modality "diamond with `TRC`" is introduced. Thus, for the diamond operator diamond there are four different modalities:

- `\<...\>`: the diamond modality,
- `\diamond_trc...\endmodality`: the diamond modality marked with the `TRC` tag,
- `\diamond_tra...\endmodality`: the diamond modality marked with the `TRA` tag,
- `\diamond_susp...\endmodality`: the diamond modality marked with the `TRS` tag.

Similarly for box and throughout operators, where shorthand notations are `\[...\]` and `\[[...\]]`, respectively. This way the transaction specific rules, notably all different assignment rules, can be written separately for the different modalities, or groups of modalities with the help of schematic modal operators described in Section 4.2.

9.9.2 Transaction Statements and Special Methods

In the theoretical part of this chapter we mentioned distinguished API method calls that the logic rules operate on: `jvmBeginTransaction`, `jvmCommitTransaction`, etc. Thus, the model of the JAVA CARD API that is associated with a proof involving transactions has to include the following class and method declarations:

─── JAVA CARD ───

```
package de.uka.ilkd.key.javacard;

public class KeYJCSystem {
  public static native void jvmBeginTransaction();
```

```
  public static native void jvmCommitTransaction();
  public static native void jvmAbortTransaction();
  public static native void jvmSuspendTransaction();
  public static native void jvmResumeTransaction();
}
```
── JAVA CARD ──

And as before, the implementation of the actual JAVA CARD transaction methods in our model API has to take the following form:

── JAVA CARD ──
```
public class JCSystem {
  private static short _transactionDepth = 0;

  public static void beginTransaction()
                         throws TransactionException {
    if(_transactionDepth > 0)
      TransactionException.throwIt(
        TransactionException.IN_PROGRESS);
    _transactionDepth++;
    de.uka.ilkd.key.javacard.KeYJCSystem.
      jvmBeginTransaction();
  }
  ...
}
```
── JAVA CARD ──

As there is no public user interface to support transaction suspending there are no public API methods suspendTransaction and resumeTransaction. The only two methods that can suspend a transaction are arrayCopyNonAtomic and arrayFillNonAtomic. In the model API these two methods can call jvmSuspendTransaction and jvmResumeTransaction directly, following the schema discussed in Section 9.7.1.

An alternative and more efficient solution to handle non-atomic method calls is to introduce distinguished (or special) (⇒ Sect. 14.4.1) methods for array filling and copying as well, following this schema:

── JAVA CARD ──
```
public class Util {
  public static short arrayFillNonAtomic(
      byte[] bArray, short  bOff, short  bLen, byte   bValue) {
    if(bArray == null) throw new NullPointerException();
    if(bLen < 0 || bOff < 0 || bOff + bLen > bArray.length)
      throw new ArrayIndexOutOfBoundsException();
    de.uka.ilkd.key.javacard.KeYJCSystem.
      jvmArrayFillNonAtomic(bArray, bOff, bLen, bValue);
```

```
    return (short)(bOff+bLen);
  }
}


public class KeYJCSystem {
  public static native void jvmArrayFillNonAtomic(
    byte[] bArray, short bOff, short bLen, byte  bValue);
}
```
—— JAVA CARD ————

This way, it is ensured that the distinguished method `jvmArrayFillNon-`
`Atomic` is called with well defined arguments and should always succeed
without throwing exceptions. A specialised taclet takes care of executing
`jvmArrayFillNonAtomic`, which boils down to creating a suitable quantified
update ($\Rightarrow$ Sect. 3.2.3) that follows the semantics of transaction suspending
accurately.

Other special methods that the KeY JAVA CARD model utilises are `jvm-`
`IsTransient` (low-level interface to `isTransient` method of the JAVA CARD
API) and `jvmMakeShort`/`jvmSetShort` (low-level methods related to repre-
sentation of short values with bytes). The KeY system distribution contains
a skeleton JAVA CARD API implementation which uses such distinguished
methods (in `examples/java_dl/jc_transactions/code`).

The distinguished `jvm...` methods are excluded from the method call
handling rules, so that specialised taclets can be defined for them. Since
concrete method names are not allowed in the rules, schema variables that
match distinguished `jvm...` calls have to be defined. Then, for example, we
can write a taclet for the rule beginTransTout this way:

———— Taclet ——
```
beginTransTout {
    \schemaVar \program jvmBeginTransaction #trb;
    \find (==> \throughout{..
       de.uka.ilkd.key.javacard.KeYJCSystem()::
          de.uka.ilkd.key.javacard.KeYJCSystem.#trb();
      ...}\endmodality post)
  "Pre-State":
    \replacewith(==> post);
  "Will␣Abort":
    \replacewith(==> \throughout_tra{.. ...}\endmodality post);
  "Will␣Commit":
    \replacewith(==> \throughout_trc{.. ...}\endmodality post))
    \heuristics(simplify_prog)
    \displayname "beginTrans"
};
```
—— Taclet ————

All the other rules for handling transaction entering, exiting, suspending, and resuming are similar.

9.9.3 Taclet Options

Transactions related rules are only active in the KeY system, when the taclet option ($\Rightarrow$ Sect. 4.4.2) `transactions` is set to `transactionsOn`. Similarly, for the throughout rules there is a taclet option `throughout`, which can be activated by setting it to `toutOn`. Also, in practice it often happens that the verified program with transactions does not contain any `abortTransaction` calls, or possibly they are never reached. In this case the abort branch of the proof resulting from applying the transaction begin rule beginTrans will always be closed by the axiom rule commitTransUnexp. So introducing the abort branch case into the proof is superfluous in the first place and skipping this branch from the start can considerably reduce the size of the proof. If the user knows or suspects that there is no call to `abortTransaction`[7] in the program that he wants to verify, he can set the taclet option `transactionAbort` to `abortOff`, which will result in choosing the set of rules that do not introduce the abort branch into the proof. In case the `abortTransaction` is reached in the program anyway, the commit branch will not be closable, because the axiom rule abortTransUnexp is not active when the `abortOff` option is set.

9.9.4 Implicit Fields

In our conditional assignment rules for arrays ($\Rightarrow$ Sect. 9.5.2, 9.7.2) we also made use of two implicit fields ($\Rightarrow$ Sect. 3.6.6):

- `<transient>` of boolean type that indicates the persistency type of an array,
- `<traInitialized>` of boolean array type that indicates whether a given array element has been conditionally updated inside a transaction.

In the KeY system's implementation the persistency field `<transient>` is of integer type instead of boolean, because in reality each object can have two different types of non-persistency, for example, a value of a transient array can be kept only as long as the applet is active, or as long as the card is powered [Sun, 2003d]. The two taclet meta-constructs ($\Rightarrow$ Sect. 4.2.8) that are used in taclets to get access to these implicit fields are:

- `#transient(#a)` to express `#a.<transient>`,
- `#traInitialized(#a)[#i]` to express `#a.<traInitialized>[#i]`.

---

[7] Keep in mind that in Java Card programs there can also be implicit calls to `abortTransaction` (e.g., when a program starts a transaction and does not close it before the end of execution) which cannot be deduced by looking at the code.

9.9.5 Conditional Assignment Rule Taclets

This is almost everything that is needed to write the taclets for conditional assignment rules ($\Rightarrow$ Sect. 9.5.2, 9.7.2). Recall that, in Section 9.5.2, we said that each subsequent transaction needs to keep its own copies of the attributes, respectively, array elements by using a different number of prime ($'$) symbols, so that conditional assignments from one transaction do not collide with conditional assignments from another transaction. Thus, it is not correct to write expressions containing prime symbols directly in taclets, for example:

```
── Taclet ──────────────────────────────────────────────

  \replacewith({#o.#a' := #se}
                  \diamond_tra{...}\endmodality post)
──────────────────────────────────────────── Taclet ──
```

Instead the rule should assign the number of primes dynamically depending on the number of transactions processed in the proof so far. On top of that an attribute or array expression may be hidden behind a schema variable, for example, expression `#se` can possibly represent `a.o` or `ar[i]`. So, for taclets we need a meta-construct that will take care of both assigning the right number of primes (a shadow number) and for constructing such primed (shadowed) expressions from the attribute or array expressions hidden in schema variables on the fly when the rule is applied. The meta-construct is called `#shadowed`.

Now we can finally present (a simplified version of) the taclet that implements the conditional assignment rule assignTRAArray:

```
── Taclet ──────────────────────────────────────────────
assignTRAArray {
  \schemaVar
    \modalOperator {diamond_tra, box_tra, throughout_tra}
      #traonly;
  \find (\modality{#traonly}{..
           #v[#se]=#se0;...
         }\endmodality post) \sameUpdateLevel
  "array␣#v␣persistent":
    \replacewith (
       {#shadowed(#v[#se]) := #shadowed(#se0)}
         \modality{#traonly}{.. ...}\endmodality post
    )
    \add (#transient(#shadowed(#v)) = 0 ==> );
  "array␣#v␣transient":
    \replacewith (
       {#v[#se] := #shadowed(#se0)}
```

```
        \modality{#traonly}{.. ...}\endmodality post
   )
   \add (#transient(#shadowed(#v)) > 0 ==>);
  \heuristics(simplify_prog, simplify_prog_subset)
  \displayname "assignment"
};
```
———————————————————————————————— Taclet ——

This simplified version of the taclet does not cover `null` pointer or array index bounds checks. The taclet exactly reflects rule assignTRAArray, and also covers the same rule for the diamond and box operators by the use of `#traonly` schematic modal operator.

Accessing the Transaction Counter Directly

In some cases it is necessary to "prime" an expression selectively in the taclet, that is, to apply the `#shadowed` meta-operator only to specific subexpressions. To achieve that it is possible to refer to the current transaction number (shadow number) with the `#transactionCounter` schema variable this way:

—— Taclet ——————————————————————————————
```
  ...
  \replacewith({#o.#a^(#transactionCounter) := #se}
                \diamond_tra{...}\endmodality post)
  ...
```
———————————————————————————————— Taclet ——

If, for example, the current transaction counter is 3, `#o` matches `this`, `#a` matches `attr`, and `#se` matches 1, a formula with following update is going to be introduced into the sequent:

—— KeY ——————————————————————————————
```
  {this.attr''' := 1}...
```
———————————————————————————————— KeY ——

When an interaction is required from the user (for example, when taclet instantiation is required) that involves primed expressions, it is possible to use prime symbols directly.

9.9.6 Examples in the KeY System

The `examples/java_dl/jc_transactions` directory of the KeY system distribution contains simple examples of the KeY problem files involving transactions. For example, the file `ex1.key` contains the following problem:

```
─── KeY ──────────────────────────────────────────────
\javaSource "code/";

\programVariables { MyClass self; int a; }

\problem {
    javacard.framework.JCSystem._transactionDepth = 0
  & !self = null & self.a = 0 & a = 0
->
  \<{
    javacard.framework.JCSystem.beginTransaction();
      self.a = 10;
      a = 10;
    javacard.framework.JCSystem.abortTransaction();
  }\> (self.a = 0 & a = 10)
}
──────────────────────────────────────────── KeY ───
```

This example is automatically provable by choosing the basic JAVA CARD DL strategy (the `transactionAbort` taclet option needs to be active). It presents how the effect of a transaction roll-back is achieved and how conditional assignment to persistent and non-persistent memory locations are properly handled—the value of `self.a` is rolled back to the value it had before the transaction was started, while local variable `a` keeps the value that it was assigned during the transaction.

9.9.7 Current Limitations

One small part of JAVA CARD transaction semantics is not yet supported in the KeY system. According to the JAVA CARD specification [Sun, 2003d], references to all the objects created during a transaction are reset to a `null` reference upon transaction abort. In our execution model such references are preserved. That implies that proofs for programs that create objects inside a transaction may be (but do not necessarily have to be) unsound. However, creating objects within a transaction is not advised by JAVA CARD coding guidelines, thus for the moment we do not consider this limitation a big issue. Future versions for of the KeY system will resolve this problem.

Part III

Using the KeY System

10

Using KeY

by
Wolfgang Ahrendt

10.1 Introduction

This whole book is about the KeY *approach* and *framework*. This chapter
now focuses on the KeY *system*, and that entirely from the user's perspective.
Naturally, the graphical user interface (GUI) will play an important role here.
However, the chapter is not all about that. Via the GUI, the system and
the user communicate, and interactively manipulate, several artefacts of the
framework, like formulae of the used logic, proofs within the used calculus,
elements of the used specification languages, among others. Therefore, these
artefacts are (in parts) very important when using the system. Even if all
of them have their own chapter/section in this book, they will appear here
as well, in a somewhat superficial manner, with pointers given to in-depth
discussions in other parts.

We aim at a largely self-contained presentation, allowing the reader to
follow the chapter, and to *start* using the KeY system, without necessarily
having to read several other chapters of the book before. The reader, however,
can gain a better understanding by following the references we give to other
parts of the book. In any case, we strongly recommend to read Chapter 1
beforehand, where the reader can get a picture of what KeY is all about. The
other chapters are *not* treated as prerequisites to this one, which of course
imposes limitations on how far we can go here. Had we built on the knowledge
and understanding provided by the other chapters, we would be able to guide
the user much further into to the application of KeY to larger resp. more
difficult scenarios. However, this would raise the threshold for getting started
with the system, thereby contradicting the philosophy of the whole project.
The KeY framework was designed from the beginning for being usable *without*
having to read a thick book first. Software verification is a difficult task
anyhow. Neither the system nor the used artefacts (like the logic) should
add to that difficulty, and are designed to instead lower the threshold for the
user. The used logic, *dynamic logic* (DL), features transparency w.r.t. the

programs to be verified, such that the code literally appears in the formulae, allowing the user to relate back to the program when proving properties about it. The "taclet" language for the declarative implementation of both, rules and lemmas, is kept so simple that we can well use a rule's declaration as a tooltip when the user is about to select the rule. The calculus itself is, however, complicated, as it captures the complicated semantics of JAVA. Still, most of these complications do not concern the user, as they are handled in a fully automatic way. Powerful strategies relieve the user from tedious, time consuming tasks, particularly when performing *symbolic execution.*

In spite of a high degree of automation, in many cases there are significant, non-trivial tasks left for the user. It is the very purpose of the GUI to support those tasks well. When proving a property which is too involved to be handled fully automatically, certain steps need to be performed in an interactive manner, in dialogue with the system. This is the case when either the automated strategies are exhausted, or else when the user deliberately performs a strategic step (like a case distinction) manually, *before* automated strategies are invoked (again). In the case of human-guided proof steps, the user is asked to solve tasks like: *selecting a proof rule* to be applied, *providing instantiations for* the proof rule's *schema variables*, or *providing instantiations for quantified variables* of the logic. In turn, the system, and its advanced GUI, are designed to support these steps well. For instance, the selection of the right rule, out of over 1500(!), is greatly simplified by allowing the user to highlight any syntactical sub-entity of the proof goal simply by positioning the mouse. A dynamic context menu will offer only the few proof rules which apply to this entity. Furthermore, these menus feature tooltips for each rule pointed to. When it comes to interactive variable instantiation, *drag-and-drop* mechanisms greatly simplify the usage of the instantiation dialogues, and in some cases even allow to omit explicit rule selection. Other supported forms of interaction in the context of proof construction are the inspection of proof trees, the pruning of proof branches, stepwise backtracking, and the triggering of proof reuse.

Performing interactive proof steps is, however, only one of the many functionalities offered by the KeY system. Also, these features play their role relatively late in the process of verifying programs. Other functionalities are (we go backwards in the verification process): controlling the automated strategies, adding lemmas and generating corresponding proof obligations, customising the calculus (for instance by choosing either of the mathematical or the JAVA semantics for integers), and generating proof obligations from specifications. Those features (and several others to be discussed below) comprise what we call the "core KeY system", "stand-alone KeY system", or "stand-alone KeY prover".

On top of the core system, there exist integrations into (currently two) standard tools for (JAVA) software development, as was discussed in the introduction to this book (Chap. 1, see particularly Fig. 1.1). One of them is

the commercial CASE[1] tool Borland Together Control Center, the other is the open source IDE Eclipse. In both cases, users can develop the whole software project, comprising both specifications and implementations, entirely in the frame of either of these (KeY-enhanced) tools, which offer the *extended functionality* of generating proof obligations from selected entities of specifications, and starting up the KeY prover accordingly.

Working with the KeY system has therefore many aspects, and there are many ways to give an introduction into those. In this chapter, we will take an "inside out" approach, starting with the *core* prover, describing *how* it communicates *which artefacts* for *which purpose* with the user, when proving a formula at hand.

In general, we will discuss the usage of the system by means of rather (in some cases extremely) simple examples. Thereby, we try to provide a good understanding of the various ingredients before their combination (seemingly) complicates things. Also, the usage of the prover will sometimes be illustrated by at first performing basic steps manually, and demonstrating automation thereafter. Please note that the toy examples used all over this chapter serve the purpose of step by step introducing the concepts and usage of the KeY system. They are not suitable for giving any indication of the capabilities of the system. (See Part IV instead.)

Before we start, there is one more basic issue which should be reflected on at this point. The evolution of both, the KeY *project* in general, and the KeY *system* in particular, has been very dynamic up to now, and will continue to be so. As far as the *system* and its GUI is concerned, it has been constantly improved and will be modified in the future as well. The author faces the difficult task of not letting the description of the tool's usage depend too much on its current appearance. The grouping of menus, the visual placement of panes and tabs, the naming of operations or options, all that can potentially change. Also, on the more conceptual level, things like the configuration policy for strategies and rule sets, among others, cannot be assumed to be frozen for all times. Even the theoretical grounds will develop further, as KeY is indeed a *research project*. A lot of ongoing research does not yet show in the current release of the KeY system, like support for mainstream languages other than JAVA, support for *disproving* wrong formulae, or the combination of deductive verification with static analysis, to name just very few. These, and others, will enhance the framework, and find their way into the system. We make a strong effort, not only in this chapter, to make the material valuable for the understanding and usage also of the future KeY.

The problem of describing a dynamic system is approached from three sides. First, we will continue to keep available the book release of the system, KeY 1.0, on the KeY book's web page. Second, in order to not restrict the reader to that release only, we will try to minimise the dependency of the material on the current version of the system and its GUI. Third, whenever

---

[1] CASE stands for Computer-Aided Software Engineering.

we talk about the specific location of a pane, tab, or menu item, or about key/mouse combinations, we stress the dynamic nature of such information *in this way*. For instance, we might say that "one can trigger the run of an automated strategy which is restricted to a highlighted term/formula *by Shift + click on it*". Menu navigation will be displayed by connecting the cascaded menus/sub-menus with "→", like "**Options → Decision Procedure Config → Simplify**". Note that menu navigation is release dependent as well.

This chapter is meant for being read with the KeY system up and running. We want to explore the system *together* with the reader, and reflect on whatever shows up along the path. Downloads of KeY, particularly its book version, KeY 1.0, are available on the project page, www.key-project.org. The example input files, which the reader frequently is asked to load, can be found on the web page for this book, www.key-project.org/thebook, as well as in your KeY system's installation, under examples/BookExamples.

10.2 Exploring Framework and System Simultaneously

Together with the reader, we want to open, for the first time, the KeY system, in order to perform first steps and understand the basic structure of the interface. We start the stand-alone KeY prover *by running* bin/runProver *in your KeY installation directory*. The **KeY–Prover** main window, together with a **Proof Assistant**[2] pops up. The latter is simply a message window, which comments on pre-selected menus or actions the user is about to make.

Like many window-based GUIs, the main window offers several menus, a toolbar, and a few panes, partly tabbed. Instead of enumerating those components one after another, we immediately load an example to demonstrate some basic interaction with the prover.

10.2.1 Exploring Basic Notions and Usage: Building a Propositional Proof

In general, the KeY prover is made for proving formulae in *dynamic logic* (DL), an extension of *first-order logic*, which in turn is an extension of *propositional logic*. We start with a very simple propositional formula, when introducing the usage of the KeY prover, because a lot of KeY concepts can already be discussed when proving the most simple theorem.

Loading the First Problem

The formula we prove first is contained in the file andCommutes.key. In general, .key is the suffix for what we call *problem files*, which may, among other things, contain a formula to be proved. (The general format of .key files is documented in Appendix B.) For now, we look into the file andCommutes.key itself (using your favourite editor):

---

[2] If the **Proof Assistant** does not appear, please check **Options → Proof Assistant**.

─── KeY Problem File ───────────────────────────────

```
\predicates {
        p;
        q;
}
\problem {
        (p & q) -> (q & p)
}
```

─────────────────────────────── KeY Problem File ───

The `\problem` block contains the formula to be proved (with `&` and `->` denoting the logical "and" and "implication", respectively). In general, all functions, predicates, and variables appearing in a problem formula are to be declared beforehand, which, in our case here, is done in the `\predicates` block. We load this file *by* **File** → **Load ...** *(or selecting* 📁 *in the tool bar) and navigating through the opened file browser*. The system not only loads the selected `.key` file, but also the whole calculus, i.e., its rules.

Reading the Initial Sequent

Afterwards, we find the text `==> p & q -> q & p` displayed in the **Current Goal** pane. This seems to be merely the `\problem` formula, but actually, the arrow "`==>`" turns it into a *sequent*. KeY uses a sequent calculus, meaning that sequents are the basic artefact on which the calculus operates. Sequents have the form $\phi_1, \ldots, \phi_n \implies \phi'_1, \ldots, \phi'_m$, with $\phi_1, \ldots, \phi_n$ and $\phi'_1, \ldots, \phi'_m$ being two (possibly empty) comma-separated lists of formulae, distinguished by the sequent arrow "$\implies$" (written as "`==>`" in both input and output of the KeY system). The intuitive meaning of a sequent is: if we assume all formulae $\phi_1, \ldots, \phi_n$ to hold, then *at least one* of the formulae $\phi'_1, \ldots, \phi'_m$ holds. In our particular calculus, the order of formulae within $\phi_1, \ldots, \phi_n$ and within $\phi'_1, \ldots, \phi'_m$ does not matter. Therefore, we can for instance write "$\Gamma \implies \phi \rightarrow \psi, \Delta$" to refer to sequents where *any* of the right-hand side formulae is an implication. (Γ and Δ are both used to refer to arbitrary, and sometimes empty, lists of formulae.) We refer to Chap. 2, Sect. 2.5, for a proper introduction of a (simple first-order) sequent calculus. The example used there is exactly the one we use here. We recommend to double-check the following steps with the on paper proof given there.

   We start proving the given sequent with the KeY system, however in a very interactive manner, step by step introducing and explaining the different aspects of the calculus and system. This purpose is really the *only* excuse to *not* let KeY prove this automatically.

   Even if we perform all steps "by hand" for now, we want the system to minimise interaction, e.g., by not asking the user for an instantiation if the system can find one itself. For this, please make sure that the "Minimize interaction" option (at **Options** → **Minimize interaction**) is checked.

Applying the First Rule

The sequent `==> p & q -> q & p` displayed in the **Current Goal** pane states that the formula `p & q -> q & p` holds *unconditionally* (no formula left of "`==>`"), and *without alternatives* (no other formula right of "`==>`"). This is an often encountered pattern for proof obligations when starting a proof: sequents with empty left-hand sides, and only the single formula we want to prove on the right-hand side. It is the duty of the *sequent calculus* to, step by step, take such formulae apart, while collecting assumptions on the left-hand side, or alternatives on the right-hand side, until the sheer shape of a sequent makes it trivially true, which is the case when *both sides have a formula in common*. (For instance, the sequent $\phi_1, \phi_2 \implies \phi_3, \phi_1$ is trivially true. Assuming both, ϕ_1 and ϕ_2, indeed implies that "at least one of ϕ_3 and ϕ_1" hold, namely ϕ_1.) It is such primitive shapes which we aim at when proving.

"Taking apart" a formula in a sense refers to breaking it up at the top-level operator. The displayed formula `p & q -> q & p` does not anymore show the brackets of the formula in the problem file. Still, for identifying the leading operator it is not required to memorise the built in operator precedences. Instead, the term structure gets clear when, with the mouse pointer, sliding back and forth over the formula area, as the sub-formula (or sub-term) under the symbol currently pointed at always gets highlighted. To get the whole formula highlighted, the user needs to point to the implication symbol "`->`", so this is where we can break up the formula.

Next we want to *select a rule* which is meant specifically to break up an implication on the right-hand side. This kind of user interaction is supported *by the system offering only those rules which apply to the highlighted formula, resp. term* (or, more precisely, to its leading symbol). A click on "`->`" will open a context menu for rule selection, offering several rules applicable to this implication, among them **impRight**:

$$\text{impRight}\ \ \frac{\Gamma, \phi \implies \psi, \Delta}{\Gamma \implies \phi \rightarrow \psi, \Delta}$$

Note that, strictly speaking, both the *premiss* $\Gamma, \phi \implies \psi, \Delta$ and the *conclusion* $\Gamma \implies \phi \rightarrow \psi, \Delta$ are not just plain sequents, but sequent *schemata*. In particular, ϕ and ψ are *schema variables*, to be instantiated with the two sub-formulae of the matching implication, when *applying* the rule.

As for any other rule, the *logical meaning* of this rule is downwards (concerning validity): if a sequent matching the premiss $\Gamma, \phi \implies \psi, \Delta$ is valid, we can conclude that the corresponding instance of the conclusion $\Gamma \implies \phi \rightarrow \psi, \Delta$ is valid as well. Accordingly, the *operational meaning* during proof construction goes upwards: the problem of proving a sequent which matches $\Gamma \implies \phi \rightarrow \psi, \Delta$ is reduced to the problem of proving the corresponding instance of $\Gamma, \phi \implies \psi, \Delta$. During proof construction, a rule is therefore applicable only to situations where the current goal matches the rule's *conclusion*. The proof will then be extended by the new sequent re-

sulting from the rule's premiss. (See below for a generalisation to multiple premisses).

To see this in action, we click at **impRight** in order to apply the rule to the current goal. This produces the new sequent `p & q ==> q & p`, which becomes the new current goal. By "goal", we mean a sequent to which no rule is yet applied. By "current goal" we mean the goal in focus, to which rules can be applied currently.

Inspecting the Emerging Proof

The user may have noticed the **Proof** tab *as part of the tabbed pane in the lower left corner*. It displays the structure of the (unfinished) proof we have achieved so far, showing all the nodes of the current proof, numbered consecutively, and labelled either by the name of the rule which was applied to that node, or by "OPEN GOAL" in case of a goal. The *blue* highlighted node is always the one which is detailed in the big pane. So far, this was always a goal, such that the big pane was called "**Current Goal**". But if the user clicks at an *inner node*, in our case on the one labelled with impRight, that node gets detailed in the big pane now called "**Inner Node**". It shows not only the sequent of that node, but also the *Upcoming rule application*, in a notation we come to in a minute.

Note that the (so far linear) proof tree displayed in the **Proof** tab has its root on the top, and grows downwards, as is common for trees displayed in GUIs. On paper, however, the traditional way to depict sequent proofs is bottom-up, as is done all over in this book. In that view, the structure of the current proof (with the upper sequent being the current goal) is:

$$\frac{p \mathbin{\&} q \Longrightarrow q \mathbin{\&} p}{\Longrightarrow p \mathbin{\&} q \mathbin{-\!>} q \mathbin{\&} p}$$

For the on-paper presentation of the proof to be developed, we again refer to Section 2.5. Here, we concentrate on the development and presentation via the KeY GUI instead.

Understanding the First Taclet

With the inner node still highlighted in the **Proof** tab, we look at the rule information given in the **Inner Node** pane, saying:

```
—— KeY Output ————————————————————————————————
impRight {
  \find ( ==> b -> c )
  \replacewith ( b ==> c )
  \heuristics ( alpha )
}
——————————————————————————————————— KeY Output ——
```

What we see here is what is called a *taclet*. Taclets are a framework for sequent calculi, a declarative language for programming the rules of a sequent calculus. The taclet framework was developed as part of the KeY project. The depicted taclet is the one which in the KeY system *defines* the rule **impRight**. In this chapter, we give just a hands-on explanation of the few taclets we come across. For a good introduction and discussion of the taclet framework, we refer to Chap. 4.

The taclet **impRight** captures what is expressed in the traditional sequent calculus style presentation of **impRight** we gave earlier, and a little more. The schema "`b -> c`" in the `\find` clause indicates that the taclet is applicable to sequents if one of its formulae is an implication, with `b` and `c` being schema variables matching the two sub-formulae of the implication. Further down the **Inner Node** pane, we see that `b` and `c` are indeed of kind "`\formula`":

─── KeY Output ──

```
\schemaVariables {
  \formula b;
  \formula c;
}
```

── KeY Output ───

The sequent arrow "`==>`" in "`\find(==> b -> c)`" further restricts the applicability of the taclet to the *top-level*[3] of the sequent only, and, in this case, to implications on the *right-hand side* of the sequent (as "`b -> c`" appears right of "`==>`"). The `\replacewith` clause describes how to construct the *new* sequent from the current one: first the matching implication (here `p & q -> q & p`) gets deleted ("`replace-`"), and then the subformulae matching `b` and `c` (here `p & q` and `q & p`) are added ("`-with`") to the sequent. Which side of the sequent `p & q` resp. `q & p` are added to is indicated by the relative position of `b` and `c` w.r.t. "`==>`" in the argument of `\replacewith`. The result is the new sequent `p & q ==> q & p`. It is a very special case here that `\find(==> b -> c)` matches the whole old sequent, and `\replacewith(b ==> c)` matches the whole new sequent. Other formulae could appear in the old sequent. Those would remain unchanged in the new sequent. In other words, the Γ and Δ traditionally appearing in on-paper presentations of sequent rules are omitted in the taclet formalism. (Finally, with `\heuristics(alpha)` the taclet declares itself to be part of the `alpha` heuristics, which only matters for the execution of automated strategies.)

The discussed taclet is the complete definition of the **impRight** rule in KeY, and all the system knows about the rule. The complete list of available taclets can be viewed in the **Rules** tab *as part of the tabbed pane in the lower left corner, within the "Taclet Base" folder*. To test this, we click that folder and scroll down the list of taclets, until impRight, on which we can click to be shown the same taclet we have just discussed. It might feel scary to see the

───────────────────────

[3] Modulo leading updates, see Sect. 10.2.3.

sheer mass of taclets available. Please note, however, that the *vast* majority of taclets is never in the picture when *interactively* applying a rule in any practical usage of the KeY system. Instead, most taclets are only used by automated symbolic execution of the programs contained in formulae (see below).

Backtracking the Proof

So far, we performed only one little step in the proof. Our aim was, however, to introduce some very basic elements of the framework and system. In fact, we even go one step back, with the help of the system. For that, we make sure that the OPEN GOAL is in focus (by clicking on it in the **Proof** tab). We can then *undo* the proof step which led to this goal, by clicking at **Goal Back** in the task bar. This action will put us back in the situation we started in, which is confirmed by both the **Current Goal** pane and the **Proof** tab. Note that **Goal Back**, here and in general, only undoes *one* step each time.

Viewing and Customising Taclet Tooltips

Before performing the next steps in our proof, we take a closer look at the *tooltips* for rule selection. (The reader may already have noticed those tooltips earlier.) If we again click at the implication symbol `->` appearing in the current goal, and *pre*-select the **impRight** rule in the opened context menu *simply by placing the mouse at* **impRight**, *without clicking yet*, we get to see a tooltip, displaying something similar, or identical, to the **impRight** taclet discussed above. The exact tooltip text depends on option settings which the user can configure. Depending on those settings, what is shown in the tooltip is *just the taclet as is*, or a certain *'significant' part* of it, in both cases either *with* or *without schema variables* already being *instantiated*. It would be unwise to commit, in this chapter, to the tooltip settings currently in place in the reader's KeY system. Instead, we control the options actively here, and discuss the respective outcome.

We open the tooltip options window *by* **View** → **ToolTip options**, and make sure that all parts of taclets are displayed *by making sure the "pretty-print whole taclet ..." checkbox is checked*. For now, we disable the instantiation of schema variables *by setting the "Maximum size ... of tooltips ... with schema variable instantiations displayed ..." to 0*. With these settings, the tooltips for pre-selected rules consist of the original taclets, nothing more, nothing less. (The reader might try this with the already familiar **impRight** rule.) This is a good setting for getting familiar with the taclets as such. The *effect* of a taclet to the current proof situation is, however, better captured by tooltips where the schema variables from the `\find` argument are already instantiated by their respective matching formula or term. *We achieve this by setting the "Maximum size ... of tooltips ... with schema variable instantiations displayed ..." to some higher value, say 40.* When trying the tooltip for

impRight with this, we see something like the original taclet, however with `b` and `c` already being instantiated with `p & q` and `q & p`, respectively:

```
──── Tooltip ────
impRight {
  \find ( ==> p & q -> q & p )
  \replacewith ( p & q ==> q & p )
  \heuristics ( alpha )
}
                                          ──── Tooltip ────
```

This "instantiated taclet"-tooltip tells us the following: If we clicked on the rule name, the formula `p & q -> q & p`, which we `\find` somewhere on the *right-hand side* of the sequent (see the formula's relative position compared to `==>` in the `\find` argument), would be `\replace`(d)`with` the two formulae `p & q` and `q & p`, where the former would be added to the *left-hand side*, and the latter to the *right-hand side* of the sequent (see their relative position compared to `==>` in the `\replacewith` argument). Note that, in this particular case, where the sequent only contains the matched formula, the arguments of `\find` and `\replacewith` which are displayed in the tooltip *happen to be* the *entire* old, resp., new sequent. This is not the case in general. The same tooltip would show up when preselecting **impRight** on the sequent: `r ==> p & q -> q & p, s`.

A closer look at the tooltip text in its current form (i.e., with the schema variables already being instantiated), reveals that the whole `\find` clause actually is redundant, as it is essentially identical with the anyhow highlighted text within the **Current Goal** pane. Also, the taclet's name is already clear from the preselected rule name in the context menu. On top of that, the `\heuristics` clause is actually irrelevant for the *interactive* selection of the rule. The only non-redundant piece of information is therefore the `\replacewith` clause (in this case). Consequently, the tooltips can be reduced to the minimum which is relevant for supporting the selection of the appropriate rule *by un-checking "pretty-print whole taclet . . . " option again*. The whole tooltip for **impRight** is the one-liner:

```
──── Tooltip ────
\replacewith ( p & q ==> q & p )
                                          ──── Tooltip ────
```

In general, the user might play around with different tooltip options in order to see which settings are most helpful. However, in the following steps, we assume these tooltips to show the full and unchanged taclet, so we switch back to our first setting *by checking "pretty-print whole taclet . . . " and setting the "Maximum size . . . of tooltips . . . with schema variable instantiations displayed . . . " to 0 again*.

Splitting Up the Proof

We apply **impRight** and consider the new goal `p & q ==> q & p`. For further decomposition we could break up the conjunctions on either sides of the sequent. By first selecting `q & p` on the right-hand side, we are offered the rule **andRight**, among others. The corresponding tooltip shows the following taclet:

─── Tooltip ──

```
andRight {
  \find ( ==> b & c )
  \replacewith ( ==> b );
  \replacewith ( ==> c )
  \heuristics ( beta, split )
}
```

── Tooltip ───

Here we see *two* `\replacewith`s, telling us that this taclet will construct *two* new goals from the old one, meaning that this is a *branching* rule.[4] Written as a sequent calculus rule, it looks like this:

$$\text{andRight}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \,\&\, \psi, \Delta}$$

We now generalise the earlier description of the meaning of rules, to also cover branching rules. The *logical meaning* of a rule is downwards: if a certain instantiation of the rule's schema variables makes *all* premises valid, then the corresponding instantiation of the conclusion is valid as well. Accordingly, the *operational meaning* during proof construction goes upwards: The problem of proving a goal which matches the conclusion is reduced to the problem of proving *all* the (accordingly instantiated) premises.

If we apply **andRight** in the system, the **Proof** tab shows the proof branching into two different `Cases`. In fact, both branches carry an OPEN GOAL. At least one of them is currently visible in the **Proof** tab, and highlighted *blue* to indicate that this is the new current goal, being detailed in the **Current Goal** pane as usual. The other OPEN GOAL might be hidden in the **Proof** tab (depending on the system settings), as the branches *not* leading to the current goal appear *collapsed* in the **Proof** tab by default. A collapsed/expanded branch can however be expanded/collapsed *by clicking on* ⊞/⊟.[5] If we expand the yet collapsed branch, we see the full structure of the proof, with

---

[4] Note that it is not the "`split`" argument of the `heuristics` clause which makes this rule branching. The "`split`" is only the name of the heuristics this taclet claims to be member of. The fact that the taclet would branch a proof is the *reason* for (and not a *consequence* of) making it member of the `split` heuristics.

[5] Bulk expansion, resp., bulk collapsing of all proof branches is offered by a context menu *via right click* in the **Proof** tab.

both OPEN GOALs being displayed. We can even switch the current goal by clicking on any of the OPEN GOALs.[6]

An on-paper presentation of the current proof would look like this:

$$\frac{\dfrac{p \,\&\, q \Longrightarrow q \qquad p \,\&\, q \Longrightarrow p}{p \,\&\, q \Longrightarrow q \,\&\, p}}{\Longrightarrow p \,\&\, q \rightarrow q \,\&\, p}$$

The reader might compare this presentation with the proof presented in the **Proof** tab by again clicking on the different nodes (or by clicking just anywhere within the **Proof** tab, and browsing the proof using the arrow keys).

Closing the First Branch

To continue, we put the OPEN GOAL `p & q ==> q` in focus again. Please recall that we want to reach a sequent where identical formulae appear on both sides (as such sequents are trivially true). We are already very close to that, just that `p & q` remains to be decomposed. Clicking at `&` offers the rule **andLeft**, as usual with the tooltip showing the taclet, here:

──── Tooltip ──

```
andLeft {
  \find ( b & c ==> )
  \replacewith ( b, c ==> )
  \heuristics ( alpha )
}
```
──────────────────────────────────── Tooltip ────

which corresponds to the sequent calculus rule:

$$\mathsf{andLeft} \ \ \frac{\Gamma, \phi, \psi \Longrightarrow \Delta}{\Gamma, \phi \,\&\, \psi \Longrightarrow \Delta}$$

We apply this rule, and arrive at the sequent `p, q ==> q`. We have arrived where we wanted to be, at a goal which is *trivially true* by the plain fact that one formula appears on both sides, *regardless* of how that formula looks like. (Of course, the sequents we were coming across in this example were all trivially true in an intuitive sense, but always only because of the particular form of the involved formulae.) In the sequent calculus, sequents of the form $\Gamma, \phi \Longrightarrow \phi, \Delta$ are considered valid *without any need of further reduction*. This argument is also represented by a rule, namely:

$$\mathsf{closeGoal} \ \ \frac{}{\Gamma, \phi \Longrightarrow \phi, \Delta}$$

---

[6] Another way of getting an overview over the open goals, and switch the current goal, is offered by the **Goals** tab.

In general, rules with no premiss *close* the branch leading to the goal they are applied to, or, as we say in short (and a little imprecise), *close the goal* they are applied to.

The representation of this rule as a taclet calls for two new keywords which we have not seen so far. One is `\closegoal`, having the effect that taclet application does not produce any new goal, but instead closes the current proof branch. The other keyword is `\assumes`, which is meant for expressing assumptions on formulae *other than* the one matching the `\find` clause. Note that, so far, the applicability of rules always depended on *one* formula only. The applicability of **closeGoal** however depends on *two* formulae (or, more precisely, on two formula occurrences). The second formula is taken care of by the `\assumes` clause in the **closeGoal** taclet:

```
──── Taclet ──────────────────────────────────────────────
closeGoal {
  \assumes ( b ==> )
  \find ( ==> b )
  \closegoal
  \heuristics ( closure )
}
────────────────────────────────────────── Taclet ────
```

Note that this taclet is not symmetric (as opposed to the **closeGoal** sequent rule given above). To apply it interactively on our **Current Goal** `p, q ==> q`, we have to put the *right-hand side* `q` into focus (cf. "`\find(==> b)`"). But the `\assumes` clause makes a taclet applicable only in the presence of further formulas, in this case the identical formula on the *left-hand side* (cf. "`\assumes(b ==>)`").[7]

This discussion of the **closeGoal** sequent rule and a corresponding **closeGoal** taclet shows that taclets are more fine grained than rules. They contain more information, and consequently there is more than one way to represent a sequent rule as a taclet. To see another way of representing the above *sequent rule* **closeGoal** by a taclet, the reader might click on the `q` on the *left-hand side* of `p, q ==> q`, and pre-select the taclet **closeGoalAntec**. The tooltip will show the taclet:

---

[7] This is not the whole truth. In KeY, one can even enforce the application of taclets with the `assumes` clause not *syntactically* satisfied by the sequent. In that case, an additional branch is created, allowing us to *prove* the assumptions not yet present in the sequent. We may, however, ignore that possibility for the time being. Moreover, in the case of closing rules, the usage of this feature is particularly useless, as it leads to a loop in the proof.

---— Tooltip ———————————————————

```
closeGoalAntec {
  \assumes ( ==> b )
  \find ( b ==> )
  \closegoal
}
```
———————————————————— Tooltip —

We, however, proceed by applying the taclet **closeGoal** on the right-hand side formula q. (With the current settings, the taclet application should happen instantly. In case there opens a dialogue, the reader might select **Cancel**, check the **Minimize interaction** option, and re-apply **closeGoal**.) After this step, the **Proof** pane tells us that the proof branch that has just been under consideration is closed, which is indicated by that branch ending with a "Closed goal" node *coloured green*. The system has automatically changed focus to the next OPEN GOAL, which is detailed in the **Current Goal** pane as the sequent p & q ==> p.

Pruning the Proof Tree

We apply **andLeft** to the & on the left, in the same fashion as we did on the other branch. Afterwards, we *could* close the new goal p, q ==> p, but we refrain from doing so. Instead, we compare the two branches, the closed and the open one, which both carry a node labelled with andLeft. When inspecting these two nodes again (by simply clicking on them), we see that we broke up the same formula, the left-hand side formula "p & q", on both branches. It appears that we branched the proof too early. Instead, we should have applied the (non-branching) **andLeft**, once and for all, before the (branching) **andRight**. *This is a good strategy in general, to delay proof branching as much as possible, thereby avoiding double work on the different branches.* Without this strategy, more realistic examples with hundreds or thousands of proof steps would become completely infeasible.

In our tiny example here, it seems not to matter much, but it is instructive to apply the late splitting also here. We want to re-do the proof from the point where we split too early. Instead of re-loading the problem file, we can *prune* the proof at the node labelled with andRight *by right-click on that node, and selecting* **Prune Proof**. As a result, large parts of the proof are pruned away, and the second node, with the sequent p & q ==> q & p, becomes the **Current Goal** again.

Closing the First Proof

This time, we apply **andLeft** *before* we split the proof via **andRight**. The two remaining goals, p, q ==> q and p, q ==> p, we close by applying **closeGoal** to the right-hand q and p, respectively. By closing all branches, we have

actually closed the entire proof, as we can see from the **Proof closed** window popping up now.

Altogether, we have proved the validity of the *sequent* at the root of the proof tree, here `==> p & q -> q & p`. As this sequent has only one formula, placed on the right-hand side, we have actually proved validity of that *formula* `p & q -> q & p`, the one stated as the `\problem` in the file we loaded.

Proving the Same Formula Automatically

As noted earlier, the reason for doing all the steps in the above proof manually was that we wanted to learn about the system and the used artefacts. Of course, one would otherwise prove such a formula automatically, which is what we do in the following.

Before loading the same problem again, we can choose whether we abandon the current proof, or alternatively keep it in the system. Abandoning a proof would be achieved via the menu: **Proof** → **Abandon Task**. It is however possible to keep several (finished or unfinished) proofs in the system, so we suggest to start the new proof while keeping the old one. This will allow us to compare the proofs more easily.

Loading the file `andCommutes.key` again can be done in the same fashion as before *or alternatively via the "Load last opened file." button* 📁 *in the toolbar*. The system might then ask whether the problem should be opened in a new environment or in the already existing one. We choose **Open in new environment** (and do so also in the following, unless stated otherwise). The system might further ask whether the previous proof should be marked for reuse. We **Cancel** that dialog here (but refer to Chap. 13 on the topic of *proof reuse*). Afterwards, we see a second *'task'* being displayed in the **Task** pane. One can even switch between the different tasks by clicking in that pane.

The newly opened proof shows the **Current Goal ==>** `p & q -> q & p`, just as last time. In order to let KeY prove this automatically, we first have to select a *proof search strategy*, which is done in the **Proof Search Strategy** tab. The most important strategy offered there is the strategy for JAVA dynamic logic **Java DL**, with its variations for loop/method treatment. However, our sequent here does not contain programs, and therefore falls in the pure first-order fragment of the logic. (Here, it is even only propositional.) Therefore, the strategy for pure first-order logic **FOL** is appropriate, and we select that. (The slider controlling the maximal number of automatic rule applications should be at least **1000**, which will suffice for all examples in this chapter).

By pressing the "automated proof search" button ▶, we start running the chosen strategy. A complete proof is constructed immediately. Its shape (see **Proof** tab) depends heavily on the current implementation of the FOL strategy. However, it is most likely different from the proof we constructed interactively before. (For a comparison, we switch between the tasks in the **Task** pane.)

Rewrite Rules

With *the current implementation of* the FOL strategy, only the first steps of the automatically constructed proof, **impRight** and **andLeft**, are identical with the interactively constructed proof from above, leading to the sequent `p, q ==> q & p`. After that, the proof does *not* branch, but instead uses the rule **replaceKnownLeft**:

─── Taclet ───

```
replaceKnownLeft {
  \assumes ( b ==>  )
  \find ( b )
  \sameUpdateLevel
  \replacewith ( true )
  \heuristics ( replace_known )
}
```

─── Taclet ───

It has the effect that any formula (`\find(b)`) which has *another appearance* on the left side of the sequent (`\assumes(b ==> )`) can be replaced by `true`. Note that the `\find` clause does not contain "`==>`", and therefore does not specify where the formula to be replaced shall appear. However, only one formula at a time gets replaced.

Taclets with a "`==>`"-free `\find` clause are called *rewrite taclets* or *rewrite rules*. The argument of `\find` is a schema variable of kind `\formula` or `\term`, matching formulae resp. terms at *arbitrary* positions, which may even be nested. (The position can be further restricted. The restriction `\sameUpdateLevel` in this taclet is however not relevant for the current example.) When we look at how the taclet was used in our proof, we see that indeed the *sub-formula* `q` of the formula `q & p` has been rewritten to `true`, resulting in the sequent `p, q ==> true & p`. The following rule application simplifies the `true` away, after which **closeGoal** is applicable again.

Saving a Proof

Before we leave the discussion of the current example, we *save* the just accomplished proof (admittedly for no other reason than practising the saving of proofs). For that, we select **File → Save ...** *or alternatively the "Save current proof." button* 🖫 *in the toolbar*. The opened file browser dialogue allows to locate and name the proof file. A sensible name would be `andCommutes.proof`, but any name would do, *as long as the file extension is* "`.proof`". It is completely legal for a proof file to have a different naming than the corresponding problem file. This way, it is possible to save several proofs for the same problem.

Proofs can actually be saved at any time, regardless of whether they are finished or not. An unfinished proof can be continued when loaded again.

10.2.2 Exploring Terms, Quantification, and Instantiation: Building First-Order Proofs

After having looked at the basic usage of the KeY prover, we want to extend the discussion to more advanced features of the logic. The example of the previous section did only use propositional connectives. Here, we discuss the basic handling of first-order formulae, containing terms, variables, quantifiers, and equality. As an example, we prove a `\problem` which we load from the file `projection.key`:

───── KeY Problem File ─────

```
\sorts {
        s;
}
\functions {
        s f(s);
        s a;
}
\problem {
        ( \forall s x; f(f(x)) = f(x) )  ->  f(a) = f(f(f(a)))
}
```

───────────────────────────────── KeY Problem File ─────

The file first declares a function `f` (of type `s → s`) and a constant `a` (of sort `s`). The first part of the `\problem` formula, `\forall s x; f(f(x)) = f(x)`, says that `f` is a *projection*: For all `x`, applying `f` twice is the same as applying `f` once. The whole `\problem` formula then states that `f(a)` and `f(f(f(a)))` are equal, given `f` is a projection.

Instantiating Quantified Formulae

We prove this simple formula interactively, for now. After loading the problem file, and applying **impRight** to the initial sequent, the **Current Goal** is: `\forall s x; f(f(x)) = f(x) ==> f(a) = f(f(f(a)))`.

   We proceed by deriving an additional assumption (i.e., left-hand side formula) `f(f(a)) = f(a)`, by instantiating `x` with `a`. For the interactive instantiation of quantifiers, KeY supports *drag and drop* of terms over quantifiers (whenever the instantiation is textually present in the current sequent). In the situation at hand, we can drag any of the two "a" onto the quantifier `\forall` *by clicking at "a", holding and moving the mouse, to release it over the "forall"*. As a result of this action, the new **Current Goal** features the *additional* assumption `f(f(a)) = f(a)`.

   There is something special to this proof step: Though it was triggered interactively, we have not been specific about which taclet to apply. The **Proof** pane, however, tells us that we just applied the taclet **instAll**. To see the very taclet, we can click at the previous proof node, marked with instAll, such that the **Inner Node** pane displays:

---— KeY Output ————————————————————————

```
instAll {
  \assumes ( \forall u; b ==>  )
  \find ( t )
  \add ( {\subst u; t}b ==>  )
}
```

————————————————————————— KeY Output ——

"{\subst u; t}b" means that (the match of) u is substituted by (the match of) t in (the match of) b, during taclet application.

Making Use of Equations

Now we can use the new equation `f(f(a)) = f(a)` to simplify the term `f(f(f(a)))`, meaning we *apply* the equation to the `f(f(a))` subterm of `f(f(f(a)))`. This action can again be performed via drag and drop, here by dragging the equation on the left side of the sequent, and dropping it over the `f(f(a))` subterm of `f(f(f(a)))`.[8] *In the current system, there opens a context menu, allowing to select either of two taclets with the identical display name* **applyEquality**. *The taclets are however different, see their tooltips. For our example, it does not matter which one is selected.*

Afterwards, the right-hand side equation has changed to `f(a) = f(f(a))`, which looks almost like the left-hand side equation. We can proceed either by swapping one equation, or by again applying the left-hand side equation on a right-hand side term. It is instructive to discuss both alternatives here.

First, we select `f(a) = f(f(a))`, and apply **commuteEq**. The resulting goal has two identical formulae on both sides of the sequent, so we *could* apply **closeGoal**. But instead, just to demonstrate the other possibility as well, we backtrack (via **Goal Back**), leading us back to the **Current Goal** `f(f(a)) = f(a), ... ==> f(a) = f(f(a))`.

The other option is to apply the left-hand equation to `f(f(a))` on the right (via drag and drop). Afterwards, we have the *tautology* `f(a) = f(a)` on the right. By preselecting that formula, we get offered the taclet **closeEq**, which transforms the equation into `true`.

Closing "by True" and "by False"

So far, all goals we ever closed featured identical formulae on both sides of the sequent. We have arrived at the second type of closable sequents: one with `true` on the *right* side. We close it by highlighting `true`, and selecting the taclet **closeByTrue**, which is defined as:

---

[8] More detailed, we move the mouse over the "=" symbol, such that the whole of `f(f(a)) = f(a)` is highlighted. We click, hold, and move the mouse, over the second "f" in `f(f(f(a)))`, such that exactly the subterm `f(f(a))` gets highlighted. Then, we release the mouse.

—— Taclet ——————————————————————

```
closeByTrue {
  \find ( ==> true )
  \closegoal
  \heuristics ( closure )
}
```

—————————————————————— Taclet ——

This finishes our proof.

Without giving an example, we mention here the third type of closable sequents, namely those with `false` on the *left* side, to be closed by:

—— Taclet ——————————————————————

```
closeByFalse {
  \find ( false ==> )
  \closegoal
  \heuristics ( closure )
}
```

—————————————————————— Taclet ——

This is actually a very important type of closable sequent. In many examples, a sequent can be proved by showing that the assumptions (i.e., the left-hand side formulae) are contradictory, meaning that `false` can be derived on the left side.

Using Taclet Instantiation Dialogues

In our previous proof, we used the "drag-and-drop-directly-in-goal" feature offered by the KeY prover. This kind of user interaction can be seen as a shortcut to another kind of user interaction: the usage of *taclet instantiation dialogues*. While the former is most convenient, the latter is more general and should be familiar to each KeY user. Therefore, we re-construct the (in spirit) same proof, this time using such a dialogue explicitly.

After again loading the problem file `projection.key` (and **Cancel**ling the **re-use** dialogue), we apply **impRight** to the initial sequent, just like before. Next, to instantiate the quantified formula `\forall s x; f(f(x)) = f(x)`, we highlight that formula, and apply the taclet **allLeft**, which is defined as:

—— Taclet ——————————————————————

```
allLeft {
  \find ( \forall u; b ==> )
  \add ( {\subst u; t}b ==> )
  \heuristics ( gamma )
}
```

—————————————————————— Taclet ——

This opens a **Choose Taclet Instantiation** dialogue, allowing the user to choose the (not yet determined) instantiations of the taclet's schema variables. The taclet at hand has three schema variables, `b`, `u`, and `t`. The instantiations of `b` and `u` are already determined to be `f(f(x)) = f(x)` and `x`, just by matching the highlighted sequent formula `\forall s x; f(f(x)) = f(x)` with the `\find` argument `\forall u; b`. The instantiation of `t` is, however, left open, to be chosen by the user. We can type "`a`" in the corresponding input field of the dialogue,[9] and click **Apply**. As a result, the `f(f(a)) = f(a)` is added to the left side of the sequent. The rest of the proof goes exactly as discussed before. The reader may finish it herself.

Loading a Proof

We want to compare the proof which we just have constructed interactively with a proof the FOL strategy would construct *automatically*. The user could load the same `\problem` again, and run the FOL strategy. However, to not make the discussion too dependent on the current implementation of the FOL strategy, we instead *load a proof* which was automatically constructed, and saved, at the time this chapter was written.

The loading of a proof is done in exactly the same way as loading a problem file, with the only difference that a `.proof` file is selected (instead of a `.key` file). We load the proof `projectionAutomat.proof`.

Discovering Meta Variables and Constraints

We inspect the loaded proof. The sequent of the first inner node (labelled with "`1:`") tells us that this is actually a proof of the same problem as before. (The name of the proof file is just an indication, nothing more.) We can see that the FOL strategy decided to, as a first step, reorient the equation `f(a) = f(f(f(a)))`. Afterwards, nothing special is happening until node "`3:`". To the sequent of that node, the FOL strategy applied **allLeft**, as we did in our previous proof. However, the resulting goal (number "`4:`") looks different:

──── KeY Output ────────────────────────────────

```
f(f(X_0)) = f(X_0),
\forall s x; f(f(x)) = f(x)
==>
f(f(f(a))) = f(a)
```

──────────────────────────────── KeY Output ────

Note that the formula newly introduced by **allLeft** is `f(f(X_0)) = f(X_0)`
(and not, like in the previous proof, `f(f(a)) = f(a)`). Not only has the
FOL strategy chosen an instantiation (of the schematic taclet variable `t`)
which is different from our previous choice. The fact that the instantiation
"`X_0`" starts with a *capital letter* tells us that this is a *meta variable*, which
intuitively stands for a term yet to be determined. Before we discuss this new
concept a bit more, we check out the next node. The sequent of node num-
ber "5:" contains the *constrained formula* `f(f(a)) = f(a) << [ X_0 = a ]`,
which intuitively says something like: "The (unconstrained) formula `f(f(a))`
`= f(a)` is only present really if the constraint `X_0 = a` is fulfilled, otherwise
we imagine it not being there."

Meta variables, and constraints over meta variables, are concepts which
serve the purpose of proof *automation*. They are less important for (purely)
interactive proving. The KeY project, however, follows an integrated ap-
proach, where automated and interactive proof steps are intertwined. There-
fore, a KeY user should at least have a rudimentary idea of what meta vari-
ables and constraints are all about.

Understanding Meta Variables and Constraints (to a Certain Extent)

Instantiation of quantified variables is a crucial task, and typically more dif-
ficult than in the example at hand. For automated strategies it is often infea-
sible to guess the right instance at the point of quantifier instantiation (like
when applying **allLeft**). To help solving this problem, the automated theorem
proving community has invented the notion of meta variables[10]. These vari-
ables are not part of the actual logic under consideration. Rather, they are
employed on the meta level, used as place-holders (for concrete terms) in the
proof.

Meta variables allow to *delay* the guessing of *concrete instances*. During
proof construction, they are introduced as *generic instances*, to be refined
later on with the help of constraints[11]. A typical point when to refine a meta
variable is for instance a situation where a concrete instance *would* allow to
close the current goal.

A thorough understanding of meta variable constraint handling goes be-
yond the scope of this chapter ($\Rightarrow$ Sect. 4.3), and luckily is *not needed* for
a user in order to effectively use the KeY prover. Some rudimentary under-
standing is however helpful, and we try to provide that here by reconstructing
the current proof interactively, in a kind of slow motion picture of the fully
automatic proof construction.

---

[10] This kind of variables are known in the tableau-style theorem proving community
under the name of "free variables", see Fitting [1996].

[11] Historically, refining meta variables was not done with the help of constraints,
but via destructive substitution. The usage of constraints for this purpose was in-
vented within the KeY project, by Giese [2001], mainly to achieve a backtracking
free calculus.

For this, we again load `projection.key` (while keeping the proof from `projectionAutomat.proof` in the system, for comparison). First, we apply **commuteEq** on `f(a) = f(f(f(a)))`, and afterwards **impRight**. Selecting **allLeft** on the quantified formula opens the **Choose Taclet Instantiation** dialogue, just as before. This time, we point the reader to the massage pane of that dialogue, telling us that the "Instantiation is OK", already. This might come as a surprise, as the instantiation of `t` is still left open. However, the proof system here allows the user to *not* commit to a concrete instantiation for `t`, in which case the system will instantiate `t` with a new meta variable, in the current implementation with "X_0". This is exactly the effect of clicking **Apply** now (while leaving the Instantiation field for `t` empty).

As a result, the new equation `f(f(X_0)) = f(X_0)` is added on the left-hand side. Next, we would like to use that equation, to rewrite the `f(f(a))` sub-term of right-hand side formula `f(f(f(a))) = f(a)`. It is intuitively clear that this is only possible *if* `X_0` is equal to `a`. In the calculus, this "if" is reflected in the following way: *Rewriting* `f(f(a))` *in* `f(f(f(a))) = f(a)` *with* `f(f(X_0)) = f(X_0)` *results in* `f(f(a)) = f(a) << [ X_0 = a ]`.

This is not the whole truth yet, as we can see when performing this step in the system. We drag the equation `f(f(X_0)) = f(X_0)`, and drop it over `f(f(a))`. In the resulting sequent, we can see that the original formula `f(f(f(a))) = f(a)` was kept, in addition to the rewritten, constrained formula. By not throwing away this formula, we still cover the case where `X_0` is *not* equal to `a`.

Now we can close our proof, by applying **closeGoal** on `f(f(a)) = f(a) << [ X_0 = a ]`. On the surface, this will immediately finish our proof. Internally, both `f(f(X_0)) = f(X_0)` and `f(f(a)) = f(a) << [ X_0 = a ]` need to serve as instantiation for the single schema variable `b` of the taclet **closeGoal**, and therefore need to be identified. What happens is that the application of the taclet first matches the two formulae, i.e., it rewrites `f(f(X_0)) = f(X_0)` to `f(f(a)) = f(a) << [ X_0 = a ]`, and actually closes the goal afterwards.

Please recall that the creation, manipulation, and usage of meta variables and their constraints is entirely done by the system, not by the user. Therefore, the purpose of the above explanations is mainly to allow the user to interact on proof goals which were constructed automatically. (For some more discussion, we refer to Sect. 4.3.)

As a general advice, when instantiating quantifiers interactively, we recommend to use *concrete* instances instead of meta variables whenever the user is clear about which instance is needed for the current proof.

Skolemising Quantified Formulae

We will now consider a slight generalisation of the theorem we have just proved. Again, we assume that `f` is a projection. But instead of showing `f(a) = f(f(f(a)))`, for a particular `a`, we show `f(y) = f(f(f(y)))` *for*

all y. For this we load `generalProjection.key`, and apply **impRight**, which results in the sequent:

— KeY Output —————————————————————————————

```
\forall s x; f(f(x)) = f(x) ==> \forall s y; f(y) = f(f(f(y)))
```
——————————————————————————————— KeY Output —

As in the previous proof, we will have to instantiate the quantified formula on the left. But this time we also have to deal with the quantifier on the right. Luckily, that quantifier can be eliminated altogether, by applying the rule **allRight**, which results in:[12]

— KeY Output —————————————————————————————

```
\forall s x;  f(f(x)) = f(x) ==> f(y_0) = f(f(f(y_0)))
```
——————————————————————————————— KeY Output —

We see that the quantifier disappeared, and the variable y got replaced. The replacement, y_0, is a *constant*, which we can see from the fact that y_0 is not quantified. Note that in our logic each logical variable appears in the scope of a quantifier binding it.[13] Therefore, y_0 can be nothing but a constant. Moreover, y_0 is a *new* symbol.

Eliminating quantifiers by introducing new constants is called *skolemisation* (after the logician Thoralf Skolem). In a sequent calculus, universal quantifiers (`\forall`) on the right, and existential quantifiers (`\exists`) on the left side, can be eliminated this way, leading to sequents which are equivalent (concerning provability), but simpler. This should not be confused with quantifier *instantiation*, which applies to the complementary cases: (`\exists`) on the right, and (`\forall`) on the left, see our discussion of **allLeft** above. (It is instructive to look at all four cases in combination, see Sect. 2.5.4, Chapt. 2.)

Skolemisation is a simple proof step, and is normally done fully automatically. We only discuss it here to give the user some understanding about new constants (or functions, see below) that might show up during proving.

To see the taclet we have just applied, we select the inner node labelled with allRight. The **Inner Node** pane reveals the taclet:

— KeY Output —————————————————————————————

```
allRight {
  \find ( ==> \forall u; b )
  \varcond ( \new(sk, \dependingOn(b)) )
  \replacewith ( ==> {\subst u; sk}b )
  \heuristics ( delta )
}
```
——————————————————————————————— KeY Output —

---

[12] Note that the particular name y_0 can differ, depending on the implementation.
[13] This is not the case for *meta* variables, as they are not logical variables.

It tells us that the rule removes the quantifier matching `\forall u;`, and that (the match of) `u` is `\substituted` by (the match of) `sk` in the remaining formula (matching `b`). During application of this taclet, the schema variable `sk` will be instantiated with a Skolem term, in many cases a Skolem constant only. The instantiation of `sk` is restricted by the schema variable condition `\varcond`: First of all, the instantiation of `sk` must be a `\new` term. Second, the particular instantiation of `sk` is determined `\dependingOn` the (match of) `b`.

Two things remain to be explained here. Why are Skolem *constants* not always sufficient, and how do proper Skolem *terms* depend on the formula at hand? Both issues are related to the potential presence of "meta variables" in a sequent. Without that, new constants would indeed be sufficient. But in the presence of meta variables, newly introduced constants could later be identified with "older" meta variables, leading to unsound reasoning. This is the reason why Skolem *terms* are used in the general case. Those terms are of the form $f(X_1, \ldots, X_n)$, with f being a new function symbol, and $X_1, \ldots, X_n$ being the meta variables appearing in the formula this term is `\dependingOn`.[14] Here, we do not give an example where proper Skolem *terms* appear. However, these explanations should help the user to not feel uncomfortable when confronted with automatically introduced Skolem constants/functions/terms.

The rest of our current proof goes exactly like for the previous problem formula. Instead of further discussing it here, we simply run the "FOL" strategy to resume the proof.

Employing External Decision Procedures

Apart from strategies, which apply taclets automatically, KeY also employs external decision procedure tools for increasing the automation of proofs. The field of decision procedures is very dynamic, and so is the way in which KeY makes use of them. The user can choose among the available decision procedure tools under **Options → Decision Procedure Config**. With **Simplify** selected there, we load `generalProjection.key` once more, and push the **Run Simplify** button in the tool bar. This closes the proof in one step(!), as the **Proof** tab is telling us. Decision procedures can be very efficient on certain problems. On the down side, we sacrificed proof transparency here.

In a more realistic setting, we use decision procedures towards the end of a proof (branch), to close first-order goals which emerged from proving problems that go beyond the scope of decision procedures.

---

[14] Note that a (finite) term is always syntactically different from its proper subterms. Therefore, the newly introduced term $f(X_1, \ldots, X_n)$ can never be instantiated in a way that makes it syntactically equal to either of $X_1, \ldots, X_n$. This property is actually the sole purpose of the form $f(X_1, \ldots, X_n)$. The fact that *only* the meta variable of the quantified formula, not those of the whole sequent, are needed goes back to a result by Hähnle and Schmitt [1994].

10.2.3 Exploring Programs in Formulae: Building Dynamic Logic Proofs

Not first-order logic, and certainly not propositional logic, is the real target of the KeY prover. Instead, the prover is designed to handle proof obligations formulated in a substantial extension of first-order logic, *dynamic logic* (DL). What is dynamic about this logic is the notion of the world, i.e., the interpretation (of function/predicate symbols) in which formulae (and sub-formulae) are evaluated. In particular, a formula and its sub-formulae can be interpreted in *different* worlds.

The other distinguished feature of DL is that descriptions of how to construct one world from another are explicit in the logic, in the form of *programs*. Accordingly, the worlds represent computation *states*. (In the following, we take "state" as a synonym for "world".) This allows us to, for instance, talk about the states both *before* and *after* executing a certain program, *within the same formula*.

Compared to first-order logic, DL employs two additional (mix-fix) operators: $\langle . \rangle .$ (diamond) and $[.].$ (box). In both cases, the first argument is a *program*, whereas the second argument is another DL formula. With $\langle p \rangle \varphi$ and $[p]\varphi$ being DL formulae, $\langle p \rangle$ and $[p]$ are called the *modalities* of the respective formula.

A formula $\langle p \rangle \varphi$ is valid in a state if, from there, an execution of p terminates normally and results in a state where φ is valid. As for the other operator, a formula $[p]\varphi$ is valid in a state from where execution of p does *either* not terminate normally *or* results in a state where φ is valid.[15] For our applications the diamond operator is way more important than the box operator, so we restrict attention to that.

One frequent pattern of DL formulae is "$\varphi \Rightarrow \langle p \rangle \psi$", stating that the program p, when started from a state where φ is valid, terminates, with ψ being valid in the post state. (Here, φ and ψ often are pure first-order formulae, but they can very well be proper DL formulae, containing programs themselves.)

Each variant of DL has to commit to a formalism used to describe the programs (i.e., the p) in the modalities. Unlike most other variants of DL, the KeY project's DL variant employs a real programming language, namely JAVA CARD. Concretely, p is a sequence of (zero, one, or more) JAVA CARD statements. Accordingly, the logic is called JAVA CARD DL.

The following is an example of a JAVA CARD DL formula:

$$x < y \Rightarrow \langle \texttt{int t = x; x = y; y = t;} \rangle\, y < x \qquad (10.1)$$

It says that in each state where the program variable x has a value smaller than that of the program variable y, the sequence of JAVA statements

---

[15] These descriptions have to be generalised when indeterministic programs are considered, which is not the case here.

"`int t = x; x = y; y = t;`" terminates, and afterwards the value of `y` is smaller than that of `x`. It is important to note that `x` and `y` are *program* variables, not to be confused with *logical* variables. In our logic, there is a strict distinction between both. Logical variables must appear in the scope of a quantifier binding them, whereas program variables cannot be quantified over. The formula (10.1) has no quantifier because it does not contain any logical variables.

As we will see in the following examples, both program variables and logical variables can appear mixed in terms and formulae, also together with logical constants, functions, and predicate symbols. However, inside the modalities, there can be nothing but (sequents of) *pure* JAVA statements.

For a more thorough discussion of JAVA CARD DL, please refer to Chap. 3.

Feeding the Prover with a DL Problem File

The file `exchange.key` contains the JAVA CARD DL formula (10.1), in the concrete syntax used in the KeY system:[16]

—— KeY Problem File ——————————————————

```
\programVariables { int x, y; }
\problem {
        x < y
    -> \<{
            int t = x;
            x=y;
            y=t;
        }\> y < x
}
```

———————————————————— KeY Problem File ——

When comparing this syntax with the notation used in (10.1), we see that diamond modality brackets "⟨" and "⟩" are written as "`\<{`" and "`}\>`" within the KeY system. (In future versions, "`{`" and "`}`" might become obsolete here, such that "`\<`" and "`\>`" would suffice.) What we can also observe from the file is that all program variables which are *not* declared in the JAVA code inside the modality (like "`t`" here) must appear within a `\programVariables` declaration of the file (like "`x`" and "`y`" here).

Instead of loading this file, and proving the problem, we try out other examples first, which are meant to slowly introduce the principles of proving JAVA CARD DL formulae with KeY.

Using the Prover as an Interpreter

We consider the file `executeByProving.key`:

---

[16] Here as in all `.key` files, line breaks and indentation do not matter other than supporting readability.

––––– KeY Problem File –––

```
\predicates { p(int,int); }
\programVariables { int i, j; }
\problem {
        \<{ i=2;
            j=(i=i+1)+4;
          }\> p(i,j)
}
```

–– KeY Problem File –––––

As the reader might guess, the `\problem` formula is not valid, as there are no assumptions made about the predicate `p`. Anyhow, we let the system *try* to prove this formula. By doing so, we will see that the KeY prover will essentially *execute* our (rather obscure) program "`i=2; j=(i=i+1)+4;`", which is possible because all values the program deals with are *concrete*. The execution of Java programs is of course not the purpose of the KeY prover, but it serves us here as a first step towards the method for handling symbolic values, *symbolic execution*, to be discussed later.

We load the file `executeByProving.key` into the system. Then, we run the automated Java Card DL strategy (by clicking the play button ▶ with the **Java DL** strategy selected in the **Proof Search Strategy** tab). The strategy stops with `==> p(3,7)` being the (only) OPEN GOAL, see also the **Proof** tab. This means that the proof *could* be closed *if* `p(3,7)` was provable, which it is not. But that is fine, because all we wanted is letting the KeY system compute the values of `i` and `j` after execution of "`i=2; j=(i=i+1)+4;`". And indeed, the fact that proving `p(3,7)` would be sufficient to prove the original formula tells us that that 3 and 7 are the final values of `i` and `j`.

We now want to inspect the (unfinished) proof itself. For this, we select the first inner node, labelled with number "1:", which contains the original sequent. By using the down-arrow key, we can scroll down the proof. The reader is encouraged to do so, before reading on, all the way down to the OPEN GOAL, to get an impression on how the calculus executes the Java statements at hand. This way, one can observe that one of the main principles in building a proof for a DL formula is to perform *program transformation* within the modality(s). In the current example, the complex second assignment `j=(i=i+1)+4;` was transformed into a sequence of simpler assignments. Once a leading assignment is simple enough, it moves out from the modality, into other parts of the formula (see below). This process continues until the modality is empty ("`\<{}\>`"). That empty modality gets eventually removed by the taclet **emptyModality**.

Discovering Updates

Our next observation is that the formulae which appear in inner nodes of this proof contain a syntactical element which is not yet covered by the

above explanations of DL. We see that already in the second inner node (number "2:"), which *in the current implementation* looks like:

```
─── KeY Output ───
==>
 {i:=2}
   \<{
       j=(i=i+1)+4;
     }\> p(i,j)
                                    ─── KeY Output ───
```

The "i:=2" within the curly brackets is an example of what is called "*up-dates*". When scrolling down the proof, we can see that leading assignments turn into updates when they move out from the modality. The updates somehow accumulate, and are simplified, in front of a "shrinking" modality. Finally, they get applied to the remaining formula once the modality is gone.

Understanding Updates (to a Certain Extent)

Updates are part of the JAVA CARD DL invented within the KeY project. Their main intention is to represent the *effect* of some JAVA code they replace. This effect can be accumulated, manipulated, simplified, and applied to other parts of the formula, in a way which is (to a certain extent) disentangled from the manipulation of the program in the modality. This allows a separation of concerns which has been fruitful for the design and usage of the calculus and the automated strategies.[17]

Elementary updates in essence are a restricted kind of JAVA assignment, where the right-hand side must be a *simple* expression, which in particular is *free of side effects*. Examples are "i:=2", or "i:=i + 1" (which we find further down in the proof). From elementary updates, more complex updates can be constructed (see Def. 3.8, Chap. 3). Here, we only mention the most important kind of compound updates, *parallel updates*, an example of which is "i:=3 || j:=7" further down in the proof.

Updates extend traditional DL in the following way: if φ is a DL formula and u is an update, then $\{u\}\varphi$ is also a DL formula. Note that this definition is recursive, such that φ in turn may have the form $\{u'\}\varphi'$, in which case the whole formula looks like $\{u\}\{u'\}\varphi'$. (The strategies try to transform such subsequent updates into one, parallel update.) As a special case, φ may not contain any modality (i.e., it is purely first-order). This situation occurs in the current proof in form of the sequent ==> {i:=3 || j:=7}p(i,j) (close to the OPEN GOAL). Once the modality is gone, the update is *applied*, in the form of a *substitution*, to the (now only first-order) formula following the update, as the reader can see when scrolling the proof. Altogether, this leads

---

[17] In the presence of pointers, like the object references in JAVA, the concept of updates serves as an alternative to having and explicit heap *as data* in the logic.

to a delayed turning of program assignments in into substitutions in the logic, as compared to other variants of DL (or of Hoare logic).

In this sense, we can say that updates are *lazily* applied. On the other hand, they are *eagerly* simplified, as we will see in the following (intentionally primitive) example. For that, we load the file `updates.key`. Then, the initial **Current Goal** looks like this:

```
────── KeY Output ──────────────────────────────────────────
==>
 \<{
     i=1;
     j=3;
     i=2;
   }\> i = 2
──────────────────────────────────────── KeY Output ──────
```

We prove this sequent interactively (twice even), just to get a better understanding of the basic steps usually performed by automated strategies. In the first round, we focus on the role of updates in the proof, whereas the discussion of the used taclets is postponed to the second round (see below).

The first assignment, `i=1;`, is simple enough to be moved out from the modality, into an update. We can perform this step by pointing on that assignment, and applying the **assignment** rule. In the resulting sequent, that assignment got removed and the update `{i:=1}`[18] appeared in front. We perform the same step on the leading assignment `j=3;`. Afterwards, and surprisingly, the **Current Goal** does *not* contain the corresponding update, `{j:=3}`. But a closer look on the **Proof** pane shows that the prover actually performed two steps. After the first, we actually had the two subsequent updates `{i:=1}{j:=3}` (as we can see when selecting the corresponding inner node). However, on this goal the prover called the built in *update simplifier*, automatically. That update simplifier is a very powerful proof rule, and one of the few rules which are not represented by a taclet. In this case, the update simplifier detected that the update `{j:=3}` is irrelevant for the validity of the sequent, and simplified it away.

We continue by calling the **assignment** rule a third time (which requires that the OPEN GOAL is selected again). When looking at the resulting **Current Goal**, we note that indeed the assignment `i=2;` turned into the update `{i:=2}`, but this time, the older update `{i:=1}` got lost. The reason is again that the prover *eagerly* applies the update simplifier, which this time turned the two updates `{i:=1}{i:=2}` (see the corresponding inner node) into `{i:=2}` only.

With the empty modality highlighted in the OPEN GOAL, we can apply the rule **emptyModality**. It deletes that modality, and results in the sequent

─────────────────────────

[18] Strictly speaking, the curly brackets are not part of the update, but rather surround it. It is however handy to ignore this syntactic subtlety when discussing examples.

==> {i:=2}(i = 2)[19]. However, we cannot immediately see that sequent, because again the update simplifier resolved that goal, by applying the update *as a substitution*. We refrain from finishing this proof interactively, and just press the play button instead.

Before moving on, we note that the examples which are discussed here are not intended (and not sufficient) for justifying the presence of updates in the logic really. Such a discussion would certainly exceed the scope of this chapter. We only mention here that one of the major tasks of the update simplifier is the proper handling of object aliasing.

Employing Active Statements

We are going to prove the same problem again, this time focusing on the connection between programs in modalities on the one hand, and taclets on the other hand. For that, we load `updates.key` again. When moving the mouse around over the single formula of the **Current Goal**,

```
\<{
    i=1;
    j=3;
    i=2;
  }\> i = 2
```

we realise that, whenever the mouse points anywhere between (and including) "\<{" and "}\>", the whole formula gets highlighted. However, the first statement is highlighted in a particular way, with a different colour, regardless of which statement we point to. This indicates that the system considers the first statement `i=1;` as the *active statement* of this DL formula.

Active statements are a central concept of the DL calculus used in KeY. They control the application/applicability of taclets. Also, all rules which modify the program inside of modalities operate on the active statement, by rewriting or removing it. Intuitively, the active statement stands for the statement next to be executed. In the current example, this simply translates to the *first* statement.

We click anywhere within the modality, and *pre*select (only) the taclet **assignment**, just to view the actual taclet presented in the tooltip:

```
─── Tooltip ───────────────────────────────
assignment {
  \find (
    \modality{#normalassign}{ ..
                          #loc=#se;
                          ... }\endmodality post
  )
```

───

[19] Note that `i = 2` here is a formula, not a JAVA assignment. An assignment would end with a ";", and could only appear within a modality.

```
  \replacewith (
    {#loc:=#se}
      \modality{#normalassign}{ ..  ... }\endmodality post
  )
  \heuristics ( simplify_prog_subset, simplify_prog )
}
```
——————————————————————————————————— Tooltip ——

The `\find` clause tells us how this taclet matches the formula at hand. First
of all, the formula must contain a modality followed by a (not further con-
strained) formula `post`. Then, the first argument of `\modality` tells which
kinds of modalities can be matched by this taclets. (We ignore that argument
here, mentioning just that the standard modality $\langle.\rangle$. is covered.) And finally,
the second argument of `\modality`, ".. #loc=#se; ..." specifies the code
which this taclet matches on. The convention is that everything between ".."
and "..." matches the *active statement*. Here, the active statement must
have the form "#loc=#se;", i.e., a statement assigning a `simple expression`
to a `location`, here `i=1;`. The "..." refers to the rest of the program (here
`j=3;i=2;`), and the match of ".." is empty, in this particular example. Hav-
ing understood the `\find` part, the `\replacewith` part tells us that the active
statement moves out into an update.

After applying the taclet, we point to the active statement `j=3;`, and
again preselect the **assignment**. The taclet in the tooltip is the same, but we
note that it matches the highlighted *sub*-formula, below the leading update.
We suggest to finish the proof by pressing the play button.

The reader might wonder why we talk about "active" rather than "first"
statements. The reason is that our calculus is designed in a way such that
block statements never are "active". By "block" we mean both unlabelled
and labelled JAVA blocks, and well as try-catch blocks. If the first statement
inside the modality is a block, then the active statement is the first statement
inside that block, if that is not a block again, and so on. This concept prevents
our logic from being bloated with control information. Instead, the calculus
works inside the blocks, until the whole block can be *resolved* (because it
is either empty or a `break`, resp., `throw` is active). The interested reader is
invited to examine this by loading the file `activeStmt.key`. Afterwards, one
can see that, as a first step in the proof, one can pull out the assignment
`i=0;`, even if that is nested within a labelled block and a try-catch block. We
suggest to perform this first step interactively, and prove the resulting goal
automatically, for inspecting the proof afterwards.

Now we are able to round up the explanation of the ".." and "..."
notation used in DL taclets. The ".." matches the opening of leading blocks,
up to the first non-block (i.e., active) statement, whereas "..." matches the
statements following the active statement, plus the corresponding closings of
the opened blocks.[20]

---
[20] ".." and "..." correspond to π and ω, respectively, in the rules in Chap. 3.

Executing Programs Symbolically

So far, all DL examples we have been trying the prover on in this chapter had in common that they worked with concrete values. This is very untypical, but served the purpose of focusing on certain aspects of the logic and calculus. However, it is time to apply the prover on problems where (some of) the values are either completely unknown, or only constrained by formulae typically having many solutions. After all, it is the ability of handling symbolic values which makes theorem proving more powerful than testing. It allows to verify a program with respect to *all* legitimate input values!

First, we load the problem `symbolicExecution.key`:

─── KeY Problem File ───

```
\predicates { p(int,int); }
\functions { int a; }
\programVariables { int i, j; }
\problem {
      {i:=a}
         \<{
             j=(i=i+1)+3;
           }\> p(i,j)
}
```

──────────────────────── KeY Problem File ───

This problem is a variation of `executeByProving.key` (see above), the difference being that the initial value of "`i`" is *symbolic*. The "`a`" is a logical *constant* (i.e., a function without arguments), and thereby represents an unknown, but fixed value in the range of `int`. The update `{i:=a}` is necessary because it would be illegal to have an assignment `i=a;` inside the modality, as "`a`" is not an element of the JAVA language, not even a program variable. This is another important purpose of updates in our logic: to serve as an interface between logical terms and program variables.

The problem is of course as unprovable as `executeByProving.key`. All we want this time is to let the prover compute the symbolic values of `i` and `j`, with respect to `a`. We get those by running the **Java DL** strategy on this problem, which results in `==> p(1+a,4+a)` being the remaining OPEN GOAL. This tells us that `1+a` and `4+a` are the final values of `i` and `j`, respectively. By further inspecting the proof, we can see how the strategy performed symbolic computation (in a way which is typically very different from interactive proof construction). That intertwined with the "execution by proving" method discussed above forms the principle of *symbolic execution*, which lies at the heart of the KeY prover.

Another example for this style of formulae is the `\problem` which we load from `postIncrement.key`:

───── KeY Problem File ─────

```
\functions { int a; }
\programVariables { int i; }
\problem {
        {i:=a}
           \<{
                i=i*(i++);
              }\> a * a = i
}
```

───────────────────── KeY Problem File ─────

Depending on the reader's understanding of JAVA, the validity of this formula is not completely obvious. But indeed, the obscure assignment `i=i*(i++);` computes the square of the original value of `i`. The point is the exact evaluation order within the assignment at hand. It is of course crucial that the calculus allows to, by symbolic execution, emulate the evaluation order exactly as it is specified in the JAVA language description, *and* that the calculus does not allow any other evaluation order.

We prove this formula automatically and, as always, suggest that the reader scrolls through the proof afterwards, not to check all details, but to get an impression on how KeY symbolically executes the program.

Quantifying over Values of Program Variables

A DL formula of the form $\langle p \rangle \varphi$, possibly preceded by updates, like $\{u\}\langle p \rangle \varphi$, can well be a sub-formula of a more complex DL formula. One example is the form $\psi \Rightarrow \langle p \rangle \varphi$, where the diamond formula is below an implication (see, for instance, formula (10.1)). A DL sub-formula can actually appear below arbitrary logical connectives, *including quantifiers*. The following problem formula from `quantifyProgVals.key` is an example for that.

───── KeY Problem File ─────

```
\programVariables { int i; }
\problem {
        \forall int x;
          {i := x}
            \<{
                i = i*(i++);
              }\> x * x = i
}
```

───────────────────── KeY Problem File ─────

Note that it would be illegal to have an assignment `i=x;` inside the modality, as "x" is not an element of the JAVA language, but a *logical* variable instead.

This formula literally says that, `\forall` initial values `i`, it holds that after the assignment `i` contains the square of that value. Intuitively, this

seems to be no different from stating the same for an *arbitrary but fixed* initial value "a", as we did in `postIncrement.key` above. And indeed, if we load `quantifyProgVals.key`, and as a first step apply the taclet **allRight**, then the **Current Goal** looks like this:

```
─── KeY Output ───────────────────────────────────
==>
 {i:=x_0}
   \<{
       i=i*(i++);
     }\> x_0 * x_0 = i
─────────────────────────────────── KeY Output ───
```

Note that `x_0` cannot be a logical variable (as was `x` in the previous sequent), because it is not bound by a quantifier. Instead, `x_0` is a *Skolem constant* (cf. the earlier discussion of Skolem terms).

We see here that, after only one proof step, the sequent is essentially no different from the initial sequent of `postIncrement.key`. This seems to indicate that quantification over values of program variables is not necessary. That might be true here, but is not the case in general! The important proof principle of *induction* applies to quantified formulae, only! Using KeY for inductive proving is so important that there is a separate chapter ($\Rightarrow$ Chap. 11) reserved for that issue.

Proving DL Problems with Program Variables

So far, most DL `\problem` formulae *explicitly* talked about *values*, either concrete ones (like "2") or symbolic ones (like the logical constant "a" and the logical variable "x"). It is however also common to have DL formulae which do not talk about any (concrete or symbolic) values explicitly, but instead only talk about *program variables* (and thereby *implicitly* about their values). As an example, we use yet another variation of the post increment problem, contained in `postIncrNoUpdate.key`:

```
─── KeY Problem File ─────────────────────────────
\programVariables { int i, j; }
\problem {
        \<{
            j=i;
            i=i*(i++);
          }\> j * j = i
}
─────────────────────────────── KeY Problem File ───
```

Here, instead of initially updating `i` with some symbolic value, we store the value of `i` into some other program variable. The equation after the modality then is a claim about the relation between (the implicit values of) the

program variables, in a state after program execution. When proving this formula automatically with KeY, we see that the proof has no real surprise as compared to the other variants of post increment. Please observe, however, that the entire proof does not make use of any symbolic value, and only talks about program variables, some of which are introduced within the proof.

In this context, it is very natural to come back to the formula

$$x < y \rightarrow \langle \texttt{int t = x; x = y; y = t;} \rangle \ y < x$$

which we discussed in the beginning of this section (Sect. 10.2.3). Also this formula only talks about program variables. It assumes the (values of) the variables having a certain relation in the initial state, and states that (the values of) these variables have a different relation after execution of the program.

We load the corresponding problem file, `exchange.key` (which was displayed on page 434) into the system. After proving this problem automatically, we want to point the reader to one interesting detail. When scrolling down this proof, we see the usual course of symbolic execution: programs are transformed into one another, simple assignments turn into updates, and updates are simplified. We stop at the inner node where the modality is already gone, and the last remaining update is *about to* disappear for the rest of the proof (by being applied as a substitution). *Currently* this inner node looks like `y >= 1 + x ==> {x:=y, y:=x}(y <= -1 + x)`. In contrast to previous examples, here it really matters that the update `{x:=y, y:=x}` is a *parallel* one. The variables `x` and `y` switch their values at once, and no auxiliary variable is needed at this point.

Calling Methods in Proofs

Even though the DL problem formulae discussed so far all contained real Java code, we did not see either of the following central Java features: classes, objects, or method calls. The following small example features all of them.

We consider the file `methodCall.key`:

──── KeY Problem File ────────────────────

```
\javaSource "methodExample/"; // location of class definitions
\programVariables { Person p; }
\problem {
        \forall int x;
          {p.age:=x} // assign initial value to "age"
            (   x >= 0
             -> \<{
                    p.birthday();
                  }\> p.age > x)
}
```

────────────────────────── KeY Problem File ────

The \javaSource declaration tells the prover where to look up the sources of classes (and interfaces) used in the file. In particular, the JAVA source file Person.java is contained in the directory methodExample/. The problem formula states that a Person p is getting older at its birthday(). (On the side, the reader may note that the update here does not immediately precede a modality, but a more general DL formula.)

Before loading this problem file, we look at the source file Person.java in methodExample/:

─── JAVA ───
```
public class Person {
    private int age = 0;
    public void setAge(int newAge) { this.age = newAge; }
    public void birthday() {
        if (age >= 0) age++;
    }
}
```
─── JAVA ───

The reader is encouraged to reflect on the validity of the above problem formula a little, before reading on.—Ready?—Luckily, we have a prover at hand to be certain. We load methodCall.key into KeY and, without hesitation, press the play button (assuming that **Java DL** is the selected strategy).

The strategy stops with the OPEN GOAL "p = null, x_0 >= 0 ==>" left.[21] There are different ways to read this goal, which however are logically equivalent. One way of proving any sequent is to show that its left-hand side is false. Here, it would be sufficient to show that p = null is false. An alternative viewpoint is the following: in a sequent calculus, we always get a logically equivalent sequent by throwing any formula to the respective other side, but negated. Therefore, we can as well read the OPEN GOAL as if it was "x_0 >= 0 ==> p != null". Then, it would be sufficient to show that p != null is true.

Whichever reading we choose, we cannot prove the sequent, because we have no knowledge whatsoever about p being null or not. When looking back to our problem formula, we see that indeed the formula is not valid, because the case where p is null was forgotten. The postcondition p.age > x depends on the method body of birthday() being executed, which it cannot in case p is null. We can even read this off from the structure of the uncompleted proof in the **Proof** pane. When tracing the branch of the OPEN GOAL, back to the first split, we can see that the proof failed in the branch marked as "Null Reference (p = null)". It was the taclet **methodCall** which triggered this split.

---

[21] If not, please select **nullCheck** as the **nullPointerPolicy** (see page) and load methodCall.key again.

The file `methodCall2.key` contains the patch of the problem formula. The problem formula from above is preceded by "`p != null ->`". We load that problem, and let KeY prove it automatically without problems. In this proof, we want to have a closer look on the way KeY handles method calls. Like in the previous proof, the first split was triggered by the taclet **methodCall**. Then, in the branch marked as "Normal Execution (p != null)", the second inner node (after some update simplification) looks like this:

────── KeY Output (10.1) ──

```
 x_0 >= 0
==>
 p = null,
 {p.age:=x_0}
   \<{
      p.birthday();
    }\> x_0 <= -1 + p.age
```
── KeY Output ──

We should not let confuse ourselves by `p = null` being present here. Recall that the comma on the right-hand side of a sequent essentially is a logical *or*. Also, as stated above, we can always imagine a formula being thrown to the other side of the sequent, but negated. Therefore, we essentially have `p != null` as an *assumption* here. Another thing to comment on is the `@Person` notation in the method call. It represents that the calculus has decided which *implementation* of `birthday` is to be chosen (which, in the presence of inheritance and hiding, can be less trivial than here).

At this point, the strategy was ready to apply **methodBodyExpand**. After that, the code inside the modality looks like this:

```
method-frame(source=Person,this=p): {
  if (age>=0) {
    age++;
  }
}
```

This `method-frame` is the only really substantial extension over JAVA which our logic allows inside modalities. It models the execution stack, and can appear nested in case of nested method calls. Apart from the class and the `this` reference, it can also specify a return variable, in case of non-`void` methods. However, the user is rarely concerned with this construction, and if so, only passively. We will not discuss this construct further here, but refer to Chap. 3, Sect. 3.6.5 instead. One interesting thing to note here, however, is that method frames are considered as *block statements* in the sense of our earlier discussion of active statements, meaning that *method frames are never active*. For our sequent at hand, this means that the active statement of the discussed formula is `if (age>=0) {age++;}`, as one can also see from the taclet which was applied next.

Controlling Strategy Settings

The expansion of methods is among the more problematic steps in program verification (together with the handling of loops). In place of recursion, an automated proof strategy working with method expansion might not even terminate. Another issue is that method expansion goes against the principle of *modular* verification, without which even mid-size examples become infeasible to verify. These are good reasons for giving the user more control over this crucial proof step.

KeY therefore allows to configure the automated strategies in a way that they *refrain* from expanding methods automatically.[22] We try this out by loading `methodCall2.key` again, and selecting **None** as the **Method treatment** option in the **Proof Search Strategy** tab. Then we start the strategy, which now stops exactly at the sequent which we discussed earlier (Sect. 10.1). We can highlight the active statement, and *could* call **methodBodyExpand** interactively. KeY would then *only* apply this very taclet, and stop again. Therefore, we first check the **Autoresume Strategy** checkbox, and then apply **methodBodyExpand**. The strategy will resume automatically, and close the proof.

Controlling Taclet Options

The proof of `methodCall2.key` has a branch for the `null` case ("Null Reference (p=null)"), but that was closed after a few steps, as `p = null` is already present, explicitly, on the right side of the sequent (**closeGoal**). It is, however, untypical that absence of null references can be derived so easily. Often, the "null branches" complicate proofs substantially. The KeY system allows to use a variant of the calculus which ignores the problem of null references. This is actually only one of the issues which are addressed by *taclet options* (see Sect. 4.4.2).

We open the taclet option dialogue, via **Options → Taclet options defaults**. Among the option categories, we select the **nullPointerPolicy**, observe that **nullCheck** is chosen as default, and change that by selecting **noNullCheck** instead. Even if the effect of this change on our very example is modest, we try it out, to see what happens in principle. We again load `methodCall2.key`, press play, and observe that indeed the finished proof has only one branch.

One has to be aware that this change has a dramatic consequence: it affects the soundness of the calculus. To demonstrate this, we load the original problem formula from `methodCall.key` again. By running the automated strategy with the current taclet options, we can see that the system now is able to prove the non-valid formula! As a consequence, one should only switch off the proper null handling if one is, for whatever reason, not interested in problems that originate from null references. Another scenario is that one first tries to prove a problem under the simplifying assumption of no null references being present, which allows to focus attention to other complications of the

---

[22] For a discussion of loop treatment, please refer to Chaps. 11 and 3.

problem at hand. Thereafter, one can re-prove the problem *with* null check again, with help of the re-use facility (see Chap. 13).

Integer Semantics Options

We briefly mention another very important taclet option, the **intRules**. Here, the user can choose between different semantics of the primitive JAVA integer types `byte`, `short`, `int`, `long`, and `char`. Options are: the mathematical integers (easy to use, but not fully sound), mathematical integers with overflow check (sound, reasonably easy to use, but unable to verify programs which depend on JAVA's modulo semantics), and the true modulo semantics of JAVA integers (sound, complete, but difficult to use). This book contains a full chapter on JAVA Integers (Chap. 12), discussing the different variants in the semantics and the calculus. Fig. 12.1 displays the corresponding GUI dialogue. Please note that KeY 1.0 comes with the mathematical integer semantics chosen as default option, to optimise usability for beginners. However, for a sound treatment of integers, the user should switch to either of the other semantics. As an alternative, we suggest to use the *proof reuse* feature of KeY (see Chap. 13). One can first construct a proof using the mathematical integer option, and then replay it with the mathematical overflow semantics selected.

10.3 Generating Proof Obligations

We have so far applied KeY on several examples which were meant to demonstrate the most essential features of the logic, the calculus, the prover, and, in particular, the usage of the prover. All those examples had in common that the proof obligations were hand crafted, and stored in `.key` files.

However, even if the logical framework and the prover technology forms an essential part of the KeY project, the whole KeY approach to formal methods is not all about that. Instead, it is very much about the *integration* of verification technology into more conventional software development methods, as was outlined in the introductory chapter of this book (Chap. 1). Sect. 1.1 gave an overview on how we use modern *object-oriented modelling* approaches as *hooks* for formal verification. In particular, KeY so far employs two modelling/specification languages: UML's Object Constraint Language (OCL) and the Java Modeling Language (JML). These languages, their usage and their theory, are described in Chap. 5 in this book.

KeY interfaces with OCL as well as JML, by translating them (and the specified JAVA code) into *proof obligations* in JAVA CARD DL. This issue, and the rich theory behind it, is described in Chap. 8.

But not only does KeY interface with certain standard specification languages. It also interfaces with *standard tools* for software development, currently the commercial CASE tool Borland Together, and the freely available IDE Eclipse. An overview over the architectural setup of this integration

was given in Fig. 1.1 (Chap. 1). Following that figure from the right to the left, we have essentially four scenarios, varying in the origin of proof obligations (POs):

1. *Hand-crafted* POs, to be loaded from `.key` files.
2. *Automatically generated* POs
   a) from JML-augmented JAVA source files, using
      i. the JML browser of the KeY stand-alone system.
      ii. Eclipse with the KeY plug-in.
   b) from OCL-augmented UML diagrams and JAVA source files, using Borland Together with KeY extensions.

Scenario 1 has been practised in the course of the previous section. Below, we focus on Scenario 2.

Using the JML Specification Browser of the KeY Stand-Alone System

JAVA classes and their methods are specified in JML using *class invariants* and *method contracts*.[23] It is part of the concept of JML that specifications are included in JAVA source code, in the form of particular *comments*. If we want to verify that JML specifications are respected by the corresponding implementations, we can let the KeY system generate corresponding proof obligations (in JAVA CARD DL) from these JML comments. The KeY stand-alone system supports this by offering a *JML specification browser*.

In the directory `Bank-JML`, the reader finds the JAVA sources of a banking scenario[24]. Using an editor, we can see that the `.java` files in `Bank-JML` indeed contain JML specifications. We focus on the file `ATM.java`, and therein on the contract of the method `enterPIN` (textually located in the comment preceding the method). This contract is also displayed in Fig. 5.14, Sect. 5.3.1 (Chap. 5). The same section contains a detailed explanation of this very JML contract!

A JML contract can be composed from several more elementary contracts, connected by the keyword "`also`". The contract of `enterPIN` is composed from three such parts, the last of which specifies the case where a wrong PIN has been entered too often:

—— JML (10.2) ————————————————

```
public normal_behavior
requires  insertedCard != null;
requires  !customerAuthenticated;
requires  pin != insertedCard.correctPIN;
requires  wrongPINCounter >= 2;
assignable insertedCard, wrongPINCounter,
           insertedCard.invalid;
```

---

[23] Method contracts are referred to as "operation contracts" in Chap. 5.
[24] This scenario is used in a course at Chalmers University, see Chap. 1.

```
ensures   insertedCard == null;
ensures   \old(insertedCard).invalid;
ensures   !customerAuthenticated;
```
———————————————————————————————————— JML ——

The displayed part of the JML contract gives rise to two POs, to be generated by the Java Card DL translation of KeY: one PO for verifying the "`assignable`" conditions, and one PO for verifying the "`ensures`" conditions. The latter PO we want to generate, and prove, with the KeY system.

First of all, we activate the JML browser on the directory `Bank-JML`, by *loading the entire directory, containing Java+JML sources* into the system. This is done in the same manner as loading problem files and proofs **by File → Load ...** *or clicking at* 📁 *in the tool bar*. It is important that we have the *directory*, here `Bank-JML`, selected when pressing the **Open** button (not any of the contained files).

The system now analyses the Java+JML sources in `Bank-JML`, and opens the **JML Specification Browser** window. In its **Classes** pane, the available classes are grouped after the packages they belong to. The application classes of our scenario all belong to the package `bank`, so we make sure that folder is expanded, and select the class ATM. The **Methods** pane shows the methods of class `ATM`. After selecting **enterPIN**, the **Proof Obligations** pane allows choosing a PO connected to that method, to be loaded into the system. We ignore the `Assignable` POs for now, and among the three others choose the one which corresponds to the piece of JML quoted above (10.2), and press **Load Proof Obligation**.

We find ourselves in a familiar situation: a new proof task is loaded into the system, and the initial sequent is presented in the **Current Goal** pane. The sequent looks very substantial. How this PO was constructed cannot be discussed here in detail. (We refer to Chap. 8.) Still, we comment a bit on the overall structure of this PO, with the intention to demystify its lengthy appearance.

The PO is an implication, with "`inReachableState`" acting as basic condition under which the rest of the PO must be true. The predicate `inReachableState` restricts the states to those reachable by *any* Java computation. For instance, `inReachableState` implies that all referenced (non-null) objects are actually created.

The remaining PO starts with some quantifiers and updates. Thereafter, we have an implication basically saying: "the (translated) `requires` part, together with the (translated) class invariant, implies that the (translated) `ensures` part holds after the method". Note that it is the translated class invariant which makes the PO so long. That however is not a burden from the proving perspective. To the contrary: being on the left side of the implication, the invariant only provides additional assumptions that may, or may not, be used for establishing the right-hand side.

By simply pressing the play button, we make KeY proving this PO automatically.
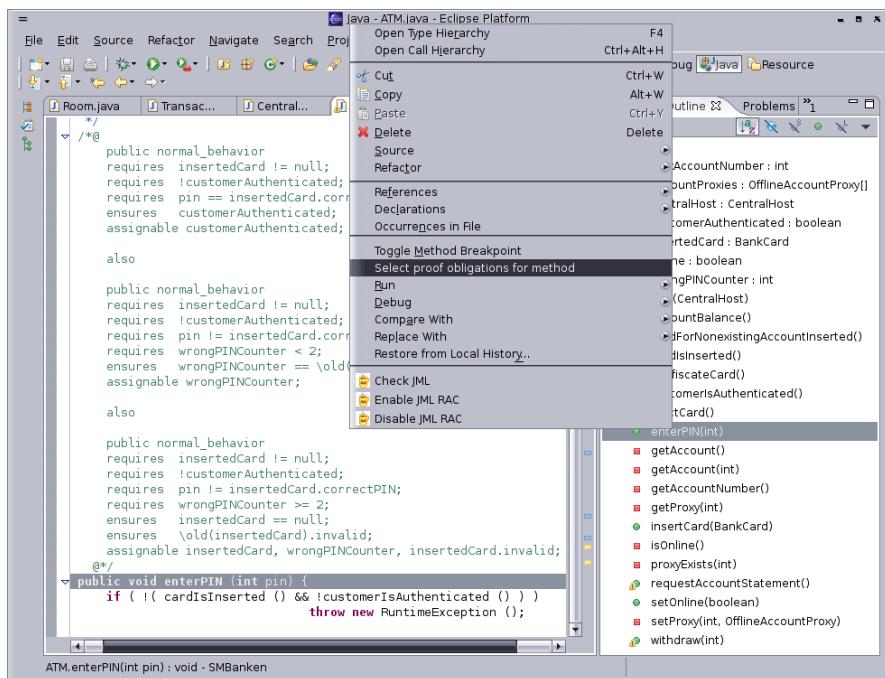
Fig. 10.1. KeY-Eclipse integration

Using Eclipse with KeY Plug-in

Contemporary software development makes more and more use of tools which integrate the different activities around the development of programs. One such tool is the freely available IDE Eclipse[25], which currently is the most widely distributed IDE for Java. It provides powerful coding support, like code templates, code completion, and import management. Eclipse also features a well documented plug-in interface.

KeY also comes as one such Eclipse plug-in. When developing Java+JML code within Eclipse, the usual context menus offer the additional functionality of selecting proof obligations, as indicated in Fig. 10.1 (for our `ATM.enterPIN` example). The KeY plug-in will start up automatically, generate the selected proof obligation, and present it in the prover window, ready for automated, resp. interactive proving. We refer to documentation and tutorials, available from the KeY project's web page for updated information about how to install—and use—a KeY-equipped Eclipse platform as a front end in the verification of JML specified Java programs.

---

[25] `www.eclipse.org`

Using Borland Together with KeY Extensions

CASE tools go beyond IDEs in their integration of software *modelling* activities, which normally includes support of (various aspects of) UML. The KeY project propagates the use of formal methods early in the software process, and therefore makes a serious attempt to integrate facilities for specification and verification into *tools* of that kind. Such an integration has been exemplified by augmenting the commercial CASE tool Borland Together with KeY extensions.

In this context, the hook for formal methods consists of UML/OCL. In the KeY-extended Borland Together, UML class diagrams can be decorated with OCL constraints. The creation of such constraints is supported a) by parsing, b) by KeY OCL idioms and KeY OCL patterns, with an corresponding pattern instantiation mechanism (see Chap. 6), and c) by a structural, multilingual editor, for simultaneous editing, and cross translation, of constraints in OCL respectively natural language (English, German) (see Chap. 7).

As for the verification side, context menus allow to, for instance, choose proof obligations directly from the class diagram view of a project. Also here, KeY will generate the chosen proof obligation, and start up the prover, ready to prove the goal at hand. We again refer to the KeY project's web page for the version-sensitive information of how to install—and use—the KeY extensions on top of Borland Together.

11

Proving by Induction

by
Angela Wallenburg

11.1 Introduction

In this chapter we describe how the concept of mathematical induction can be used in a practical way to prove program correctness. To complete an inductive proof the user needs to guide the theorem prover in KeY along the way. The knowledge about the fundamentals of induction that is required for this will be introduced. Since this chapter is written in tutorial style it is a good idea to work out the examples in the KeY prover in parallel to reading. It can also be read as a general introduction to induction in program verification.

11.2 The Need for Induction

Mathematical induction[1] is a proof method that can be used to prove properties about infinite (or very large) data types. In program verification, we want to reason about many of the abstract data objects that are commonly used by programmers: integers, lists, trees etc; as well as about properties of iterative programs. For this, induction is an essential tool.

For instance, consider the scenario that you want to prove a loop totally correct—that it terminates and fulfills the specified postcondition—and the number of loop iterations is unknown or very large. How many times should you apply the loopUnwind rule (3.6.4) to unwind the loop body?

This chapter is devoted to explaining the basic concepts of (mathematical) induction, how these are implemented in KeY, some general difficulties with inductive theorem proving and how to interact with the KeY prover in the most fruitful way to construct induction proofs. We consider programs that terminate normally. For abrupt termination, see [Beckert and Sasse, 2001]. Further limitations are discussed in Sect. 11.9. Let us start by taking a first look at the most common induction rule and show it at work on a small example in first-order logic ($\Rightarrow$ Chap. 2).

---

[1] As opposed to *philosophical induction*.

11.2.1 A First Look at an Induction Rule

Let us start with the first-order Peano induction rule in its simplest form. If you have encountered induction before, it is likely that you have seen a rule similar to this one:

$$\text{peanoInduct} \quad \frac{\Gamma \Longrightarrow \phi(0),\ \Delta \qquad \Gamma \Longrightarrow \forall n.(\phi(n) \rightarrow \phi(n+1)),\ \Delta}{\Gamma \Longrightarrow \forall n.\phi(n),\ \Delta}$$

Here (and in the following) we use the convention that $n{:}\mathbb{N}$. Induction is performed over the induction variable n and $\phi(n)$ is the induction formula. An application of this rule results in two branches—a base case and a step case. The idea of induction is the following: In the base case we show that the induction formula holds for $n \doteq 0$, and in the step case we show that if we assume that the induction formula holds for an arbitrary n, then it holds for $n+1$. By the principle of induction, we can then conclude that the formula holds for all natural numbers.

It is common to either introduce a lemma and prove that using induction, or to apply the cut rule to make use of the result.

$$\text{cut} \quad \frac{\Gamma \Longrightarrow \psi,\ \Delta \qquad \Gamma,\ \psi \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

In order to avoid an extra cut, and simply to facilitate direct usage of induction (almost) anywhere along the way of a proof attempt, the KeY prover has an induction rule that combines the above rules into one:

$$\text{natInduct} \quad \frac{\begin{array}{l} \Gamma \Longrightarrow \phi(0),\ \Delta \\ \Gamma \Longrightarrow \forall n.(\phi(n) \rightarrow \phi(n+1)),\ \Delta \\ \Gamma,\ \forall n.\phi(n) \Longrightarrow \Delta \end{array}}{\Gamma \Longrightarrow \Delta}$$

This 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 (Δ in the rule above). It gives three branches: 1) *base case*, 2) *step case*, and 3) *use case*. This rule represents a simple form of induction, but it is still very powerful, as we see in the next example.

11.2.2 A Small Example

In this section we show the entire proof of a tiny example in first order logic. Consider the predicate *even* that has the following definition:

$$even(0) \tag{11.1}$$

$$\forall x.(even(x) \rightarrow even(x+2)) \tag{11.2}$$

Now the task is to prove that 2 times 7 is even. For proving this, we have two options. One way is to apply the definition 7 times. The other way is to use *induction* to prove that $even(2*n)$ holds for all natural numbers n. Of course, we proceed with the more general choice and show the inductive proof. Please refer to Fig. 11.1 for an overview of the proof. We start with the following proof obligation:

$$even(0), \forall x.(even(x) \rightarrow even(x+2)) \Longrightarrow even(2*7) \ . \qquad \text{(P1)}$$

Then let us apply the simple induction rule natInduct that we have seen, and use $even(2*n)$ as the induction formula. The proof gets divided into three branches:

$$even(0), \ \forall x.(even(x) \rightarrow even(x+2)) \Longrightarrow even(2*0), \ even(2*7) \qquad \text{(P2)}$$

$$even(0), \ \forall x.(even(x) \rightarrow even(x+2)) \Longrightarrow$$
$$\forall n.(even(2*n) \rightarrow even(2*(n+1))), \ even(2*7) \qquad \text{(P3)}$$

$$even(0), \ \forall x.(even(x) \rightarrow even(x+2)), \ \forall n.(even(2*n) \Longrightarrow even(2*7)) \qquad \text{(P4)}$$

These are the base case (P2), the step case (P3) and the use case (P4) respectively. Assuming the existence of basic arithmetic taclets for addition and multiplication, we proceed with each case in turn:

Base Case

By applying the rule for multiplication with 0 in the succedent of (P2) we get:

$$even(0), \ \forall x.(even(x) \rightarrow even(x+2)) \Longrightarrow even(0), \ even(2*7) \qquad \text{(P5)}$$

which is closable and we are done with the base case.

Step Case

To the formula (P3), we apply the rule allRight and get:

$$even(0), \ \forall x.(even(x) \rightarrow even(x+2)) \Longrightarrow$$
$$even(2*n_c) \rightarrow even(2*(n_c+1)), \ even(2*7) \qquad \text{(P6)}$$

Recall ($\Rightarrow$ Chap. 2) that the rule allRight replaces the universally quantified variable with a (Skolem) constant, here n_c, which represents an arbitrary natural number. We move on with the proof by distributing the multiplication:

$$even(0), \ \forall x.(even(x) \rightarrow even(x+2)) \Longrightarrow$$
$$even(2*n_c) \rightarrow even(2*n_c+2), \ even(2*7) \qquad \text{(P7)}$$

Now we make the observation that we would be able to close this branch if we could supply $2 * n_c$ for the universally quantified variable x. We can do this so called *instantiation* by applying the rule allLeft:

$$even(0), even(2 * n_c) \mathbin{-\!>} even(2 * n_c + 2) \Longrightarrow \\ even(2 * n_c) \mathbin{-\!>} even(2 * n_c + 2), even(2 * 7) \tag{P8}$$

and the step proof branch can also be closed. By doing this (and due to the induction principle) we have now proven $\forall n.even(2 * n)$.

Use Case

Now it only remains to make use of the result that we have proven with induction. This we do by instantiation again. Recall formula (P4). We use the rule allLeft to instantiate the universally quantified variable n with 7:

$$even(0), \forall x.(even(x) \mathbin{-\!>} even(x + 2)), even(2 * 7) \Longrightarrow even(2 * 7) \tag{P9}$$

and then we have closed all three branches and we have proved both $even(2*7)$ and $\forall n.even(2*n)$. Below is a proof tree to provide an overview of the proof.
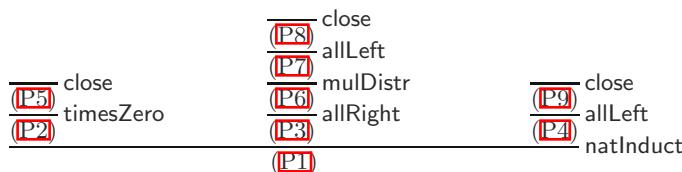
Fig. 11.1. Proof tree for $even(2 * 7)$

11.3 Basics of Induction in KeY

In this section we briefly explain the fundamentals of mathematical induction. At the same time we introduce the notation that we use in the remainder of this chapter. We also describe how induction and its related necessities are implemented in Java Card DL and KeY.

11.3.1 Induction Rule

By now you have seen the simplest induction rule in KeY at work. The rule can be applied both to first-order ($\Rightarrow$ Chap. 2) and Java Card DL ($\Rightarrow$ Chap. 3) formulas. In KeY the deduction rules are implemented as *taclets* ($\Rightarrow$ Chap. 4), and so are the induction rules. Here is the natInduct taclet expressed in .key syntax ($\Rightarrow$ App. B):

─── KeY ───

```
nat_induction {
  "Base␣Case": \add ( ==> {\subst nv; 0}(b) );
  "Step␣Case": \add ( ==> \forall nv; (nv >= 0 & b) ->
                                       {\subst nv; (nv + 1)}b));
  "Use␣Case":  \add ( \forall nv; (nv >= 0 -> b) ==>)
};
```

── KeY ───

11.3.2 Induction Variable

The induction variable is the variable that we perform induction over. Its domain can be for instance some subset of the integers (and this domain gets most of the attention in this chapter) but in general it can be any well-founded set. In the natInduct rule above the induction variable is n.

One important issue to note here is that the induction variable needs to be a *logical variable*. This is due to the fact that we cannot quantify over program variables, or locations, in KeY and JAVA CARD DL. So in order to prove some inductive property about a program, one needs to "connect" the induction variable to the program variable intended in the update (the state description within curly brackets) of the induction formula. A concrete example of how to do this is can be found in Sect. 11.4 below. Also note that all programs in this chapter have mathematical integer semantics, not JAVA integer semantics.

11.3.3 Induction Formula

The *induction formula* represents what we really want to prove with the induction. This formula depends on the induction variable. In the rule natInduct from above the induction formula is called $\phi(n)$. When using induction, we prove that the induction formula holds for all the elements in the domain of the induction variable, for our example $\forall n.\phi(n)$.

The choice of induction formula greatly influences the proof (attempt). Different instances of the same induction formula occur in all the branches of an inductive proof. In the step case, two instances occur: the induction hypothesis and the induction conclusion. The *induction hypothesis* is the assumption of the step case $\phi(n)$ for an arbitrary n. The *induction conclusion*, $\phi(n+1)$, is what needs to be proven under the assumption of the induction hypothesis.

11.3.4 Induction Principle

The *induction principle* is the argument that the induction formula holds for all numbers once the base case and the step case have been proved; it constitutes the main soundness argument.

Theorem 11.1 (Principle of Mathematical Induction). *The truth of an infinite sequence of propositions P_i for $i = 1, \ldots, \infty$ is established if (1) P_1 is true, and (2) P_k implies P_{k+1} for all k.*

This principle is sometimes also known as the method of induction. The rule natInduct, as seen earlier in this chapter, would not be sound without this theorem. Later on we will use more advanced induction rules and it is important to note that any induction rule requires an argument for its soundness similar to the theorem above. This we have to keep in mind in order to avoid introducing unsound induction taclets into KeY. One can say that the induction principle is implemented in the use case of an induction taclet in KeY, because in the use case it is assumed that the induction formula holds for the entire induction set. If we do not have an accompanying induction principle with our induction rule, or if we rely on a flawed induction principle, it is the use case that would be unsound.

11.4 A Simple Program Loop Example

Now it is time for a small but complete program example—a simple loop. Consider the—from a program verification point of view—incomplete specification: "Decrementing a variable as long as the counter is positive sets the variable to zero." Such a decrementor can be implemented as a loop in JAVA:

```
─── Java ───
int i;
// some initialisation code
while (i > 0) {
    i--;
}
                                                          ─── Java ───
```

Let us say that we want to show that this loop terminates and arrives at a state where a certain condition, the postcondition (here $i \doteq 0$), is fulfilled. What do we need to assume about the value of the variable i before the loop in order to be able to prove this? This we have to put in the precondition. So we let our proof obligation be $\forall il.\phi(il)$, where $\phi(il)$ is the following JAVA CARD DL formula:

$$il \geq 0 \rightarrow \{i := il\}$$
$$\langle \text{ while (i > 0) } \{$$
$$\quad i--;$$
$$\} \ \rangle \ (i \doteq 0) \tag{P1}$$

Notice that the only available variable to reason about in our example is `i`, which is a program variable. We cannot quantify over program variables in JAVA CARD DL, but we need somehow to quantify over `i` to show that this program is correct regardless of its value. Thus we have introduced the logical variable il which is connected to `i` in the update, within the curly brackets. Its value is updated as the proof (and symbolic execution) proceeds. So we can quantify over il instead of the program variable and our proof obligation becomes $\forall il.\phi(il)$.

As the rest (apart from the update) of the formula is concerned, it states that the precondition $il \geq 0$ implies total correctness of the loop. It contains the total correctness assertion, that is the diamond brackets $\langle\ \rangle$ with the program code that we want to prove correct (the while-loop). Finally, the postcondition is stated following the diamond brackets.

11.4.1 Preparing the Proof

Now we proceed to describe how to make the proof. KeY is an interactive theorem prover and to construct an induction proof the user has to 1) supply the induction variable, 2) choose which induction rule to use, and 3) supply the induction formula.

For 1), in our example the obvious choice for the induction variable is il since it is the logical variable that corresponds to the only available variable `i`. Regarding 2), the standard Peano induction rule natInduct is good enough for this simple example. Finally, for deciding the induction formula (3), let us make the naive choice and just supply the formula in the proof obligation itself: $\phi(il)$.

11.4.2 The Proof in JAVA CARD DL

With the original proof obligation in (P1) as the induction formula $\phi(il)$ and il as the induction variable, we apply the induction rule natInduct and get a base case, a step case and a use case. Let us proceed with each of the cases in turn. See Fig. 11.2 for an overview of the proof. The formulas in our proofs are relatively large and usually there are only small changes between proof steps. Here and in the following we print the unchanged parts of a formula in gray in order to make it easier to follow the proofs.

Base Case

In the base case, we start with an instantiation with 0 of the induction formula, $\phi(0)$, (P2). Application of concreteImpl simplifies it to (P3).

$\Longrightarrow$
$0 \geq 0 \to$
  $\{\texttt{i} := 0\}$
  $\langle\,$ while (i > 0) {
      i--;
  } $\rangle\,(\texttt{i} \doteq 0)$ (P2)

$\Longrightarrow$
$\{\texttt{i} := 0\}$
$\langle\,$ while (i > 0) {   (P3)
    i--;
} $\rangle\,(\texttt{i} \doteq 0)$

Then we proceed to unwind one iteration of the loop body with the loopUnwind rule, which results in (P4). Since $\texttt{i} \doteq 0$ (see the update) the loop terminates when we apply ifThenElseFalse, and we end up with (P5), which is easily closed.

$\Longrightarrow$
$\{\texttt{i} := 0\}$
$\langle\,$ if (i > 0) {
    i--;
    while (i > 0) {   (P4)
      i--;
    }
} $\rangle\,(\texttt{i} \doteq 0)$

$\Longrightarrow$
$\{\texttt{i} := 0\}\,\langle\{\}\rangle\,(\texttt{i} \doteq 0)$ (P5)

Step Case

In the step case, we start with (P6). First we eliminate the universal quantifier using the allLeft rule, and the bound logical variable il gets replaced by a Skolem constant i_c that is an arbitrary integer. The resulting proof obligation is (P7).

$\Longrightarrow$
$\forall il.(il \geq 0\ \&$
$(il \geq 0 \to$
  $\{\texttt{i} := il\}$
  $\langle\,$ while (i > 0) {
    i--;
  } $\rangle\,(\texttt{i} \doteq 0))$ (P6)
$\to$
$il + 1 \geq 0 \to$
  $\{\texttt{i} := il + 1\}$
  $\langle\,$ while (i > 0) {
    i--;
  } $\rangle\,(\texttt{i} \doteq 0))$

$\Longrightarrow$
$il_c \geq 0\ \&$
$(il_c \geq 0 \to$
  $\{\texttt{i} := il_c\}$
  $\langle\,$ while (i > 0) {
    i--;
  } $\rangle\,(\texttt{i} \doteq 0))$ (P7)
$\to$
$il_c + 1 \geq 0 \to$
  $\{\texttt{i} := il_c + 1\}$
  $\langle\,$ while (i > 0) {
    i--;
  } $\rangle\,(\texttt{i} \doteq 0)$

After automatic processing the standard simplifications (impRight, andLeft, replaceKnownLeft, concreteImpl) have been performed (P8). Notice that in

(P8) we have the induction hypothesis in the antecedent and the induction conclusion in the succedent, just as expected. Then we unwind one iteration of the loop in the induction conclusion using the loopUnwind rule, and (P9) shows the state after that.

$i_c + 1 \geq 0,$
$i_c \geq 0,$
$\{\texttt{i} := i_c\}$
$\langle$ `while (i > 0) {`
    `i--;`
  `}` $\rangle$ $(\texttt{i} \doteq 0)$ (P8)
$\implies$
$\{\texttt{i} := i_c + 1\}$
$\langle$ `while (i > 0) {`
    `i--;`
  `}` $\rangle$ $(\texttt{i} \doteq 0)$

$i_c + 1 \geq 0,$
$i_c \geq 0,$
$\{\texttt{i} := i_c\}$
$\langle$ `while (i > 0) {`
    `i--;`
  `}` $\rangle$ $(\texttt{i} \doteq 0)$
$\implies$ (P9)
$\{\texttt{i} := i_c + 1\}$
$\langle$ `if (i > 0) {`
    `i--;`
    `while (i > 0) {`
      `i--;`
    `}`
  `}` $\rangle$ $(\texttt{i} \doteq 0)$

The loopUnwind rule application is one instance where the calculus mimics symbolic execution. We proceed to symbolically execute the next statement, using the ifEval (introducing the new variable b) and ifElseSplit rules. As the name suggests, an application of ifElseSplit causes the proof tree to branch. We get one branch for $i_c + 1 \leq 0$ (P10) and one for $i_c + 1 > 0$ (P11):

$i_c + 1 \leq 0,$
$i_c + 1 \geq 0,$
$i_c \geq 0,$
$\{\texttt{i} := i_c\}$
$\langle$ `while (i > 0) {`
    `i--;`
  `}` $\rangle$ $(\texttt{i} \doteq 0)$
$\implies$ (P10)
$\{\texttt{i} := i_c + 1\}$
$\langle$ `b = false;`
  `if (b) {`
    `i--;`
    `while (i > 0) {`
      `i--;`
    `}`
  `}` $\rangle$ $(\texttt{i} \doteq 0)$

$i_c + 1 > 0,$
$i_c + 1 \geq 0,$
$i_c \geq 0,$
$\{\texttt{i} := i_c\}$
$\langle$ `while (i > 0) {`
    `i--;`
  `}` $\rangle$ $(\texttt{i} \doteq 0)$
$\implies$ (P11)
$\{\texttt{i} := i_c + 1\}$
$\langle$ `b = true;`
  `if (b) {`
    `i--;`
    `while (i > 0) {`
      `i--;`
    `}`
  `}` $\rangle$ $(\texttt{i} \doteq 0)$

In the antecedent of (P10), the condition $i_c + 1 \leq 0$ contradicts $i_c \geq 0$, and therefore this branch is easily closed. In the other branch (P11) we have to

proceed with symbolic execution in the succedent. The statement `i--;` is evaluated using the assignment rule. Note that the state change can be seen in the update in (P12).

$$
\begin{array}{ll}
\begin{aligned}
& i_c + 1 > 0, \\
& i_c + 1 \geq 0, \\
& i_c \geq 0, \\
& \{i := i_c\} \\
& \langle\, \text{while (i > 0) \{} \\
& \quad \text{i--;} \\
& \quad \text{\} }\rangle\, (i \doteq 0) \\
& \Longrightarrow \\
& \{i := i_c + 1 - 1\} \\
& \langle\, \text{while (i > 0) \{} \\
& \quad \text{i--;} \\
& \quad \text{\} }\rangle\, (i \doteq 0)
\end{aligned}
& (\text{P12})
\end{array}
\qquad
\begin{array}{ll}
\begin{aligned}
& i_c + 1 > 0, \\
& i_c + 1 \geq 0, \\
& i_c \geq 0, \\
& \{i := i_c\} \\
& \langle\, \text{while (i > 0) \{} \\
& \quad \text{i--;} \\
& \quad \text{\} }\rangle\, (i \doteq 0) \\
& \Longrightarrow \\
& \{i := i_c\} \\
& \langle\, \text{while (i > 0) \{} \\
& \quad \text{i--;} \\
& \quad \text{\} }\rangle\, (i \doteq 0)
\end{aligned}
& (\text{P13})
\end{array}
$$

After update simplification, in (P13) we can make the observation that the induction hypothesis and the induction conclusion are syntactically equal and we can close the step case of this inductive proof.
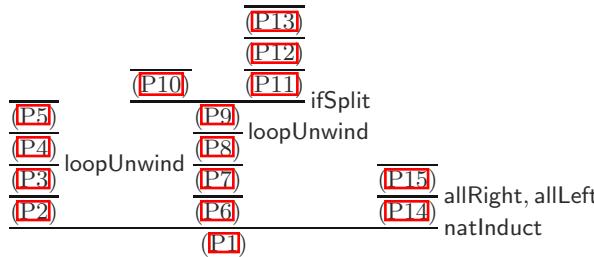
Use Case

Finally, it is time to apply the result that we have proven with the base case, the step case and the induction principle. The use case of the natInduct rule lets us make use of the result. For our program example, the use case is (P14):

$$
\begin{array}{ll}
\begin{aligned}
& \forall il.(il \geq 0 \,\&\, \\
& (il \geq 0 \rightarrow \\
& \quad \{i := il\} \\
& \quad \langle\, \text{while (i > 0) \{} \\
& \qquad \text{i--;} \\
& \qquad \text{\} }\rangle\, (i \doteq 0))) \\
& \Longrightarrow \\
& \forall il.(il \geq 0 \rightarrow \\
& \quad \{i := il\} \\
& \quad \langle\, \text{while (i > 0) \{} \\
& \qquad \text{i--;} \\
& \qquad \text{\} }\rangle\, (i \doteq 0))
\end{aligned}
& (\text{P14})
\end{array}
\qquad
\begin{array}{ll}
\begin{aligned}
& i_c \geq 0, \\
& i_c \geq 0 \rightarrow \\
& \quad \{i := i_c\} \\
& \quad \langle\, \text{while (i > 0) \{} \\
& \qquad \text{i--;} \\
& \qquad \text{\} }\rangle\, (i \doteq 0) \\
& \Longrightarrow \\
& i_c \geq 0 \rightarrow \\
& \quad \{i := i_c\} \\
& \quad \langle\, \text{while (i > 0) \{} \\
& \qquad \text{i--;} \\
& \qquad \text{\} }\rangle\, (i \doteq 0)
\end{aligned}
& (\text{P15})
\end{array}
$$

First, we have to apply allRight to get rid of the universal quantifier in the succedent, and il gets replaced by the Skolem constant i_c. Then, after automatic processing (or andLeft) we can proceed to instantiate the universal quantifier in the antecedent, using allLeft to supply the constant i_c, leaving (P15). Now we can close the use case and we are done with the proof!

Fig. 11.2. Proof tree for a simple decrementing loop

Note that the precondition $0 \le il$ in the proof obligation of this example was not really needed. Since we used the induction rule for natural numbers this condition is present anyway. However, we kept it for the proof obligation to be self-contained.

11.4.3 Making the Proof in the KeY System

You have seen a simple inductive program proof in JAVA CARD DL. Now there is some good news. The size of the proof description in the above section does not reflect the difficulty of the proof—the proof is very easy to complete in KeY—but just the rather high level of detail.

In fact, most of the steps shown in the proof above can be performed automatically by the KeY prover. With the correct settings in the prover all steps can be automatically derived except for the application of induction itself using natInduct, the unwinding of the loops using the loopUnwind rule, and some instantiations of the universally quantified variables using allLeft. The two latter kinds of user-interactions are rather straight-forward, and most likely subject to automation in the future. In general, the harder part is the correct application of the induction; the following four sections of this chapter are devoted to describing some of the difficulties and solutions.

If you have not done that already, now is a good point to try to make the proof in the KeY system. The proof obligation (P1) that we have seen in JAVA CARD DLcan be expressed in `.key` syntax:

──── KeY ────

```
\programVariables {
    int i;
}
\problem {
    \forall int il;
    {i:=il} 0 <= i -> (( \<{while (i>0) i--;}\> i=0)))
}
```
── KeY ────

This formula can be found in the file `decrToZero.key` in the KeY distribution, and uploaded into the KeY standalone prover. In Chap. 10 the details are presented on how interact with the system more concretely and how to run the automatic processing.

11.5 Choosing the Induction Variable

11.5.1 The Difficulty of Guiding Induction Proofs

Induction is a rather simple concept, however, the use of it can be very complicated as we will see in the following sections. It is common that a proof attempt of a valid formula fails. As most proof attempts are rather involved, performing mechanical induction can be a very cumbersome and frustrating experience. In this and the following sections some of the fundamental difficulties are addressed and the user interaction that is required to guide the prover to success is be explained. We will also see how failed proof attempts often give valuable insights into a better approach. The problems we discuss concern the choices necessary for the application of induction, that is, the choice of the induction variable (this section), the induction rule (Sect. 11.6) and the induction formula (Sect. 11.7).

11.5.2 How to Choose the Induction Variable

When guiding an induction proof, the user has to supply information along the way. Generally, the choice of the induction variable is the first one we make.

To choose the variable, start by studying the problem, the proof obligation at hand. Where is the "inductiveness" in the problem? If you are trying to prove the total correctness of a loop, termination and data correctness are proved in one go. A useful trick is to look at the terminating condition of the loop. The value of the induction variable should be in the domain of the base case when the loop condition is false. And when the loop condition is true, the value of the induction variable should be in the domain of the step case.

In the examples we have seen so far, it has been trivial to choose the induction variable, because there has been only one variable available in each of the cases. Still, for the example in Sect. 11.4 we can confirm this rule of thumb by observing that the value of the induction variable il is in the domain of the base case (that is, 0), whenever the loop terminates.

Sometimes none of the variables at hand can serve as the induction variable, and the trick is then to create a new variable that is defined to be a function of one or more of the existing variables. A common case is to introduce one new (logical) variable that is defined as a relation among existing variables. For instance, consider the following example.

Example 11.2 (Loop incrementing to n). Assume that the specification of the loop below stipulates that it terminates for non-negative `n` in a state where `i` $\doteq$ `n`.

─── JAVA ───

```java
int i,n;
// some initialisation code
i = 0;
while (i < n) {
  i++;
}
```

─────────────────────────────── JAVA ───

Note that `n` and `i` are program variables and `n` is not updated by the loop so we can regard it as a constant. But we do not know the value of it so we need to prove that the loop terminates for all values of `n` and so we need to introduce an extra logical variable nl to universally quantify over. For simplicity we use a precondition that restricts nl to non-negative numbers. Also note that the value of `i` is assigned to 0 just before the loop. We can then construct a JAVA CARD DL proof obligation (P1). After simplification, using the allRight and assignment rules, the Skolem constant n_c is introduced and we have (P2).

$$
\begin{array}{l}
\Longrightarrow \\
\forall nl.nl \geq 0 -> \\
\quad \{\texttt{n} := nl\} \\
\quad \langle\ \texttt{i = 0;} \\
\quad\ \ \texttt{while (i < n) \{} \\
\quad\ \ \ \ \texttt{i++;} \\
\quad\ \ \texttt{\} }\rangle\ (\texttt{i} \doteq nl)
\end{array}
\qquad (\text{P1})
$$

$$
\begin{array}{l}
n_c \geq 0 \\
\Longrightarrow \\
\{\texttt{i} := 0\,\|\,\texttt{n} := n_c\} \\
\langle\ \texttt{while (i < n) \{} \\
\quad\ \texttt{i++;} \\
\texttt{\} }\rangle\ (\texttt{i} \doteq n_c)
\end{array}
\qquad (\text{P2})
$$

Now, what would we use as the induction variable to prove (P2)? The initial guess might be to do induction over `i`, but we would not be able to close the base case then since the loop does not terminate for `i` $\doteq$ 0. We cannot use the constant `n` as the induction variable either, so this is a case where we need to come up with a new (logical) variable to perform induction over. Let us call this new logical variable kl. Given that the loop should terminate when `i` $\geq$ `n` (the negation of the loop condition) and $kl \doteq 0$ (in the base case), we can define `i` $\doteq n_c - kl$. So we apply induction with kl as the induction variable and with $\psi(kl)$ (P3) as the induction formula. The requirement that $kl \geq 0$ need not be explicit since that is part of natInduct. After applying induction, we get three proof branches. The first one (P4), for the base case, is easy to close because the program state is `i` = `n` = n_c, so when we apply the loopUnwind rule the loop is not entered and the postcondition is valid for this state.

$$\forall nl.nl \geq 0 \rightarrow$$
$$\{\texttt{i} := nl - kl \,||\, \texttt{n} := nl\}$$
$$\langle \texttt{ while (i < n) \{} \qquad (\text{P3})$$
$$\texttt{i++;}$$
$$\texttt{\} } \rangle (\texttt{i} \doteq nl)$$

$$n_c \geq 0$$
$$\Longrightarrow$$
$$\{\texttt{i} := n_c - 0 \,||\, \texttt{n} := n_c\}$$
$$\langle \texttt{ while (i < n) \{} \qquad (\text{P4})$$
$$\texttt{i++;}$$
$$\texttt{\} } \rangle (\texttt{i} \doteq n_c)$$

For the step case (P5), we have to unwind the loop body in the succedent, using the loopUnwind rule as usual, and automatic processing triggers ifSplit to branch again. The branch that involves the preconditions can be closed easily. In the other branch, first the user has to unwind the loop body in the succedent, apply automatic processing, and then use allLeft to instantiate nl with n_c (P6). After simplification the antecedent and succedent are syntactically equivalent and the branch can be closed.

$$k_c \geq 0,$$
$$n_c \geq 0,$$
$$\forall nl.nl \geq 0 \rightarrow$$
$$\{\texttt{i} := nl - k_c \,||\, \texttt{n} := nl\}$$
$$\langle \texttt{ while (i < n) \{}$$
$$\texttt{i++;}$$
$$\texttt{\} } \rangle (\texttt{i} \doteq nl)$$
$$\Longrightarrow$$
$$\{\texttt{i} := n_c - (k_c + 1) \,||\, \texttt{n} := n_c\}$$
$$\langle \texttt{ while (i < n) \{}$$
$$\texttt{i++;}$$
$$\texttt{\} } \rangle (\texttt{i} \doteq n_c)$$
$$(\text{P5})$$

$$k_c \geq 0,$$
$$n_c \geq 0,$$
$$\{\texttt{i} := n_c - k_c \,||\, \texttt{n} := n_c\}$$
$$\langle \texttt{ while (i < n) \{}$$
$$\texttt{i++;}$$
$$\texttt{\} } \rangle (\texttt{i} \doteq n_c)$$
$$\Longrightarrow$$
$$\{\texttt{i} := n_c - (k_c + 1) + 1 \,||\, \texttt{n} := n_c\}$$
$$\langle \texttt{ while (i < n) \{}$$
$$\texttt{i++;}$$
$$\texttt{\} } \rangle (\texttt{i} \doteq n_c)$$
$$(\text{P6})$$

Now the use case (P7) remains. Here we need to show the original proof goal (P2) using the the induction formula $\psi(kl)$ (P3) that we have proved by now. We must instantiate (using allLeft) the universal quantifiers correctly. The trick is to compare the updates of the antecedent and the succedent, and to find the instantiations for kl and nl so that syntactic equivalence can be achieved. We start with the outer quantifier (kl), and instantiate it with n_c, and then after simplification we instantiate n_c for nl as well, to get (P8). After simplification, the use case can be closed as well and we are done with the proof.

$$n_c \geq 0,$$
$$\forall kl.kl \geq 0 \rightarrow$$
$$\quad \forall nl.nl \geq 0 \rightarrow$$
$$\quad\quad \{i := nl - kl \,||\, n := nl\}$$
$$\quad\quad \langle\, \text{while (i < n) \{}$$
$$\quad\quad\quad \text{i++;}$$
$$\quad\quad \text{\} } \rangle\, (i \doteq nl)$$
$$\Longrightarrow$$
$$\{i := 0 \,||\, n := n_c\}$$
$$\langle\, \text{while (i < n) \{}$$
$$\quad \text{i++;}$$
$$\text{\} } \rangle\, (i \doteq n_c)$$

(P7)

$$n_c \geq 0,$$
$$\{i := n_c - n_c \,||\, n := n_c\}$$
$$\langle\, \text{while (i < n) \{}$$
$$\quad \text{i++;}$$
$$\text{\} } \rangle\, (i \doteq n_c)$$
$$\Longrightarrow$$
$$\{i := 0 \,||\, n := n_c\}$$
$$\langle\, \text{while (i < n) \{}$$
$$\quad \text{i++;}$$
$$\text{\} } \rangle\, (i \doteq n_c)$$

(P8)

Again, once you know how to apply the induction rule correctly, the proof is very easy to perform in the KeY prover. Most of the proof steps can be done automatically. The user interaction required is similar to what we have seen before, only this time the induction variable was a bit trickier to find. So go ahead and load the problem expressed in `.key` syntax:

——— KeY ———

```
\programVariables {
   int i;
   int n;
}

\problem{
   \forall int nl; (nl >= 0 ->
      {n:=nl} \<{i=0; while (i<n) i++;}\> i = nl)
}
```

——————————————————— KeY ———

11.6 Different Induction Rules

So far we have seen the rule natInduct that implements (first-order) Peano induction in KeY. This form of induction allows us to prove that a formula is valid for all natural numbers. For instance, we can prove the preceding examples quite easily. This is because in those examples the induction variable was decremented by steps of one and ended at zero. However, if we alter the simple decrementor from Sect. 11.4 slightly to for instance stepping down by a non-zero constant c it becomes very hard to make immediate use of the natInduct rule. Consider the proof obligation $\forall il.\phi(il)$ where $\phi(il)$ is:

$$\forall cl.(il \geq 0 \ \& \ cl \geq 1 \ ->$$

```
{i := il || c := cl}
⟨ while (i > 0) {
    if(i >= c) {
       i = i - c;
    } else {
       i--;
    }
} ⟩ i ≐ 0)
```

In this example the obvious choice for the induction variable is `i` (see the terminating condition of the loop). Picking the right formula is generally a lot more complicated as we see in the next section. However, the simplest possible choice of induction formula for proving correctness of this loop would be $\phi(il)$. It is completely schematic and requires no interaction with the user. The hypothesis, however, is too weak when using natInduct. Roughly speaking, in a proof attempt of the standard step case, $\forall il.(\phi(il) \ -> \ \phi(il+1))$, the following happens: the while-loop is unwound for $il+1$ and the proof branches at the conditional statement. One case (the one with `i--;`) is possible to prove, because $\{i := il\}$`(i+1)--;` is equal to il after symbolic execution and simplification. The proof obligation for this case simplifies to $\forall il.(\phi(il) \ \& \ il < cl \ -> \ \phi(il))$, which is valid. In the other case symbolic execution gives $il+1-cl$ so that the resulting proof obligation $\forall il.(\phi(il) \ \& \ il \geq c \ -> \ \phi(il+1-cl))$ is in general unprovable. So now we have arrived at the inconvenient fact that despite that the formula is valid for all natural numbers we can not prove the unmodified formula using natInduct.

11.6.1 Customised Induction Rules

To remedy this inconvenience, a method to generate *customised induction rules* [Olsson and Wallenburg, 2005, Hähnle and Wallenburg, 2003] has been developed within the KeY project. A customised induction rule can be (automatically) generated[2] to match a particular proof obligation. Its construction ensures soundness. For our example above, the following customised induction rule can be produced (for brevity we have omitted the use case):

constDecrRule
$$\frac{\Gamma \Longrightarrow \phi(0) \ \& \cdots \& \ \phi(c-1), \ \Delta \qquad \Gamma \Longrightarrow \forall il.(\phi(il) \ -> \ \phi(il+c)), \ \Delta}{\Gamma \Longrightarrow \forall n.\phi(n), \ \Delta}$$

---

[2] At the time of writing, the generation of customised induction rules is not yet a part of the KeY distribution.

Note that this customised induction rule is powerful enough to make the proof go through almost automatically with the unchanged induction formula $\phi(il)$ which is very desirable in our effort to minimise user interaction.

In fact, these customised induction rules are generated from information gained during naive failed proof attempts and they always follow a certain pattern:

$$
\text{customInduct} \; \frac{
\begin{array}{c}
\Gamma \Longrightarrow \forall i.BC(i) \mathbin{-\!\!>} \phi(i), \; \Delta \\
\Gamma \Longrightarrow \forall i.(BP_1(i) \;\&\; \phi(p_1(i)) \mathbin{-\!\!>} \phi(i)), \; \Delta \\
\vdots \\
\Gamma \Longrightarrow \forall i.(BP_n(i) \;\&\; \phi(p_n(i)) \mathbin{-\!\!>} \phi(i)), \; \Delta
\end{array}
}{
\Gamma \Longrightarrow \forall i.\phi(i), \; \Delta
}
$$

where $BC(i) \mathbin{<\!\!\!-\!\!\!>} \neg BP_1(i) \;\&\; \ldots \;\&\; \neg BP_n(i)$ must hold. The predicate BP_j is the condition required for ensuring execution of that branch of the program which causes the induction variable i to have the term $p_j(i)$ as value after execution. Before we can trust this rule we must make sure that it is sound, i.e., that there is an induction principle that allows us to conclude $\forall i.\phi(i)$. We need to explain a few things ahead of that so the soundness discussion is postponed to Sect. 11.6.3. But first we continue to study the usefulness of customised induction rules.

By using customised induction rules, the gain is twofold; first, they decrease the need for generalisation and second, they branch the proof at an earlier stage so that the resulting proof becomes easier to develop or a failed proof attempt becomes easier to "debug". Next we study a detailed example of a problem that benefits from the use of customised induction rules.

Example 11.3 (Russian multiplication). Let us have a look at a program that computes the product of two numbers, al and bl, using only division and multiplication by 2, i.e., bit-shifts. The program uses an accumulator for the product that has the initial value of zl. Consider the problem of proving $\forall al.\phi(al)$, where $\phi(al)$ is:

$$
\begin{aligned}
&\forall bl, zl. \\
&\{\texttt{a} := al \,\|\, \texttt{b} := bl \,\|\, \texttt{z} := zl\} \\
&\langle\, \texttt{while (a != 0) \{} \\
&\quad \texttt{if(a \% 2 != 0) z = z + b;} \\
&\quad \texttt{a = a / 2;} \\
&\quad \texttt{b = b * 2;} \\
&\quad \texttt{\} }\,\rangle\; \texttt{z} \doteq zl + al * bl
\end{aligned}
$$

Using the natInduct rule, we get the instantiations of the induction formula $\phi(al)$ and $\phi(al+1)$ in the step case (P9):

$$\forall bl, zl.$$
$$\{\mathtt{a} := al_c \,||\, \mathtt{b} := bl \,||\, \mathtt{z} := zl\}$$
$$\langle\, \mathtt{while\ (a\ !=\ 0)\ \{}$$
$$\quad \mathtt{if(a\ \%\ 2\ !=\ 0)\ z\ =\ z\ +\ b;}$$
$$\quad \mathtt{a\ =\ a\ /\ 2;}$$
$$\quad \mathtt{b\ =\ b\ *\ 2;}$$
$$\,\}\,\rangle\, \mathtt{z} \doteq zl + al_c * bl$$
$$\Longrightarrow$$
$$\{\mathtt{a} := al_c + 1 \,||\, \mathtt{b} := bl_c \,||\, \mathtt{z} := zl_c\}$$
$$\langle\, \mathtt{while\ (a\ !=\ 0)\ \{}$$
$$\quad \mathtt{if(a\ \%\ 2\ !=\ 0)\ z\ =\ z\ +\ b;}$$
$$\quad \mathtt{a\ =\ a\ /\ 2;}$$
$$\quad \mathtt{b\ =\ b\ *\ 2;}$$
$$\,\}\,\rangle\, \mathtt{z} \doteq zl_c + (al_c + 1) * bl_c$$
$$\text{(P9)}$$

$$\forall bl, zl.$$
$$\{\mathtt{a} := al_c \,||\, \mathtt{b} := bl \,||\, \mathtt{z} := zl\}$$
$$\langle\, \mathtt{while\ (a\ !=\ 0)\ \{}$$
$$\quad \mathtt{if(a\ \%\ 2\ !=\ 0)\ z\ =\ z\ +\ b;}$$
$$\quad \mathtt{a\ =\ a\ /\ 2;}$$
$$\quad \mathtt{b\ =\ b\ *\ 2;}$$
$$\,\}\,\rangle\, \mathtt{z} \doteq zl + al_c * bl$$
$$\Longrightarrow$$
$$\{\mathtt{a} := (al_c + 1)/2 \,||\, \mathtt{b} := 2 * bl_c \,||$$
$$\mathtt{z} := zl_c\}$$
$$\langle\, \mathtt{while\ (a\ !=\ 0)\ \{}$$
$$\quad \mathtt{if(a\ \%\ 2\ !=\ 0)\ z\ =\ z\ +\ b;}$$
$$\quad \mathtt{a\ =\ a\ /\ 2;}$$
$$\quad \mathtt{b\ =\ b\ *\ 2;}$$
$$\,\}\,\rangle\, \mathtt{z} \doteq zl_c + (al_c + 1) * bl_c$$
$$\text{(P10)}$$

But the proof attempt that follows is doomed already after the symbolic execution of the loop body in the succedent, (P10). Syntactic equivalence between the updates of a cannot be achieved and the proof obligation cannot be closed. There is no point in proceeding with the attempt to perform substantiations of bl and zl.

Instead, we need to use a more powerful induction rule. If we analyse the program by the method in [Olsson and Wallenburg, 2005] we can generate a customised induction rule that is suitable for proving the particular program.

$$\text{russianCustomInduct} \quad \frac{\begin{array}{l} \Gamma \Longrightarrow \phi(0),\ \Delta \\ \Gamma \Longrightarrow \forall n.(n \bmod 2 \doteq 0\ \&\ \phi(n/2) \to \phi(n)),\ \Delta \\ \Gamma \Longrightarrow \forall n.(n \bmod 2\,! \doteq 0\ \&\ \phi(n/2) \to \phi(n)),\ \Delta \end{array}}{\Gamma \Longrightarrow \forall n.\phi(n),\ \Delta}$$

Using the customised rule instead our proof attempt works in the following way. We apply the russianCustomInduct rule, and study the *even* step case, (P11):

$al_c \bmod 2 \doteq 0,$
$\forall bl, zl.$
$\{\mathtt{a} := al_c/2 \,\|\, \mathtt{b} := bl \,\|\, \mathtt{z} := zl\}$
$\langle$ while (a != 0) {
   if(a % 2 != 0) z = z + b;
   a = a / 2;
   b = b * 2;
  } $\rangle$ $\mathtt{z} = zl + (al_c/2) * bl$
$\Longrightarrow$
$\{\mathtt{a} := al_c \,\|\, \mathtt{b} := bl_c \,\|\, \mathtt{z} := zl_c\}$
$\langle$ while (a != 0) {
   if(a % 2 != 0) z = z + b;
   a = a / 2;
   b = b * 2;
  } $\rangle$ $\mathtt{z} \doteq zl_c + al_c * bl_c$

(P11)

$al_c \bmod 2 \doteq 0,$
$\forall bl, zl.$
$\{\mathtt{a} := al_c/2 \,\|\, \mathtt{b} := bl \,\|\, \mathtt{z} := zl\}$
$\langle$ while (a != 0) {
   if(a % 2 != 0) z = z + b;
   a = a / 2;
   b = b * 2;
  } $\rangle$ $\mathtt{z} \doteq zl + (al_c/2) * bl$
$\Longrightarrow$
$\{\mathtt{a} := al_c/2 \,\|\, \mathtt{b} := 2 * bl_c \,\|\,$
  $\mathtt{z} := zl_c\}$
$\langle$ while (a != 0) {
   if(a % 2 != 0) z = z + b;
   a = a / 2;
   b = b * 2;
  } $\rangle$ $\mathtt{z} \doteq zl_c + al_c * bl_c$

(P12)

We follow the standard proving process for loops and unwind the loop in the succedent. This time, the effect of the loop body matches the induction step as we can see by comparing the updates of a in the antecedent and succedent of (P12). For this proof obligation, all that remains is to perform the obvious instantiations for bl and zl; find the right values to achieve syntactic equivalence by studying the updates in the succedent.

Finally, we arrive at stage (P13) where, after some basic arithmetic simplification, we can close the proof branch. The step case for odd numbers follows the same pattern, but involves more arithmetic in the final step. For instance, we need to take advantage of the fact that if al_c is odd, then $al_c/2 \doteq (al_c - 1)/2$. This proof branch is left as an exercise to the reader.

$al_c \bmod 2 \doteq 0,$
$\{\mathtt{a} := al_c/2 \,\|\, \mathtt{b} := 2 * bl_c \,\|\, \mathtt{z} := zl_c\}$
$\langle$ while (a != 0) {
   if(a % 2 != 0) z = z + b;
   a = a / 2;
   b = b * 2;
  } $\rangle$ $\mathtt{z} = zl_c + (al_c/2) * 2 * bl_c$
$\Longrightarrow$
$\{\mathtt{a} := al_c, \ \mathtt{b} := 2 * bl_c, \ \mathtt{z} := zl_c\}$
$\langle$ while (a != 0) {
   if(a % 2 != 0) z = z + b;
   a = a / 2;
   b = b * 2;
  } $\rangle$ $\mathtt{z} \doteq zl_c + al_c * bl_c$

(P13)

11.6.2 The Noetherian Induction Rule

In order to prepare for the soundness discussion in the following section, we will introduce the perhaps most important variant of induction—Noetherian induction. *Noetherian induction*, or *Well-founded induction*, is the most general form of induction, meaning that all other induction rules are special cases of this one.

Theorem 11.4 (Principle of Noetherian Induction). *Let $(M, \prec_M)$ be a well-founded set. If $\forall m \in M$ we have that $(\forall k \in M.k \prec_M m \to \phi(k)) \to \phi(m)$ holds, then $\forall m \in M.\phi(m)$.*

Note that this induction schema is valid for any induction set $(M, \prec_M)$, provided that it is well-founded. Below is the principle expressed as a proof rule.

$$\text{noetherInduct} \quad \frac{\Gamma \Longrightarrow \forall m.(\forall k.(k \prec_M m \to \phi(k))) \to \phi(m), \; \Delta \quad \Gamma, \; \forall m.\phi(m) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

The induction hypothesis is stronger in the Noetherian induction rule than in other induction rules. Using this rule, we avoid the need for generalisation of the proof obligation in many cases. For instance the Russian multiplication example above could also be proven using Noetherian induction. Still, the user-interaction involved in the remaining proof after application of the Noetherian induction rule can be very complicated because of the inherent mix of data and control-flow correctness and the size of the proof. Therefore, a customised induction rule might be preferable, even though it is not the strongest form of induction.

11.6.3 Soundness of Induction Rules

As we pointed out in Sect. 11.3, for an induction rule to be sound, it must be supported by an induction principle such as the principle of mathematical induction, or the principle of well-founded induction above. One way to establish soundness of an arbitrary induction rule is to show 1) that the rule is an instance of the Noetherian induction principle, and 2) that the induction set is well-founded. This is possible since Noetherian induction is the most general form of induction.

For example, we can (informally here) convince ourselves that the principle of mathematical induction is sound by considering the relation $\prec_{+1}$ where $\prec_{+1} \Longleftrightarrow \{(i, i+1) \mid i \in \mathbb{N}\}$. Instantiating $(M, \prec_M)$ with $(\mathbb{N}, \prec_{+1})$ reduces the first premiss of noetherInduct to the two familiar proof obligations $\phi(0)$ and $\forall n.(\phi(n) \to \phi(n+1))$. To see this, let m be either 0 or $n+1$ where $n : \mathbb{N}$. For $m = 0$, there is no $k : \mathbb{N}$ such that $k \prec_{+1} 0$ and the entire proof obligation

reduces to $\phi(m)$, which is $\phi(0)$ in this case. For $m = n + 1$, since there is exactly one k such that $k \prec_{+1} n + 1$, namely $k = n$, $\forall k.k \prec_{+1} n + 1 \Rightarrow \phi(k)$ reduces to $\phi(n)$ and we get the proof obligation of the step case in natInduct. Furthermore, since the set $(\mathbb{N}, \prec_{+1})$ is well-founded, we can see that natInduct is a sound induction rule.

Similarly, another example of a very useful and well-known induction schema for natural numbers is the following, sometimes called *strong induction on* $\mathbb{N}$:

Theorem 11.5 (Second Principle of Mathematical Induction). *If the formula* $\forall n.(\forall k.(k < n \Rightarrow P(k))) \Rightarrow P(n)$ *holds, then* $\forall n.P(n)$ *can be concluded.*

This is clearly an instance of Noetherian induction with the well-founded induction set being $(\mathbb{N}, <)$.

The customised induction rules are also instances of well-founded induction. Consider the basic customised induction rule CustomInduct above. This can be shown to be an instance of Noetherian induction, using the induction set $(\mathbb{N}, \prec)$, where $i \prec j$ holds iff $BP_1(j)$ & $p_1(j) \doteq i \mid \ldots \mid BP_n(j)$ & $p_n(j) \doteq i$, see [Olsson and Wallenburg, 2005]. However, only if the induction set $(\mathbb{N}, \prec)$ is well-founded, the induction rule is sound. For a general problem, or an arbitrary customised rule, this is not guaranteed. In fact, to prove well-foundedness of $(\mathbb{N}, \prec)$ is equivalent to proving termination of the loop in the problem. So, the way we have stated CustomInduct, it is unsound. In order construct sound rules mechanically, the method [Olsson and Wallenburg, 2005] introduces extra proof obligations that imposes restrictions on the order $\prec$, to for instance require that all branches in the loop are decreasing the induction variable.

In summary: do not forget the induction principle! You cannot just write a new induction taclet for every problem: the use case must implement a proper induction principle to ensure soundness.

11.7 Generalisation of Induction Formulae

Despite using the correct induction variable and induction rule, a proof still might fail. Even Noetherian induction sometimes is not enough to prove a valid formula. This is because of the inherent incompleteness of first-order rules, see Sect. 2.7. However, many times we can work around this problem by proving something stronger than is actually needed and then prove that if the stronger property holds, the original proof obligation also holds. This is called generalisation since the new, stronger induction formula is a generalisation of the original formula. In the following section we show how this technique can be used.

11.7.1 Cubic Sum Example

In this section we describe the correctness proof of a program that computes the so called cubic sum and therefore should fulfill the following property:

$$\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4 \ . \tag{P1}$$

This problem can be formalised in JAVA CARD DL and presented to the KeY prover using the following input:

—— KeY ——

```
\programVariables{
  int n,i,r;
}
\problem{
  \forall int nl; (0 <= nl) ->
    {n:=nl}
    \<{ i=0;
        r=0;
        while (i < n) {
          i++;
          r = r + (i*i*i);
        }
    }\> 4*r = nl*nl*(nl+1)*(nl+1) )
}
```

—— KeY ——

As usual when we are about to apply induction, we have to make some choices: Which induction variable, which induction rule and which induction formula should we use?

For the induction variable, a glance at the terminating condition tells us that $n - i$ is zero when the loop terminates. Then it is a good idea to introduce a new logical variable for this expression and let that be the induction variable. Let us call this induction variable kl, where $kl = n - i$. We know that kl has to be a logical variable since we need to quantify over it when we apply induction.

By observing the change to the induction variable kl that happens inside the loop body, we can decide which induction rule to use. Since the induction variable kl is decremented by one, this happens implicitly at `i++;`, we see that the Peano induction rule is a good match. Despite this match, a proof attempt would fail, and another induction rule would not help. Let us have a look at why.

The central problem here is to suitably generalise the induction formula. Let us have a look at the starting point for the induction application. The proof obligation after the usual simplification involving allRight is (P2). In

order to make the formulas more readable we write nl_c^2 instead of $nl_c * nl_c$, etc.

$$\Longrightarrow$$
$$nl_c < 0,$$
$$\{\texttt{i} := 0 \,\|\, \texttt{n} := nl_c \,\|\, \texttt{r} := 0\}$$
$$\langle \; \texttt{while (i < n) \{} \qquad\qquad (\text{P2})$$
$$\quad \texttt{i++;}$$
$$\quad \texttt{r= r+(i*i*i);}$$
$$\texttt{\} } \,\rangle\; 4\texttt{r} \doteq nl_c^2(nl_c + 1)^2$$

$$0 \leq nl_c \;\text{->}$$
$$\{\texttt{i} := nl_c - kl \,\|\, \texttt{n} := nl_c \,\|\, \texttt{r} := 0\}$$
$$\langle \; \texttt{while (i < n) \{}$$
$$\quad \texttt{i++;}$$
$$\quad \texttt{r= r+(i*i*i);}$$
$$\texttt{\} } \,\rangle\; 4\texttt{r} \doteq nl_c^2(nl_c + 1)^2$$
$$\hspace{6cm} (\text{P3})$$

If we make a naive choice for the induction formula (P3), taking the original formula and just changing the update of $\texttt{i}$ to connect it with the induction variable kl and with nl, the corresponding naive proof attempt fails. Only the use case is provable. In the base case, after loopUnwind, the loop does indeed terminate but the postcondition is not fulfilled, leaving an open proof branch with $0 \doteq nl_c^2(nl_c + 1)^2$. Furthermore, the step case, after loopUnwind and simplification, is impossible to close due to the syntactic differences of the updates: $\{\texttt{r} := 0\}$ in the antecedent and $\{\texttt{r} := (nl_c - kl)^3\}$ in the succedent. So we have reached a position in the induction proving process where we need to generalise the induction formula. The latter of these two problems, the syntactic difference in the updates, is remedied in a rather straightforward fashion by introducing a new universally quantified variable. In our example, let us introduce a variable rl that represents the initial value of $\texttt{r}$. But we are not done yet. As hinted by the failed base case, we also need to generalise the post condition. So how would one generalise the post condition to make it go through all the branches: base case, step case and use case? In general this requires some intuition about the problem at hand, and this can be very tricky. One way to solve the problem more systematically is to make another proof attempt and when it fails, simply learn from that. Since we have one mandatory patch for our example already, the introduction of rl, let us try with that one and see what we can learn. So we use the following induction formula:

$$\forall rl.(0 \leq rl \;\text{->}$$
$$\{\texttt{i} := nl_c - kl \,\|\, \texttt{n} := nl_c \,\|\, \texttt{r} := rl\}$$
$$\langle \; \texttt{while (i < n) \{}$$
$$\quad \texttt{i++;} \qquad\qquad\qquad\qquad\qquad (\text{P4})$$
$$\quad \texttt{r = r+(i*i*i);}$$
$$\texttt{\} } \,\rangle\; 4\texttt{r} \doteq nl_c^2(nl_c + 1)^2)$$

We repeat the procedure of applying simplification and unwinding the while loop in both the base case and the step case. This time when we get to the syntactically different updates in the step case, the solution is right at hand: we simply use allLeft to instantiate the universally quantified rl in the antecedent with the value of $\texttt{r}$ in the succedent's update. The use case is easy

to prove with just one more instantiation than before. But—as expected—at the base case the proof attempt still fails and this time it leaves the proof goal $4rl_c \doteq nl_c^2(nl_c + 1)^2$. So we got one step further, and we have a little more information about how to generalise the postcondition.

To be more concrete, let us call the sought-after post condition $Q(kl, rl)$, which depends on the induction variable kl and the new universally quantified variable rl. From the three proof obligations of our most recent failed proof attempt, and from the fact that we know how we need to proceed with the proof, we can generate three constraints that must hold:

$$\{\mathtt{r} := rl_c\}\ Q(0, rl_c) \tag{P5}$$

$$Q(kl_c, rl_c + (nl_c - kl_c)^3)\ \; <\!\!> \;\ Q(kl_c + 1, rl_c) \tag{P6}$$

$$Q(nl_c, 0)\ \; <\!\!> \;\ 4\mathtt{r} \doteq nl_c^2(nl_c + 1)^2 \tag{P7}$$

In the following we describe how those constraints were generated. Starting with the base case, what do we know from the proof obligation and the proving process? We know that the induction variable kl is 0 and we also know that after skolemisation, the update is $\{\mathtt{r} := rl_c\}$, and therefore after unwinding the loop, we have the proof obligation (P5). Next, consider a proof attempt of the step case. After skolemisation of kl, unwinding of the loop in the succedent, skolemisation of rl in the succedent, and simplification, the correct instantiation of the universally quantified rl in the antecedent has to be done. To achieve syntactic equality in the updates, the value of $\mathtt{r}$, i.e., $rl_c + (nl_c - kl_c)^3$, is supplied. It is important to note that this substitution also effects the postcondition Q in the antecedent, since Q depends on rl, remember (P4). What now remains to achieve is syntactic equality between the postconditions, thus we can generate the constraint (P6). Finally, what do we get from the use case? The two instantiations that have to be performed, nl_c for kl and 0 for rl are easily identified based on the values of the updates. Again, syntactic equality of the updates remains to be proven and this gives us the third constraint (P7).

Now, by solving our constraints (P5), (P6), and (P7) we can get hold of our desired postcondition. It turns out that the constraints are satisfied by the following $Q(kl, rl)$:

$$4(\mathtt{r} - rl) \doteq nl_c^2(nl_c + 1)^2 - (nl_c - kl)^2(nl_c - kl + 1)^2\ .$$

Using this $Q(kl, rl)$ as our postcondition we can create an induction formula $\phi(kl)$ that we by now know is provable:

$$\forall rl.(0 \leq rl \rightarrow$$
$$\{\texttt{i} := nl_c - kl \,\|\, \texttt{n} := nl_c \,\|\, \texttt{r} := rl\}$$
$$\langle \;\texttt{while (i < n) \{}$$
$$\quad\texttt{i++;}$$
$$\quad\texttt{r = r+(i*i*i);}$$
$$\texttt{\} } \rangle \; 4(\texttt{r} - rl) \doteq nl_c^2(nl_c + 1)^2 - (nl_c - kl)^2(nl_c - kl + 1)^2)$$

(P8)

Note that in order to solve the constraint (P6) or, equivalently, to prove the step case, we need to perform a series of arithmetic rule applications that at the time of writing cannot be performed automatically by the KeY system. These include for example multiplication distributivity and the square rule. When performing a proof such as the cubic sum, you may find that you miss certain arithmetic rules. In that case, those can be implemented as taclets, uploaded to the system, proven (as lemmas, to be reused) and applied. For brevity we have excluded some straight-forward but technical details of this proof, in particular the arithmetical tricks do not belong to this discussion. The exact details on how to prove the cubic sum program can be found in the distribution of .key files that comes with this book.

11.8 Summary: The Induction Proving Process

At this point we take a look back to summarise what we have learned about making induction proofs in KeY. We present an overview of the induction proving process and some related useful tricks. This guide can be used to get started in the practice of inductive theorem proving or as a quick reference later on.

1. **Load the .key file** that contains the problem and possible extra taclets that can be introduced as lemmas when necessary (do not forget to prove these taclets as well).
2. **Apply strategy** to the original proof obligation. Make sure to have the **correct settings** in the prover (for loops: simple JAVA CARD DL, mathematical integers and no automatic unwinding of loops). Study the remaining proof goal.
3. **Choose an induction variable**. If you are trying to prove correctness of a loop, see the termination condition. Pick the induction variable so that when it is in the base case, the loop condition is false. You may have to introduce a new logical variable to be defined in terms of the existing variables, please refer to Sect. 11.5 and 11.7 for examples of this.
4. **Choose an induction rule**. See the update to the induction variable inside the loop; if it does not decrement the induction variable with 1, then it is easier to go for a customised induction rule or the Noetherian induction rule right away. Different induction rules are discussed in Sect. 11.6.

5. **Choose an induction formula**. Start with the proof obligation (the naive approach). Then edit the formula so that your induction variable appears in one of the updates. The simple example in Sect. 11.4 shows how to do this.

6. **Apply the induction rule**. Point to the sequent arrow and pick the rule to use, then you will be given the opportunity to supply the induction variable and the induction formula in the instantiation dialog that appears. Drag-and-drop is often very useful at this point.

7. After applying induction, the proof contains several branches; 1) either one or no base case, 2) at least one step case, and 3) one use case. **Simplify the branches** one at a time in a similar fashion:
   Apply strategy and invoke one of the external theorem provers throughout the process to maximize automation. It is useful to apply a strategy locally (right-click on the proof goal) on the current branch of the proof tree. You can also prune the proof tree locally if you wish to undo many steps in one go.

   a) **Base/Step cases**:
      Unwind the body of the loop in the succedent. Perform suitable instantiations. See the updates to find the right choices, they have to be the same in the antecedent and the succedent. You might need to manually apply arithmetical taclets to achieve syntactic equivalence in the updates and post conditions. Sometimes it is helpful to construct a lemma for an arithmetic property, prove that separately, and later hopefully reuse it.

   b) **Use case**:
      Suitable instantiation of the previously proven induction formula should do the trick. See the updates to find the right choices.

   If there is a case that cannot be closed and you strongly believe that the program and its specification are correct then move on to generalisation of the induction formula.

8. **Generalise the induction formula**, if needed. It is the updates and the pre- and postconditions that have to be changed, the program stays the same. A typical generalisation is for instance to introduce an all-quantified logical variable, for example, for the initial value of a program variable, and then connect the variables in the updates. Usually these additional logical variables need to be introduced in the postcondition so that the arithmetic works out. Try applying induction again.
   When you are experienced, possible generalisations can be spotted and introduced already at step (3). Otherwise this can be rather time-consuming. The best approach for learning is to start with a formula that is similar (or even identical) to the proof goal, make a proof attempt, see what is missing, be creative, generalise and try again. Sect. 11.7 discusses generalisation of the induction formula in greater detail.

11.9 Conclusion

In this chapter we described the induction proving process in the KeY system from the viewpoint of practical program verification. We discussed the mandatory requirements of user guidance of induction proofs and as a side effect we addressed the fundamentals of induction. We limited ourselves to reason about integers, mathematical integers instead of Java integers, and had a focus on proving total correctness of loops. This is an interesting class of problems, because here induction is both necessary and also rather difficult to use.

First-order logic with arithmetic is far from complete, as we have seen in Sect. 2.7. With the addition of induction we can prove most arithmetical properties that are practically interesting. Even though we suffer from Gödel's result, induction is still practically useful in program verification. As we saw, the main challenge is to deal with the complexity of user interaction. Some user interaction that is still necessary in the examples of this chapter will soon be avoidable by stronger support for automation in the KeY prover. We continue to work towards automation.

12

Java Integers

by
Steffen Schlager

Specification languages typically offer idealistic data types that are not available in programming languages. For example, the integer data types available in the specification languages UML/OCL, Z, B, and Java Card DL (which can also be seen as a specification language) are isomorphic to the mathematical integers $\mathbb{Z}$ having an infinite domain. On the other hand, the built-in integer types in Java and many other programming languages have finite domains. As a consequence the semantics of data types on specification and implementation level differ (at least on the boundaries of the domain).

In this section we clarify the quite subtle relation between integer data types in Java Card DL and Java. It turns out that the Java integers are not a *refinement* of the type `integer` in Java Card DL but a so-called *retrenchment* which constitutes a generalisation of refinement. Note, that the problems and solutions given in this section are not restricted to Java Card DL and Java. They analogously apply to other specification languages like e.g., UML/OCL, Z and B and to several programming languages like e.g., C and C++.

This section may also serve as a lightweight introduction to refinement and retrenchment (for a more detailed and systematic introduction we recommend [Derrick and Boiten, 2001] and [Banach and Poppleton, 1998], respectively). This section is based on material published in [Beckert and Schlager, 2004, 2005, Schlager, 2002].

12.1 Motivation

Refinement is a well-established and accepted technique for the systematic development of correct software systems. Starting from an initial formal specification, refinement steps are performed to obtain a correct implementation—a software system satisfying the specification. Each refinement step may add more details, e.g., by removing non-determinism.

To guarantee correctness of the system being developed, the refinement steps themselves must be correct, i.e., they must adhere to certain rules. Then, the refinement process preserves all properties of the abstract system. They are *automatically* satisfied by the concrete system as well and it is not necessary to re-prove them.

Here we concentrate on the refinement of data types (data refinement), in particular the refinement of integer data types. Following [He et al], [1986], a data refinement is correct if in all circumstances and for all purposes the concrete data type can be validly used in place of the abstract one, i.e., an observer cannot distinguish whether the concrete or the abstract type is used.

Consider, for example, the following Java method, which—supposedly—implements an operation computing the sum of two integers:

```
── Java ────────────────────────────────────────────
int sum(int x, int y) {
  return x+y;
}
────────────────────────────────────────── Java ──
```

The correctness of this implementation with respect to the specification can be expressed in Java Card DL as the formula (inlining the method body in the diamond modality)

$$\forall i.\forall j.\{\texttt{x} := i \,||\, \texttt{y} := j\} \,\langle\texttt{result=x+y;}\rangle\texttt{result} \doteq i + j$$

where i, j are variables of type `integer`.

At first sight this formula seems to be valid, i.e., the method is correct and returns the *mathematical* sum of the parameter values. In truth, however, the implementation is *not* correct. The reason is that the (finite) Java type `int` used in the implementation does not correctly refine the (infinite) Java Card DL type `integer`. In particular, the Java operation $+_\texttt{int}$ (addition on Java type `int`) is not a refinement of the operation $+_\texttt{integer}$ (addition on the mathematical integers), because it computes the sum of the two arguments *modulo the size of the type* `int`, e.g.:

$$2147483647 +_\texttt{integer} 1 = \quad 2147483648 \quad \text{but}$$
$$2147483647 +_\texttt{int} \quad 1 = -2147483648$$

To overcome the problem outlined above there are basically two approaches.

The idealistic approach, trying to stick to the refinement principle, amounts to changing the (semantics of the) types involved such that the concrete type is in fact a refinement of the abstract one. Possibly appealing from the theoretical point of view, in practice both modifying the abstract type and adapting the concrete type has serious drawbacks.

On the one hand, changing and adapting the implementation data type (e.g., using big number arithmetics instead of the built-in integer types),

introduces unnecessary and serious inefficiencies into the implementation. Moreover, such data type implementations are not always readily available. For example, in Java there exists the class `BigInteger` whose instances represent arbitrary precision integers. Note, that `BigInteger` is not a primitive type. It is tedious to use and leads to clumsy expressions. In Java Card type `BigInteger` is not available.

On the other hand, using an implementation language data type on the specification level—this approach is, for example, pursued in the Java Modeling Language (JML) [Leavens et al., 1999]—contradicts the idea that specifications should be abstract and hide implementation details. Humans think in terms of infinite (mathematical) types. It is, thus, not surprising that quite a number of JML specifications are inadequate, i.e., do not have the intended meaning [Chalin, 2003].[1] If not uncovered early, inadequate specifications are expensive to fix since they are at the root of the development process. Moreover, some implementation details may not even be known during specification, e.g., the implementation language or the concrete data types that are used. Obviously, an early specification containing such details is neither reusable nor comprehensive and hence more likely to be inadequate.

The pragmatic, engineering-oriented approach accepts the fact that the real world is not perfect, i.e., that the available types are not related by a refinement relation and tries to cope with it. It allows a *limited* and *controlled* incorrectness in the "refinement" steps.

The price we have to pay for allowing a limited incorrectness in the "refinement" steps, is that the implementation is not anymore automatically correct "by refinement." Rather, additional proofs are required. This approach, which can be seen as a generalisation of refinement, is an instance of Banach and Poppleton's *retrenchment* paradigm [Banach and Poppleton, 1998] which constitutes the basis for this section.

Before we briefly present refinement and retrenchment we shall introduce the semantics of the data types and their operations we consider in this chapter. The semantics of type `integer` and the usual operations like addition, subtraction, multiplication, etc. is clear (see Definition 3.18, Chapter 3); it corresponds to the semantics of the mathematical integers $\mathbb{Z}$ and we do not say anything about it. However, the semantics of integers in Java is not that obvious and is discussed in the next section.

12.2 Integer Types in Java

Java offers the four signed primitive integer data types `byte`, `short`, `int`, and `long` which have different but finite domains (see Table 12.1). We do

---

[1] To solve this problem, Chalin [2003] proposes to extend the JML, which does not support infinite integer types, with a type `bigInt` with infinite range. That, however, introduces into JML the problem of "incorrect refinement".

not consider the unsigned primitive type `char` in this chapter since there are some differences that prevent a uniform treatment of all primitive integer types. Nevertheless, in principle our approach can also be applied to `char`.

In the following we refer to the minimal and maximal value of a type T by $\mathtt{MIN}_T$ and $\mathtt{MAX}_T$, respectively. For example, the minimum value -128 of type `byte` is denoted by $\mathtt{MIN}_\mathtt{byte}$.

Table 12.1. Primitive signed Java integer types, the corresponding domain, and the number of bits needed for storing values

data type	domain	bits
byte	-128 to 127	8
short	-32768 to 32767	16
int	-2147483648 to 2147483647	32
long	-9223372036854775808 to 9223372036854775807	64

Internally, values of these types are stored in so-called signed two's complement which allows for efficient computation of arithmetical operations. If the result of an operation does not fit in the range of the expression type, i.e., the value cannot be represented using the provided number of bits, so-called *overflow*[2] occurs and the surplus number of bits are simply discarded by the Java virtual machine (for the technical details the reader is referred to [Schlager, 2002]). In mathematical terms this corresponds to computing the result modulo the size $-2 * \mathtt{MIN}_T$ of the data type T. The values of the Java types are, due to the negative values, non-standard representatives of the induced equivalence classes.[3] Therefore we have to normalise the value before applying the modulo function. Undoing the normalisation thereafter yields the correct result corresponding to the Java semantics. This is achieved by the family mod_T of functions defined as follows:

Definition 12.1. *For* $T \in \{\mathtt{byte}, \mathtt{short}, \mathtt{int}, \mathtt{long}\}$ *the family of functions* $mod_T : \mathtt{integer} \rightarrow \mathtt{integer}$ *is defined as*

$$mod_T(x) = (x - \mathtt{MIN}_T) \bmod (-2 * \mathtt{MIN}_T) + \mathtt{MIN}_T .$$

Example 12.2. The value 128 causes overflow in the Java domain of type `byte`. Applying function $mod_\mathtt{byte}$ yields

---

[2] The Java language specification [Gosling et al., 2000] distinguishes overflow and underflow for integer operation depending on the sign of the value. However, we have experienced that people are often misguided by the term "underflow" which is more commonly used in the context of floating point arithmetic with a different semantics. To avoid confusion we merely use the term "overflow" since the difference between overflow and underflow does not play a role in this chapter.

[3] The standard representatives for equivalence classes induced by modulo n are $0, 1, \ldots, n - 1$.

$$mod_{\texttt{byte}}(128) \;=\; (128-(-128)) \; mod \; (-2*(-128))+(-128) \;=\; -128 \; .$$

It is important to mention that the Java virtual machine does not indicate overflow in any way, e.g., by throwing an exception (as it is done in case of a division by zero). Some programming languages do indicate integer overflow by throwing an appropriate exception making debugging of programs containing errors related to overflow much easier. The concept of *checked* and *unchecked* execution contexts in the language C# even goes one step further: overflow in a checked context causes an exception being thrown whereas overflow in an unchecked context is treated as in Java.

Example 12.3. In the following C# code fragment variable `z` is set to zero if `x + y` causes overflow on type `short`.

```
── C# ──────────────────────────────────────────
try {
 // addition in checked context
  z = checked((short)(x + y));
} catch (System.OverflowException e) {
  z = 0;
}
──────────────────────────────────── C# ──
```

12.2.1 Implicit Type Casts (Numeric Promotion)

In Java the type of an arithmetic expression may be different from the types of the arguments, even if their types are equal. It is determined by so-called *unary* or *binary numeric promotion*, depending on the arity of the operator involved. The rules for numeric promotion are defined in the Java language specification but we shall briefly recall them here, omitting the cases for type `char`.

Binary numeric promotion is applied to arithmetic expressions with a binary operator (except for bit-operators). If (at least) one argument is of type `long`, the expression is of type `long`. Otherwise, i.e., if none of the arguments is of type `long` the result is of type `int`. Note, that the rules become more complex if one also considers floating point types which we ignore in this chapter.

Unary numeric promotion is applied to unary integer arithmetic expressions (except for bit-operators and pre- and postfix operators `++` and `--`) according to the following rule: If the type of the argument is `long` then the type of the unary expression is `long`, otherwise it is `int`.

Example 12.4. Let `b` be of type `byte` and let `1` be a numeric literal which is always of type `int` (unless the characters `l` or `L` are attached to it—indicating a literal of type `long`).

The type of expression `++b` is `byte` since, as mentioned above, unary promotion is *not* applied to increment and decrement operations. In contrast, as a result of binary numeric promotion the type of the expression `b+1` is `int` since the type of literal `1` is `int`. Thus, expression `++b` is not equivalent to `b+1` but to `(byte)(b+1)`. For example, if `b` has the value 127 both the expressions `++b` and `(byte)(b+1)` evaluate to -128 (overflow on `byte`) whereas `b+1` evaluates to 128 (no overflow on `int`).

12.2.2 Differences Between Java and Java Card

Java Card does not offer the primitive type `long` and availability of type `int` depends on the implementation of the Java Card virtual machine. If `int` is not supported the virtual machine rejects programs containing that type. However, even if type `int` does not occur explicitly in a Java program it may still be used to store intermediate results of computations. In the code fragment

```
Java
short a = 32767;
short b = 1;
short c = 2;
a = (short) ((a + b) / c);
                                                    Java
```

the Java virtual machine implicitly evaluates expression `a + b` on type `int` (as explained in Section 12.2.1) yielding a result of 32768. The subsequent division evaluates to 16384. The final cast operation to `short` does not have any effect here and is only required to ensure type correctness.

A Java Card virtual machine that does not support `int` cannot perform implicit type casts to `int` in order to store intermediate results. As a consequence, overflow may already occur on type `short`. If the same program is executed on a Java Card virtual machine without support for `int` the intermediate result of `a + b` causes overflow and results in -32768. Evaluating division and cast operations finally yields a result of -16384.

The previous example contradicts the rule mentioned in [Chen, 2000] that any arithmetic operation should render the same result when executed on a Java virtual machine and on a Java Card virtual machine. To prevent such discrepancies in computation typically the converter, which transforms Java byte code into Java Card byte code, reports an error if a program contains expressions like the one above.

For a uniform treatment of Java and Java Card, in the following we assume that implicit type casts performed by the Java Card virtual machine but *not* by the Java virtual machine are made explicit in the source code by adding appropriate type casts. This assumption also guarantees that the converter does not abort with errors mentioned above.

12.3 Refinement and Retrenchment

In this section we shall give a lightweight introduction on refinement and retrenchment as far as required for this chapter.

12.3.1 Preliminaries

In Section 3.3 we defined the semantics of a program as a binary relation between the initial and final program state. Input and output of a program is considered being part of the program state. In this section we are not interested in internal states of programs but merely in input and output of a program. We therefore make input and output explicit and call the program an operation.

Definition 12.5 (Operation, Termination). *Let S be a set of states, Input a set of possible inputs, and Output a set of possible outputs. An* operation

$$Op \ \subseteq \ S \times Input \times S \times Output$$

is a relation on states and input/output values. We write $s[\![out = Op(in)]\!]s'$ iff $(s, in, s', out) \in Op$.

Operation Op started in a state $s \in S$ with input in terminates iff there exists a state $s' \in S$ and output out such that $s[\![out = Op(in)]\!]s'$. This is denoted by $s \models Op(in)\downarrow$.

Without loss of generality, we only consider operations that have output. Operations without output are considered to return an arbitrary value. For non-deterministic programming languages the above definition of termination expresses that there is a possibility for the operation to terminate and is thus similar to the semantics of the diamond modality (see Definition 3.34, Chapter 3). For a deterministic language like JAVA CARD, $s \models Op(in)\downarrow$ means that the operation always terminates when started in s.

Specifications of operations are as usual pairs of pre- and postcondition. But, since it is more convenient in this chapter, pre- and postconditions are not given as (OCL, JML, or JAVA CARD DL) formulae as in Chapter 5 but as their denotations, i.e., as the sets of states and input and output values satisfying the corresponding formulae.

Definition 12.6 (Operation Specification). *Given sets S of states, Input of input, and Output of output values, an* operation specification *is a pair $(Pre, Post)$ of predicates*

$$Pre \subseteq S \times Input \ \ and \ \ Post \subseteq S \times Input \times S \times Output \ .$$

The correctness of an operation with respect to a given specification is defined as usual, i.e., if an operation satisfies the precondition then it has to satisfy the postcondition.

Definition 12.7 (Operation Correctness). *Given sets S of states, Input of input, Output of output values, and an operation specification $(Pre, Post)$. An operation $Op \subseteq S \times Input \times S \times Output$ is correct w.r.t. $(Pre, Post)$ iff, for all $(s, in, s', out) \in Op$,*

$$(s, in) \in Pre \;\; implies \;\; (s, in, s', out) \in Post \; .$$

The above definition constrains only terminating runs of an operation, i.e., reflects the partial correctness semantics mentioned in Section 5.2.3.

12.3.2 Refinement

Very generally speaking, refinement is a relation between two systems, an abstract system and a concrete one. The concrete system $\mathcal{C}$ is said to refine (or to implement) the abstract system $\mathcal{A}$ if $Pre_\mathcal{A} \subseteq Pre_\mathcal{C}$ and $Post_\mathcal{C} \subseteq Post_\mathcal{A}$, i.e., if the precondition of the concrete system is equal or less restrictive than the one of the abstract system and vice versa for the postcondition. This means a refined system can at least be used where the abstract system can be used and always yields a result that is compatible with the result of the abstract system.

Example 12.8. Suppose a resource manager that allocates identical but numbered resources to clients.

Upon request an abstract resource manager assigns an arbitrary but free resource to the client.

A concrete refined manager of this abstract manager (which can also be considered as the specification) could be a system that always allocates the free resource with the smallest number (thus eliminating non-determinism). Additionally, the refined system could also manage resources that are released by the client so that they can be re-allocated. This is possible since a refined system may do more than the abstract one.

Refinement (and also retrenchment) in general allows the operations of abstract and concrete data type to act on different state spaces. A *retrieve relation* connects abstract and concrete states, i.e., defines which abstract states are represented by which concrete states.

Example 12.9. Suppose the abstract resource manager from Example 12.8 uses a *set* of integers to manage the available free resources.

The concrete manager uses an *array* instead of a set (since this makes it easier to find the free resource with the smallest number).

However, the order of the elements in the array is not important and a reasonable retrieve relation could identify a state of the abstract system with all states of the concrete system where the elements of the array are a permutation of the elements in the set.

The above example already suggests that refinement is not restricted to systems but can also be applied to data types and operations. Intuitively, an operation adding an element to an array is a refinement of an operation adding an element to a set only if it makes sure that the element to be added in not already contained in the array. This means it must be guaranteed that the state of the concrete data type is related (via the retrieve relation) to the state of the abstract data type after every operation.

The integer data type which we consider in this chapter is particular in the sense that it does not have a state. Hence the operations do not perform a state transition but merely compute an output for the given input. As a consequence, we neither have to consider different state spaces nor a retrieve relation. Therefore we obtain simpler definitions of so-called *operation refinement* and later *operation retrenchment*.

Definition 12.10 (Operation Refinement). *An operation Op_{concr} is an operation refinement of an operation Op_{abstr} (over the same state space S) iff, for all states s, s', all input values in such that $s \models Op_{abstr}(in)\!\downarrow$, and all output values out,*

$$if \ s[\![out = Op_{concr}(in)]\!]s' \ then \ s[\![out = Op_{abstr}(in)]\!]s' \ .$$

This definition states that the abstract operation can do anything (and possibly more) the concrete operation can do. For example, this means that the concrete operation eliminates non-determinism.

It is important to mention that (operation) refinement does not allow input and output of abstract and concrete operations to differ. Not least because of this it is obvious that the JAVA integers are not a refinement of the mathematical integers $\mathbb{Z}$. Furthermore, operations on JAVA integers may yield a result different from the result on $\mathbb{Z}$ (see the example in Section 12.1).

12.3.3 Retrenchment

In practical system design, situations where abstract types cannot be correctly refined by more concrete types occur quite often. This observation was the motivation for several liberalisations of the strict refinement notion like e.g., IO refinement [Boiten and Derrick, 1998] which allows input and output of abstract and concrete operations to differ. Retrenchment [Banach and Poppleton, 1998] constitutes the most radical generalisation of refinement which, in principle, allows to relate arbitrary abstract and concrete data types and operations.

We now give the definition of *operation retrenchment* which is a special case of retrenchment adapted to our setting:

Definition 12.11 (Operation Retrenchment). *A concrete operation*

$$Op_{concr} \subseteq S \times Input_{concr} \times S \times Output_{concr}$$

is an operation retrenchment *of an abstract operation*

$$Op_{abstr} \subseteq S \times Input_{abstr} \times S \times Output_{abstr}$$

(over the same state space S) via

- *a within relation* $W \subseteq Input_{abstr} \times Input_{concr} \times S$,
- *a concedes relation* $C \subseteq S \times S \times Output_{abstr} \times Output_{concr}$,
- *and an output relation* $O \subseteq Output_{abstr} \times Output_{concr}$,

iff,
for all states s, s', input values in_{abstr}, in_{concr} such that $s \models Op_{abstr}(in_{concr})\downarrow$,
and output values out_{concr},

> *if* $s[\![out_{concr} = Op_{concr}(in_{concr})]\!]s'$ *and* $W(in_{abstr}, in_{concr}, s)$,
> *then there exists an output value* out_{abstr} *with*
> $s[\![out_{abstr} = Op_{abstr}(in_{abstr})]\!]s'$ *and*
> $(O(out_{abstr}, out_{concr})$ *or* $C(s, s', out_{abstr}, out_{concr}))$.

The formal definition of operation retrenchment contains three relations, called *within* relation W, *concedes* relation C, and *output* relation O.

The within relation W is used to limit the set of states and inputs for which the relationship between abstract and concrete operations needs to be established. In contrast, refinement requires to establish the relation for every pair of state and input/output.

The output and concedes relation O and C allow to restrict abstract and concrete output values. Usually, the output relation is used to define the "normal" case, whereas the concedes relation defines exceptional behaviour. This is of course not prescribed by the formal definition since O and C are disjunctively connected relations on the output values. However, as we see later, separating the normal and the exceptional case in O and C simplifies setting up proof rules for verifying the correctness of a retrenchment considerably.

If we define $W \equiv \{(in_{abstr}, in_{concr}, s) \mid in_{abstr} = in_{concr}\}$, i.e., exclude from consideration cases where abstract and concrete input differ, $C \equiv \emptyset$, and $O \equiv \{(out_{abstr}, out_{concr}) \mid out_{abstr} = out_{concr}\}$ then Definitions 12.10 and 12.11 coincide, i.e., operation refinement is indeed a special case of operation retrenchment.

The advantage of casting non-refinement steps into the retrenchment framework is that it becomes explicit where exactly the refinement conditions are violated, namely where the within relation W does not hold or where the concedes relation C becomes true. In these cases correctness cannot be shown once and for all (as it is the case with correct refinement). Correctness of a program containing retrenchment is shown by individually verifying these critical situations. Note, that this requires additional proofs which are still done at proof time. After these proofs have been done, no run-time checks are required.

The additional proof obligations arise from the conditions the following correctness theorem is based on. The theorem has been proven in a slightly different form in [Beckert and Schlager, 2005].

Theorem 12.12 (Correctness w.r.t. Retrenchment). *Let Op_{abstr} be an operation that is correct with respect to a given specification $(Pre, Post)$. Let Op_{concr} be an operation that is an operation retrenchment of Op_{abstr} via relations W, C, and O (as in Definition 12.11) and let $Input_{concr} = Input_{abstr}$ and $Output_{concr} = Output_{abstr}$.*

Then Op_{concr} is correct with respect to $(Pre, Post)$ if for all inputs in_{concr} and all states s with $(s, in_{concr}) \in Pre$

1. *in case that $(in_{abstr}, in_{concr}, s) \in W$ for some input value in_{abstr} (cases where the within relation holds),*

$$O(out_{abstr}, out_{concr}) \quad or \quad C(s, s', out_{abstr}, out_{concr})$$
$$implies$$
$$\qquad in_{abstr} = in_{concr} \quad and \quad out_{abstr} = out_{concr}$$
$$\qquad or$$
$$\qquad (s, in_{concr}, s', out_{concr}) \in Post$$
$$for\ all\ states\ s'\ and$$
$$output\ values\ out_{concr}\ with\ s[\![out_{concr} = Op(in_{concr})]\!]s'\ and$$

2. *in case that $(in_{abstr}, in_{concr}, s) \notin W$ for all input values in_{abstr} (cases excluded by the within relation),*

$$(s, in_{concr}, s', out_{concr}) \in Post$$
$$\qquad for\ all\ states\ s'\ and$$
$$\qquad output\ values\ out_{concr}\ with\ s[\![out_{concr} = Op(in_{concr})]\!]s'\ .$$

The above theorem states under which conditions a retrenched operation is correct under the assumption that the abstract operation is correct with respect to the given specification:

- If there exists an input in_{abstr} for in_{concr} such that W holds, i.e., the case is of interest for the retrenchment, then either the postcondition holds immediately or O and C have to imply that abstract and concrete input and output are equal. Then the correctness of the concrete operation follows from the correctness of the abstract one.
- The case is excluded from consideration, i.e., for any in_{abstr} W does not hold. Then both abstract and concrete operation are unrelated and the correctness of the concrete operation has to be established by showing that the postcondition holds for this particular situation.

As compared to refinement, when retrenchment is used to establish correctness of the concrete operation Op_{concr}, the additional Conditions (1) and (2) in Theorem 12.12 have to be proven. Note that, although Condition (1),

which handles cases for which the within relation W is true, looks more complicated than Condition (2.), it is actually the "harmless" one. In particular, if the definitions

$$W \equiv \{(in_{abstr}, in_{concr}, s) \mid in_{abstr} = in_{concr}\}$$
$$O \equiv \{(out_{abstr}, out_{concr}) \mid out_{abstr} = out_{concr}\}$$
$$C \equiv \emptyset$$

for the within, the output, and the concedes relation are used, Condition (1.) is trivially true for all cases where it applies. Even if other definitions for W, O, and C are used, the particular postcondition *Post* has rarely to be considered. On the other hand, checking Condition (2.) always involves the particular postcondition and, thus, the particular specification. To summarise, if W, O, and C are well chosen, Condition (1.) it trivially true, or at least can be proven once and for all, whereas Condition (2.) has to be proven for each particular specification.

12.4 Retrenching Integers in KeY

In this section we demonstrate how retrenchment can be used to exactly describe the relation between infinite integers `integer` in JAVA CARD DL and finite integers in JAVA. We shall consider the following operations available on both data types.

Definition 12.13. *Let S be an arbitrary set of states. Then the (abstract) operations*

$$Op_{abstr,\circ} \subseteq S \times (\texttt{integer} \times \texttt{integer}) \times S \times \texttt{integer}$$

for $\circ \in \{+, -, *, /, \%\}$ are defined by

$$s[\![out = Op_{abstr,\circ}(\langle in_1, in_2 \rangle)]\!]s' \quad \textit{iff} \quad s = s' \ \textit{and} \ out = in_1 \circ in_2$$

for all states s, s', input values $\langle in_1, in_2 \rangle$ and output values out.

Note, that we exclude bit-wise operations from consideration since they are not defined on `integer`. For the defined operations we discuss the two retrenchments implemented in the KeY system. As concrete retrenched types we consider the JAVA type `int` but everything holds analogously as well for `byte`, `short`, and `long`.

In the first retrenchment R_{Java} that is described in Section 12.4.1 it is shown that, for those cases violating the refinement conditions, the "refined" data type operations are sufficient (this amounts to weakening the postcondition).

For the example of addition, one has to show for the particular situation that using $x\texttt{+}_{\texttt{int}}y$ instead of $x +_{\texttt{integer}} y$ does not lead to incorrectness, i.e.,

that the property to be proven is independent of the used type (see the example in Section 12.4.1).

In the second retrenchment R_{KeY} (Section 12.4.2) cases violating the refinement conditions are shown not to occur. This amounts to strengthening the preconditions for the invocation of the "refined" data type operations.

Considering as an example the operations $\mathtt{+_{int}}$ and $\mathtt{+_{integer}}$, one has to show for each individual invocation of $\mathtt{x+y}$ in a JAVA CARD program that the result does not exceed the (finite) range of $\mathtt{int}$, i.e., does not lead to overflow.

The KeY system supports both R_{Java} and R_{KeY}. However, we argue that the second possibility R_{KeY} is the better choice and strongly suggest its use.

12.4.1 Retrenchment R_{Java} by Weakening the Postcondition

Definition 12.14 formalises the semantics of the operations on type $\mathtt{int}$ as defined in the JAVA language specification, i.e., the operations do not modify the state and the result is calculated modulo the size of the type of the expression T.

Definition 12.14. *The (concrete) operations*

$$Op^{Java}_{concr,\circ} \subseteq S \times (\mathtt{integer} \times \mathtt{integer}) \times S \times \mathtt{integer}$$

for $\circ \in \{\{+,-,*,/,\%\}$ are defined by

$$s[\![out = Op^{Java}_{concr,\circ}(\langle in_1, in_2 \rangle)]\!]s' \quad iff \quad s = s' \ and \ out = mod_T(in_1 \circ in_2)$$

for all states s, s', input values $\langle in_1, in_2 \rangle$ and output values out, where T is the type of $in_1 \circ in_2$ determined by binary numeric promotion.

The following theorem provides the within, output, and concedes relations for which the operations $Op^{Java}_{concr,\circ}$ are retrenchments of $Op_{abstr,\circ}$ ($\circ \in \{+,-,*,/,\%\}$). This retrenchment is called R_{Java} in the sequel.

Theorem 12.15. *For every $\circ \in \{+,-,*,/,\%\}$, the operation $Op^{Java}_{concr,\circ}$ is an operation retrenchment of $Op_{abstr,\circ}$ via the relations defined by:*

$$
\begin{aligned}
W &\equiv \{(in_{abstr}, in_{concr}, s) \mid in_{abstr} = in_{concr}\} \\
O &\equiv \{(out_{abstr}, out_{concr}) \mid out_{abstr} = out_{concr}\} \\
C &\equiv \{(s, s', out_{abstr}, out_{concr}) \mid s = s' \ and \ out_{concr} = mod_T(out_{abstr}) \\
&\qquad\qquad and \ out_{abstr} \neq out_{concr}\}
\end{aligned}
$$

Note, that the condition $out_{abstr} \neq out_{concr}$ in the above definition of C could as well be omitted. However, making it explicit has the advantage of making the concedes relation C and the output relation O disjoint. This allows to clearly distinguish between "normal" behaviour (O holds) and "exceptional" behaviour (C holds).

This retrenchment says that we only take into account cases where both abstract and concrete input values are equal. Then, either no overflow occurs and both output values have to be equal (output relation O) or overflow occurs and the concrete output out_{concr} is equal to the result of applying function mod_T to out_{abstr} (concedes relation C).

Example of Retrenchment R_{Java}

The following example illustrates the effects of this retrenchment concerning the correctness of a Java implementation with respect to specification using the integers `integer`.

Assume an operation $addTwo$ that is supposed to increase the argument by two and a specification for $addTwo$ consisting of the precondition $even(x)$ and the postcondition $even(result)$, where $result$ denotes the output of the operations. The Java implementation of $addTwo$ is as follows:

──── Java ────
```java
int addTwo(int x) {
  return x+2;
}
```
──────────────────────────────────── Java ────

If we assume that the Java type `int` coincides with the mathematical integers `integer`, it is obvious that the implementation satisfies the specification.

But what happens if we consider the true semantics of `int` instead, i.e., the retrenchment R_{Java} from above? Obviously, for every input $in_{concr} \in$ `int` there is an input $in_{abstr} \in$ `integer` with $in_{abstr} = in_{concr}$ (since the domain of `int` is a subset of `integer`), i.e., an abstract value for which W holds always exists. Thus, we can forget about the proof obligation from Condition (2.) in Theorem 12.12, and we only have to consider Condition (1.) assuming $in_{abstr} = in_{concr}$. Then Condition (1.) reduces to:

$$in_{concr} +_{\texttt{int}} 2 = in_{abstr} +_{\texttt{integer}} 2 \text{ or}$$
$$in_{concr} +_{\texttt{int}} 2 = mod_{\texttt{int}}(in_{abstr} +_{\texttt{integer}} 2) \text{ and}$$
$$in_{concr} +_{\texttt{int}} 2 \neq in_{abstr} +_{\texttt{integer}} 2$$
implies
$$in_{abstr} = in_{concr} \text{ and } in_{abstr} +_{\texttt{integer}} 2 = in_{concr} +_{\texttt{int}} 2 \text{ or}$$
$$even(in_{concr} +_{\texttt{int}} 2).$$

We assume that the precondition $even(in_{concr})$ holds and consider the left-hand side of the implication.

The first disjunct arises from the output relation O and constitutes the "normal" case where the concrete operation yields the same result as the abstract operation. Together with the assumptions it obviously implies the (first disjunct of the) right-hand side of the implication.

The second disjunct corresponds to the concedes relation of the retrenchment, which says that the result of the concrete operation is equal to the result of the abstract operation modulo $-2 * \mathtt{MIN_{int}}$, i.e., this is the case where the concrete operations does not correctly simulate the abstract one. Incidentally, in our example the size $-2 * \mathtt{MIN_{int}}$ of $\mathtt{int}$ is even and, thus, also $(x + 2)\%(-2 * \mathtt{MIN_{int}})$ is even. This means that, even if the concrete operation has a different behaviour than the abstract one the postcondition is still satisfied, i.e., the implementation is correct.

Assessment of Retrenchment R_{Java}

As the above example shows, using R_{Java} it is possible for a program to be correct even if it contains expressions such that abstract and concrete operations yield different results.

In general, if the correctness of a program has been established using retrenchment Theorem 12.12 via a necessarily non-trivial concedes relation C (i.e., the theorem does not hold any more if C is replaced by false) we call the program *incidentally correct.*

We think the term "incidental correctness" is justified since "luckily" the postcondition holds even if the concrete type yields a result different from the abstract result. A user who is not aware of that may think that the same automatically holds true for other postconditions, which it does not.

Programmers tend to use retrenched types as if they were refined types, i.e., the fact that there is a non-trivial concedes relation is ignored. And in fact, the concedes relation often does not play a role since the concrete program solely works on the part of the domain where the concedes relation is not required (most JAVA programs do not contain overflow). However, for critical applications we cannot trust our luck and hope that a program does not "make use" of the concedes relation. If it does, there are two possibilities. First, the concedes relation does not entail the postcondition. Then the program is in fact incorrect. Second, the concedes relation implies the postcondition, in which case the program is correct.

Still, we argue that the second case may cause problems in an ongoing development process. The reason is that the program can run into situations where the concrete type has a different behaviour than the abstract one, though the specification still holds in these exceptional cases. Since the program behaves correctly it may remain hidden from the programmer that the program runs into exceptional situations. Only an inspection of the correctness proof would reveal the fact the concedes relation is actually used. If the developer does not do that (the proof may be constructed automatically or by someone else), the true behaviour may diverge from the developer's intuition and understanding of the program. This is dangerous in an ongoing software project, where programs and even specifications are often modified. Then, a wrong understanding of the internal program behaviour and the fact

that the particular concedes relation may not work for other postconditions easily leads to errors that are hard to find.

12.4.2 Retrenchment R_{KeY} by Strengthening the Precondition

The second retrenchment R_{KeY} is based on strengthening the precondition of the operations to ensure that the operations only act on the parts of the domain where `int` actually behaves like `integer`, i.e., where no overflow occurs. This results in additional proof obligations stating that the result of an operation does not exceed the (finite) range of the expression type. By verifying these additional proof obligations, we establish that the programming language types are only used to the extent that they indeed are a refinement of the specification language types (the non-refinement cases do not occur). As already said, this check cannot be done once and for all, as it is the case for a correct refinement, but has to be repeated for each particular specification and program. Since this is tedious and error-prone if done by hand the generation of the additional proof obligation is integrated into the verification calculus of the KeY system (see Section 12.5.1).

The operations $Op^{KeY}_{concr,\circ}$ ($\circ \in \{+, -, *, /, \%\}$) that we define for the second retrenchment differ from the abstract operations only on those parts of the input domain `integer` $\times$ `integer` where the result of the operation would exceed the bounds $\mathtt{MIN}_T$ or $\mathtt{MAX}_T$. For these cases the semantics of $Op^{KeY}_{concr,\circ}$ is not specified, i.e., it is not guaranteed to terminate and in case of termination it returns an arbitrary value.

Definition 12.16. *The family $Range_T \subseteq$ `integer` of predicates is defined by*

$$Range_T(x) \quad \textit{iff} \quad \mathtt{MIN}_T \leq x \leq \mathtt{MAX}_T \ ,$$

where T is the type of expression x.

Definition 12.17. *For $\circ \in \{+, -, *, /, \%\}$, the concrete operation*

$$Op^{KeY}_{concr,\circ} \subseteq S \times (\texttt{integer} \times \texttt{integer}) \times S \times \texttt{integer}$$

is defined, for all states s, s', input values $\langle in_1, in_2 \rangle$, and output values out, by

$$s[\![out = Op^{KeY}_{concr,\circ}(\langle in_1, in_2 \rangle)]\!]s' \quad \textit{iff}$$
$$\textit{(i)} \quad s = s' \ ,$$
$$\textit{(ii)} \quad out = in_1 \circ in_2 \ , \textit{ and}$$
$$\textit{(iii)} \ Range_{T_1}(in_1) \ \textit{and} \ Range_{T_2}(in_2) \ \textit{implies} \ Range_T(in_1 \circ in_2) \ .$$

where T_1, T_2, T is the type of in_1, in_2, and $in_1 \circ in_2$ (for the latter determined by binary numeric promotion), respectively.

Analogous to Section 12.4.1 we provide the within, the concedes, and the output relation for which the operations $Op^{KeY}_{concr,\circ}$ are retrenchments of $Op_{abstr,\circ}$ ($\circ \in \{+, -, *, /, \%\}$). This retrenchment is called R_{KeY} in the following.

Theorem 12.18. *For every* $\circ \in \{+, -, *, /, \%\}$, *the operation* $Op^{KeY}_{concr,\circ}$ *is an operation retrenchment of* $Op_{abstr,\circ}$ *via the relations defined by:*

$$W \equiv \{(in_{abstr}, in_{concr}, s) \mid in_{abstr} = in_{concr} \text{ and}$$
$$Range_{T_1}(in_{concr,1}) \text{ and } Range_{T_2}(in_{concr,2})$$
$$implies\ Range_T(in_{concr,1} \circ in_{concr,2})\}$$
$$O \equiv \{(out_{abstr}, out_{concr}) \mid out_{abstr} = out_{concr}\}$$
$$C \equiv \emptyset$$

where T_1, T_2, T *is the type of* $in_{concr,1}, in_{concr,2}$, *and* $in_{concr,1} \circ in_{concr,2}$ *(for the latter determined by binary numeric promotion), respectively.*

The within relation W states that we only consider cases where both abstract and concrete inputs are equal and the concrete operation does not cause overflow. Assuming W holds we have to show that O or C holds (Condition (1) in Theorem 12.12). Since $C \equiv \emptyset$ this amounts to showing that O holds, i.e., that the outputs are also equal. This retrenchment does not say anything about the case where overflow occurs, i.e., where the within relation does not hold (Condition (2) in Theorem 12.12). Thus, to ensure correctness of an implementation we have to ensure that W always holds. In the calculus of the KeY system this is achieved by generating proof obligations to establish W for every occurrence of an arithmetical operation $Op^{Java}_{concr,\circ}$ (consider for example the first premiss of rule assignmentAdditionToUpdateCheckingOF in Section 12.5.1).

12.5 Implementation

The implementation of the two retrenchments presented in Section 12.4.1 and 12.4.2 in the KeY system is based on different rules sets which can be activated by the user in the "Taclet Options Defaults" dialog in the options menu of the KeY prover shown in Figure 12.1.

The following rule sets are available:

- intRules : javaSemantics contains rules for integer arithmetic based on retrenchment R_{Java} defined in Section 12.4.1, i.e., faithfully reflects the semantics defined in the JAVA language specification.
- intRules : arithmeticSemanticsCheckingOF covers the rules for retrenchment R_{KeY} from Section 12.4.2, i.e., overflow is not allowed to occur.
- intRules : arithmeticSemanticsIgnoringOF which ignores the problems arising from the different semantics of (finite) programming language data types and (idealistic) specification language data types. That is, in these
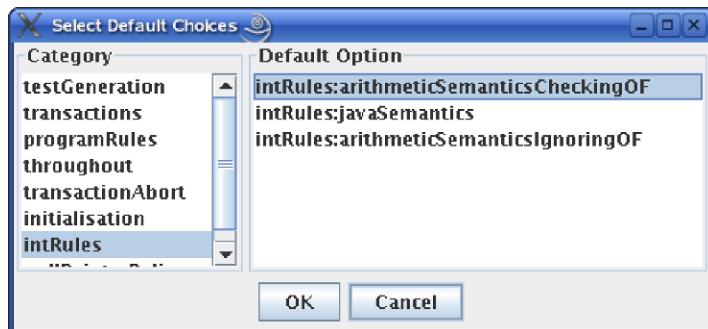
Fig. 12.1. Dialog for choosing integer semantics in the KeY system

rules the Java integers are (wrongly) assumed to implement the semantics of the mathematical integers $\mathbb{Z}$. As a consequence, using this rule set a closed proof does in general not imply the correctness of the implementation with respect to a specification since the rules do not correctly reflect the Java semantics.

Which rule set, i.e., which retrenchment, should be chosen depends on the requirement of the level of dependability one wants to guarantee. Functional correctness can be achieved only using the first two of the options above. The rule set intRules : arithmeticSemanticsCheckingOF in addition ensures that the verified program does not contain overflow.

The rule set intRules : arithmeticSemanticsIgnoringOF does not guarantee functional correctness. Nevertheless, since the calculus rules are simpler this semantics can be used for a first verification attempt if the main focus is different from functional correctness.

12.5.1 Sequent Calculus Rules

In the following rule schemata, *var* is a local program variable (of an arithmetical type) whose access cannot cause side-effects. For expressions with potential side-effects (like, e.g., an attribute access that might cause a `NullPointerException`) the rules for symbolic execution cannot be applied and program transformation rules that evaluate the complex expression and assign the result to a new local variable have to be applied first. Similarly, *se* satisfies the restrictions on *var* as well or it is an integer literal (whose evaluation is also without side-effects). There is no restriction on *expr*, which is an arbitrary Java expression of a primitive integer type (its evaluation may have side-effects).

Program Transformation Rules

Program transformation rules are independent of the chosen integer semantics since they transform complex expressions into a sequence of simpler but

semantically equivalent expressions. As examples we present rules for handling postfix increment expressions and compound assignments.

The Rule for Prefix Increment

This rule transforms a prefix increment into a normal JAVA addition.

$$\text{preIncrement} \quad \frac{\implies \langle\pi \;\; var=(T)(var+1)\,; \;\; \omega\rangle\phi}{\implies \langle\pi \;\; \text{++}var\,; \;\; \omega\rangle\phi}$$

T is the declared type of var. The explicit type cast is necessary to preserve the semantics since the arguments of + are implicitly cast to int or long by binary numeric promotion which is not the case for the prefix increment operator ++ (see Section 12.2.1).

The Rule for Compound Assignment

This rule transforms a statement containing the compound assignment operator += into a semantically equivalent statement with the simple assignment operator = (again, T is the declared type of var).

$$\text{compoundAssignmentUnfold} \quad \frac{\implies \langle\pi \;\; var=(T)(var+expr)\,; \;\; \omega\rangle\phi}{\implies \langle\pi \;\; var\text{+=}expr\,; \;\; \omega\rangle\phi}$$

For the soundness of both rules it is essential that var does not have side-effects because var is evaluated twice in the premises and only once in the conclusions.

Rules for Symbolic Execution

In contrast to program transformation rules, we here have different rules for retrenchments R_{Java} and R_{KeY}.

The Addition Rule for Retrenchment R_{Java}

$$\text{assignmentAdditionToUpdateJavaSemantics} \quad \frac{\implies \{var := modulo_T(se_1 + se_2)\}\, \langle\pi\,\omega\rangle\phi}{\implies \langle\pi \;\; var=se_1\text{+}se_2\,; \;\; \omega\rangle\phi}$$

T is the type of the expression $se_1\text{+}se_2$ which is determined by binary numeric promotion. Depending on T the schematic function $modulo_T$ is instantiated with the corresponding JAVA CARD DL function symbols $moduloByte$, $moduloShort$, $moduloInt$, $moduloLong$, or $moduloChar$ (see Appendix A.1.2) which have the same semantics as the meta-level function mod_T.

The Addition Rule for Retrenchment R_{KeY}

assignmentAdditionToUpdateCheckingOF
$$\frac{inRange_{T_1}(se_1), inRange_{T_2}(se_2) \Rightarrow inRange_T(se_1 + se_2)}{\Rightarrow \{var := se_1 + se_2\} \langle \pi\, \omega \rangle \phi}$$
$$\Rightarrow \langle \pi\ var{=}se_1{+}se_2\texttt{;}\ \omega \rangle \phi$$

T_1, T_2, T are types of se_1, se_2, and $se_1{+}se_2$, respectively. Depending on these types the schematic predicate $inRange_T$ is instantiated with the corresponding Java Card DL predicate *inByte,inShort,inInt,inLong*, or *inChar* (see Appendix A.1.5) which have the same semantics as the meta-level predicate $Range_T$.

The first premiss establishes the within relation W of the retrenchment R_{KeY} (see Theorem 12.18). If both arguments are within the provided range, then the result must be within range as well, i.e., no overflow occurs.

In the second premiss the Java statement is symbolically executed, i.e., the statement disappears from the program and is translated into a state update.

Note, that the rule contains two different symbols for addition with different semantics: The symbol "+" occurring in the conclusion of the rule denotes the Java operation. The symbol "+", which occurs in the two premisses, represents the abstract addition operation on `integer`.

Obviously, completeness is lost using retrenchment R_{KeY} since attempting to verify any program containing overflow results in proof goals that cannot be closed. However, we discourage from making use of overflow and believe that the loss of completeness is not a disadvantage. Moreover, if overflow cannot be excluded for some reason the rule set based on retrenchment R_{Java} can be used which is complete (also for programs containing overflow).

12.5.2 Example

As an example we want to prove that `++b` is in fact equivalent to `(byte)(b+1)` (see Example 12.4) and set up the following proof obligation:

$$b \doteq c \Longrightarrow \langle \texttt{boolean r=(++b==(byte)(c+1));} \rangle r \doteq \text{TRUE} \qquad (12.1)$$

Firstly, by choosing the integer semantics *intRules:javaSemantics* in the taclet options we activate the rule set reflecting retrenchment R_{Java} where overflow is handled as defined in the Java language specification.

After some applications of program transformation rules breaking up the quite complex expression from above we obtain

$$b \doteq c \Longrightarrow \langle \texttt{byte i0=++b; byte i1=(byte)(c+1); r=i0==i1;} \rangle r \doteq \text{TRUE}$$

As one can see fresh variables `i0` and `i1` of type `byte` are introduced and assigned the result of the left and respectively right side of the complex comparison expression from above.

After removing the declaration of the fresh variable `i0` rule preIncrement can be applied yielding

$$\text{b} \doteq \text{c} \Longrightarrow \backslash\!\langle\; \texttt{b=(byte)(b+1); i0=b;}$$
$$\texttt{byte i1=(byte)(c+1); r=i0==i1;} \backslash\!\rangle \text{r} \doteq \text{TRUE} \qquad (12.2)$$

Again, some program transformations are applied and eventually rule assignmentAdditionToUpdateJavaSemantics which gives

$$\text{b} \doteq \text{c} \Longrightarrow \{\texttt{i0} := moduloByte(addJint(\text{b}, 1))\}$$
$$\langle\texttt{byte i1=(byte)(c+1); r=i0==i1;}\rangle \text{r} \doteq \text{TRUE} \qquad (12.3)$$

As one can see the JAVA statements `b=(byte)(b+1); i0=b;` have been worked off resulting in the update $\{\texttt{i0} := moduloByte(addJint(\text{b}, 1))\}$.

Then the proof proceeds similar to above and we obtain

$$\text{b} \doteq \text{c} \Longrightarrow \{\; \texttt{i0} := moduloByte(addJint(\text{b}, 1)),$$
$$\texttt{i1} := moduloByte(addJint(\text{c}, 1))\}\langle\texttt{r=i0==i1;}\rangle\text{r} \doteq \text{TRUE}$$

We make use of the equation $\text{b} \doteq \text{c}$ in the antecedent and eventually finish the proof after a few simple steps.

In a second attempt we try to discharge the same proof obligation (12.1) using the integer semantics *intRules:arithmeticSemanticsCheckingOF*, i.e., using the rule set reflecting retrenchment R_{KeY}.

The proof is similar to the one above until we reach goal (12.2). After some simplifications, instead of rule assignmentAdditionToUpdateJavaSemantics we now have to apply rule assignmentAdditionToUpdateCheckingOF and a rule for type casts (not shown here) which results in the following three new goals:

$$\text{b} \doteq \text{c} \;\; \Longrightarrow \{\texttt{i0} := \text{b} + 1\} \qquad\qquad\qquad\qquad (12.4)$$
$$\langle\texttt{byte i1=(byte)(c+1); r=i0==i1;}\rangle\text{r} \doteq \text{TRUE}$$
$$\text{b} \doteq \text{c} \;,\; inByte(\text{b}),\; inByte(1) \Longrightarrow inInt(\text{b} + 1) \qquad (12.5)$$
$$\text{b} \doteq \text{c} \;\; \Longrightarrow inByte(\text{b} + 1) \qquad\qquad\qquad\qquad (12.6)$$

Goal (12.4) corresponds to (12.3) but here we have addition $+$ on `integer` instead of the function *addJint* from above. Furthermore, instead of the cast function *moduloByte* we here have the additional goal (12.6) ensuring that the cast does not cause overflow and, thus, has no effect. Goal (12.5) guarantees that the expression $\text{b} + 1$ does not cause overflow. Obviously, both sequents (12.5) and (12.6) are not valid, i.e., do not holds in all states—e.g., if `b` has the value 127 both sequents do not hold. If we continue the proof with (12.4) we obtain two similar proof obligations for `c` that cannot be discharged. The reason for that is, that retrenchment R_{KeY} requires a program to be free of overflow for all possible input values. To make the above program provable in R_{KeY} we therefore have to restrict the possible values of `b` and `c` to prevent overflow, e.g., by adding a precondition $\text{b} \geq 0 \wedge \text{b} < 100$ which

then also restricts the range of c since we have $b \doteq c$. We obtain the valid sequent

$$b \doteq c, b \geq 0, b < 100 \Longrightarrow \langle \texttt{boolean r=(++b==(byte)(c+1));} \rangle r \doteq \text{TRUE}$$

that can be derived with the KeY prover using the integer semantics checking for overflow.

12.6 Pitfalls Related to Integers

In addition to problems related to the finiteness of the Java integer types there are further pitfalls arising from the semantics of division and remainder operations.

In Java integer division / always rounds towards 0, i.e., the result of n/d with $d \neq 0$ is the integer value q defined as follows:

- $|q|$ is the (unique) value satisfying $|n| - |d| < |d \cdot q| \leq |n|$
- q is positive if both n and d have same signs and negative otherwise.

The remainder operator % yields the remainder of an implied integer division and is defined such that $(n/d)*d+(n\%d) = n$ holds (even in case of overflow).

In mathematics there are several definitions of integer division whereas the most commonly used is the Euclidean definition: For n, d with $d \neq 0$ there are uniquely defined numbers q, r with $n = q * d + r$ and $0 \leq r < |d|$ where q is the quotient n/d and r the remainder $n\%d$.

It can be shown that both definitions coincide for a non-negative dividend n. For negative dividends though the two definitions yield different results for integer division and remainder operations as Table 12.2 shows.

Table 12.2. Examples of integer division and modulo operations corresponding to the Java definition (columns 2–3) and the Euclidean definition (columns 4–5)

(n, d)	$n \; jdiv \; d$	$n \; jmod \; d$	n/d	$n\%d$
$(5, 3)$	1	2	1	2
$(5, -3)$	-1	2	-1	2
$(-5, 3)$	-1	-2	-2	1
$(-5, -3)$	1	-2	2	1

In Java Card DL there are two function symbols for division: $jdiv$ having the Java semantics (without overflow though) and / having the Euclidean semantics. The same applies to remainder functions $jmod$ and %.

Example 12.19. Let the program variables d, n, and q be of type int. Then the formula

$$\texttt{d} \, !\doteq 0 \; \& \; !(\texttt{d} \doteq -1 \; \& \; \texttt{d} \doteq \text{MIN}_{\texttt{int}}) \; \text{->} \; \langle \texttt{q=n/d;} \rangle \texttt{q} \doteq jdiv(\texttt{n}, \texttt{d})$$

is valid independently of the chosen integer semantics. The only case for JAVA division to cause overflow is $\texttt{MIN}_{\texttt{int}}/-1$ which is excluded by the left-hand side of the implication.

The formula

$$\texttt{d} \mathbin{!\dot=} 0 \mathbin{\&} \mathbin{!}(\texttt{d} \dot= -1 \mathbin{\&} \texttt{d} \dot= \texttt{MIN}_{\texttt{int}}) \mathbin{-\!>} \langle\texttt{q=n/d;}\rangle\texttt{q} \dot= \texttt{n/d}$$

is not valid since for negative values of $\texttt{n}$ or $\texttt{d}$ the results of JAVA division and the Euclidean division differ. Adding additional assumptions $\texttt{n} > 0 \mathbin{\&} \texttt{d} > 0$ ensuring that both $\texttt{n}$ and $\texttt{d}$ are positive makes the formula valid.

12.7 Conclusion

Developing software systems by stepwise refinement is often not easy and sometimes even impossible. In particular, the last step from an already refined specification to code often violates the principles of refinement. One reason is that idealistic data types that are available in specification languages, such as the natural numbers $\mathbb{Z}$ and the real numbers $\mathbb{R}$, are not available in programming languages. Instead, programming language data types have to be used that are not a correct refinement of the abstract specification language types. However, the programming language data types are not completely different from the abstract types—on parts of the domain they even behave as if they were a correct refinement. In this chapter we used the mathematical integers $\mathbb{Z}$ on the one hand and the primitive JAVA type $\texttt{int}$ on the other hand to illustrate the problems and to describe an approach to overcome the problem.

The idea of "correctness by construction" using refinement has to be adapted when, e.g., replacing $\mathbb{Z}$ with $\texttt{int}$, and additional proofs become necessary to ensure correctness of the system. We used the retrenchment framework [Banach and Poppleton, 1998] to formally describe the non-refinement steps. The advantage of casting non-refinement steps into the retrenchment framework is that it becomes explicit where exactly the refinement conditions are violated and, thus, where correctness cannot be shown once and for all (as it is the case with correct refinement). Instead, we prove the correctness of a program containing retrenchment by individually verifying critical situations. After these proofs have been done, no run-time checks are required. The additional proof obligations are systematically generated from the retrenchment clauses. Case studies showed that most of the additional proof obligations can be discharged automatically by the KeY system or by external decision procedures like CVC [Stump et al., 2002] and the Simplify tool, which is part of ESC/Java [Detlefs et al., 1998].

As far as we know, the KeY system is the first tool that implements retrenchment into a deductive calculus.

12.8 Related Work

Retrenchment was first mentioned in [Banach and Poppleton, 1998] as an answer to the problem that refinement is too restrictive for many practical applications. In [Banach and Poppleton, 1999] an even more general variant of retrenchment is presented.

Research on arithmetic in verification focused so far mainly formalising and verifying properties of floating-point arithmetic [Harrison, 1999, 2000] (following the IEEE 754 standard). However, there are good reasons not to neglect integer arithmetic and in particular integer arithmetic on finite programming language data types. For example, integer overflow was involved in the notorious Ariane 501 rocket self-destruction, which resulted from converting a 64-bit floating-point number into a 16-bit signed integer. To avoid such accidents in the future the ESA inquiry report [European Space Agency, 1996] explicitly recommended to "verify the range of values taken by any internal or communication variables in the software."

Approaches to the verification of Java programs that take the finiteness of Java's integer types into consideration—but not their relationship to the infinite integer types in specification languages—have been presented in [Jacobs, 2003, Stenzel, 2005].

The verification techniques described in [Poetzsch-Heffter and Müller, 1999, von Oheimb, 2001a, Huisman, 2001] treat Java's integer types as if they were infinite, i.e., the overflow problem is ignored.

Closely related to our approach is Chalin's work [Chalin, 2003, 2004]. He argues that the semantics of JML's arithmetic types (which are finite as in Java) diverges from the user's intuition. In fact, a high number of published JML specifications are shown to be inadequate due to that problem. As a solution, in [Chalin, 2003] Chalin proposes JMLa—an extension of JML with an infinite arithmetic type `bigInt`. In [Chalin, 2004] JMLa is extended to JMLb which offers three different semantics for JML integer expressions, called *math modes*:

- *Java math* which corresponds to the semantics of Java and, thus, to retrenchment R_{Java}.
- *Bigint math* which corresponds to the semantics on which the rule set intRules : arithmeticSemanticsIgnoringOF is based. Technically this is achieved by an implicit type cast of arithmetic expressions that might cause overflow to the infinite type `bigInt`.
- *Safe math* which is like Java math except that overflow is signalled by means of an exception (like C# checked mode). This mode is not that strict as retrenchment R_{KeY}. It does not prohibit overflow but it still can contribute to avoid incidental correctness by signalling overflow with an exception.

Breunesse [2006] defines JMLc, another extension of JML and a slight variation of JMLb. Since the differences between JMLb and JMLc are quite subtle, we do not go into details here.

13

Proof Reuse

by
Vladimir Klebanov

13.1 Introduction

The Need for Proof Reuse in Software Verification

Experience shows that the prevalent use case of program verification systems is not a single prover run. It is far more likely that a proof attempt fails, and that the program (and/or the specification) has to be revised. Then, after a small change, it is better to adapt and reuse the existing partial proof than to verify the program again from first principles. A particular advantage is that proof reuse can reduce the number of required user interactions.

Here we present such a technique for proof reuse. In fact, towards the end of this chapter ($\Rightarrow$ Sect. 13.9), we will show how our method can improve the user experience for a whole range of verification scenarios. Until then, we limit ourselves to the setting described above, with the further assumption that only the implementation changes and the specification remains unchanged.

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 the full range of proof reuse applications in deductive verification of JAVA software.

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" ($\Rightarrow$ Sect. 13.8).

(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 the Section 3.4. 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) finalise 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 3.6.3, 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.

13.2 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:

```java
int x;
int res;
res=x/x;
```

Its intended behaviour 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 13.1 (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:

───── JAVA ───

```java
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 13.1 (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.

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 premises, 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 13.7.

13.3 The Main Reuse Algorithm
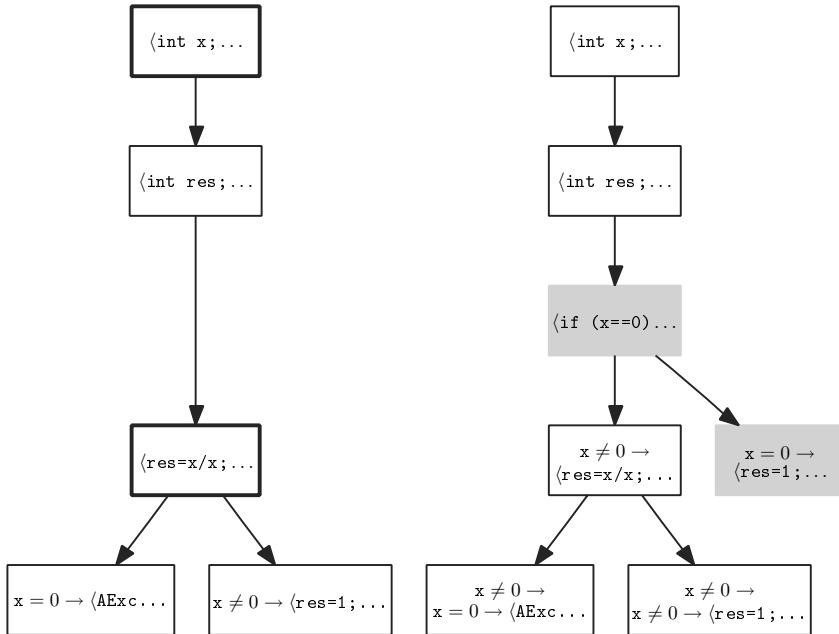
Basic Ideas

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:

─────────────────────

[2] In fact, JAVA requires initialising 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.

Fig. 13.1. Schematic proofs (a) before and (b) after program correction. The left-most branch of the old proof cannot be closed, since the program contains a bug. `AExc` is shorthand for `throw new ArithmeticException();`.

(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 generalise and extract the essence of the above choices in the old proof.

For finding the rules that are candidates for choice (a), such a generalisation is readily available. The rule schemata are natural generalisations 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 ($\Rightarrow$ Sect. 13.4). 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 generalisation for choice (b) and is used to prioritise 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 13.2. 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 initialised with the list of *initial candidates C_0*, which is computed by the function *initialCandidateList* from the differences in programs.

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 13.3.[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.

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.

---

[3] We show a nested loop implementation for its clarity. The actual implementation uses an optimised incremental computation algorithm.

[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 13.5.

— Pseudocode —

input *oldProof*, *oldProgram*, *newProgram*, *specification*;

newProof := initialProofGoal(*newProgram*, *specification*);
C_0 := initialCandidateList(*oldProof*, Δ(*oldProgram*, *newProgram*));
$C := C_0$;
while *newProof* has open goals **do**
   ⟨*candidate*, *newFocus*⟩ := chooseReuse(C, *oldProof*, *newProof*);
   if ⟨*candidate*, *newFocus*⟩ ≠ ⊥ **then**
      newProof := result of applying rule(*candidate*) at *newFocus* in *newProof*;
      if *candidate* ∉ 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 —

Fig. 13.2. Main reuse and proof construction algorithm

— Pseudocode —

    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* ∪ ⟨c, p⟩;
             fi;
          od;
       od;
    od;
    if *possibleReuses* = {} **then return** ⊥ **fi**;
    select ⟨c, p⟩ from *possibleReuses* with score(⟨c, p⟩) maximal;
    if score(⟨c, p⟩) > ε **then**
       return ⟨c, p⟩;
    else
       return ⊥;
    fi;

— Pseudocode —

Fig. 13.3. Function for the best possible reuse pair

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 13.4 and 13.5, respectively. Note that our algorithm is "modular" in the sense that the answers can be given independently.

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 symbolised 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.

13.4 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 linearise 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 behavioural 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 13.1. 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:

$$A = \begin{cases} \texttt{LocalVarDecl, LocalVarDecl,} \\ \texttt{Assignment(int)} \end{cases}$$

$$B = \begin{cases} \texttt{LocalVarDecl, LocalVarDecl,} \\ \underline{\texttt{If}}, \texttt{Assignment(int)}, \underline{\texttt{Assignment(int)}} \end{cases}$$

The underlined parts correspond to the insertions in the minimal edit script. It consists of the two commands $2\,I\,\texttt{If}$ and $4\,I\,\texttt{Assignment(int)}$.

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 13.2 (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]

---

[5] Please note that all numbers provided here are for orientation purposes only. The numbers in your version of the KeY system may vary.

$$P(e) \quad = \quad \begin{cases} \sum_{k=1}^{t} \dfrac{0.75}{x+k} & \text{if } e = x \, I \, b_1 \, b_2 \cdots b_t \\[2mm] \dfrac{1}{x+1} & \text{if } e = x \, D \end{cases}$$

We remind that x is the numeric position of the insertion/deletion as counted from the beginning of the linearised 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 $\delta(A, B)$ differs in general from $\delta(B, A)$. Statement insertions are penalised 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.

Example 13.3 (Example 13.1 continued). We now consider the minimal edit script for the programs α and β presented above. It consists of the two commands $2\,I\,$`If` and $4\,I\,$`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.

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 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 linearised, 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.

13.5 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 13.2, 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 x;              -- old
int res;          int res;            +++ new
res=x/x;          if(x==0) {          @@ -1,3 +1,7 @@
                      res=1;           int x;
                  } else {             int res;
                      res=x/x;        +if(x==0) {
                  }                   +  res=1;
                                      +}else {
                                       res=x/x;
                                      +}
        (a)               (b)                 (c)
```

Fig. 13.4. Change detection with GNU diff: (a) old program, (b) new program, and (c) output of "`diff -uw`"

Example 13.4. The output of GNU diff for our running example is shown in Figure 13.4. 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 13.1 (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, 2003a], 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 semantically aware diff of two JAVA programs [Apiwattanapong et al., 2004]. Unfortunately, this work is limited to JAVA bytecode, which is cumbersome to version control.

13.6 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 analyse 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 colour the nodes in the proof tree it constructs with different colours. 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.

13.7 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 13.4, which

shows that we obtain two candidates in our case. These are the nodes with a bold border in Figure 13.1 (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):

$$\Longrightarrow \langle\texttt{int x; int res;} \hspace{3.5cm} \text{(G0)}$$
$$\texttt{if (x==0) res=1; else res=x/x;}\rangle(\texttt{res} = 1)$$

The similarity score for the single possible reuse pair (see Example 13.1 for the computation) is -0.4, and reuse is performed. We get the new goal

$$\Longrightarrow \langle\texttt{int res;} \hspace{5.2cm} \text{(G1)}$$
$$\texttt{if (x==0) res=1; else res=x/x;}\rangle(\texttt{res} = 1)$$

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

$$\Longrightarrow \langle\texttt{if (x==0) res=1; else res=x/x;}\rangle(\texttt{res} = 1) \hspace{1cm} \text{(G2)}$$

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:

$$\Longrightarrow \texttt{x} = 0 \mathrel{\texttt{->}} \langle\texttt{res=1;}\rangle(\texttt{res} = 1) \hspace{2cm} \text{(G2.1)}$$
$$\Longrightarrow \texttt{!(x} = 0) \mathrel{\texttt{->}} \langle\texttt{res=x/x;}\rangle(\texttt{res} = 1) \hspace{1.5cm} \text{(G2.2)}$$

There are still two identical candidate proof steps, both with the rule handling "res=x/x;" in the old proof. It 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:

$$\Longrightarrow \ !(\mathtt{x} = 0) \ \text{->}$$
$$!(\mathtt{x} = 0) \ \text{->} \ (\mathtt{res} = div(x, x) \ \text{->} \ \langle\rangle \, (\mathtt{res} = 1)) \tag{G2.2.1}$$

$$\Longrightarrow \ !(\mathtt{x} = 0) \ \text{->}$$
$$\mathtt{x} = 0 \ \text{->} \ \langle\texttt{throw new ArithmeticException();}\rangle \, (\mathtt{res} = 1) \tag{G2.2.2}$$

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:

$$!(\mathtt{x} = 0) \ \text{->} \ (\{\mathtt{res} = div(x, x)\} \, \langle\rangle \, (\mathtt{res} = 1)) \tag{C-N}$$
$$\mathtt{x} = 0 \ \text{->} \ \langle\texttt{throw new ArithmeticException();}\rangle \, (\mathtt{res} = 1) \tag{C-Z}$$

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.

13.8 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.

---

[6] The numbers can vary with the version of the KeY system.

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 analyse 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., 2000], 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 [Paulson, 1994]. 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.

13.9 Other Uses for Reuse: Reuse as a Proof Search Framework

In this section we discuss how proof reuse fulfills a need that goes beyond the basic scenario that we have presented so far.

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 method call rule (⇒ Sect. 3.6.5) and the typeAbstract rule (⇒ Sect. 2.5.6)—the particular rule instance depends on the set of classes constituting the program (⇒ Def. 3.10, 3.14). Using an old instance in the new context may be unsound.

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. 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. This issue is also addressed in Sect. 8.5.4.

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.

---

[7] Another take on this problem is given in Section 8.5.4. Furthermore, see [Roth, 2006] for a detailed discussion.

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.

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.

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.

The Case of Using Customisable Calculus Modules

Another opportunity for proof reuse arises when using customisable 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 customisable modules include: null dereferencing checks (on or off), treatment of static initialisation (on or off), integer semantics (three different ones) and others. The user can select a particular rule set for each area via the taclet options mechanism ($\Rightarrow$ Sect. 4.4.2).

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 an an example, which Chapter 12 discusses in detail. The approach of choice here is to verify a program using the mathematical integer semantics, and afterwards repeat the proof with the 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 ($\Rightarrow$ Sect. 12.5.1).

$$\text{assignmentAdditionToUpdateCheckingOF}$$
$$\frac{\boxed{Range_T(se_1),\ Range_T(se_2) \Longrightarrow Range_T(se_1 + se_2)}}{\Longrightarrow \langle \pi\ var\texttt{=}se_1\texttt{+}se_2;\ \omega \rangle \phi}$$
$$\Longrightarrow \{var := se_1 + se_2\}\langle \pi\ \omega \rangle \phi$$

This means that the R_{KeY}-proof is just the original proof with 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.

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:[8]

1. logic syntax
2. parser/disambiguation
3. formalisation of the JAVA language semantics
4. logical structure of the rules
5. rule execution engine

We now briefly discuss the important cases (3) and (4). We are currently extending the reuse facility to automate translation of proofs between versions of the proof system affected by these changes.

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.[9] The KeY project on regular bases performs the only measure suitable to mitigate this: cross-checking our rules with other formalisations 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 13.5. As one can see, the changes are minor and of local nature, which should allow a similarity-guided proof reuse.

---

[8] See [Beckert et al, 2005a] for a complete survey.

[9] For an in-depth discussion of the calculus soundness issue please see Section 3.4.2.

$$a = null \Longrightarrow \langle \pi \ \texttt{NPE;} \ \omega \rangle \phi$$
$$a \neq null \wedge (i < 0 \vee i \geq a.length) \Longrightarrow \langle \pi \ \texttt{AOBE;} \ \omega \rangle \phi$$
$$\underline{a \neq null \wedge i \geq 0 \wedge i < a.length \Longrightarrow \{a[i] := val\}\langle \pi \ \omega \rangle \phi}$$
$$\Longrightarrow \langle \pi \ \texttt{a[i]=val} \ \omega \rangle \phi$$

$$a = null \Longrightarrow \langle \pi \ \texttt{NPE;} \ \omega \rangle \phi$$
$$a \neq null \wedge (i < 0 \vee i \geq a.length) \Longrightarrow \langle \pi \ \texttt{AOBE;} \ \omega \rangle \phi$$
$$\boxed{a \neq null \wedge i \geq 0 \wedge i < a.length \wedge \neg storable(val, a) \Longrightarrow \langle \pi \ \texttt{ASE;} \ \omega \rangle \phi}$$
$$\underline{a \neq null \wedge i \geq 0 \wedge i < a.length \boxed{\wedge storable(val, a)} \Longrightarrow \{a[i] := val\}\langle \pi \ \omega \rangle \phi}$$
$$\Longrightarrow \langle \pi \ \texttt{a[i]=val} \ \omega \rangle \phi$$

Abbreviations: `NPE=throw new NullPointerException()`
`AOBE=throw new ArrayIndexOutOfBoundsException()`
`ASE=throw new ArrayStoreException()`

Fig. 13.5. A rule for array assignment: initial and revised version (differences are boxed)

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 the form splitting the proof (e.g. ifElseSplit) to a form employing a conditional formula (rule ifElse, both rules are given in Section 3.6.3), 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.

13.10 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 artefacts, 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 (see the sidebar on page 3.4.4), 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.

Part IV

Case Studies

14

The Demoney Case Study

by
Wojciech Mostowski

14.1 Introduction

So far in this book no specific JAVA CARD examples have been discussed, apart
from very simple transactions related examples in Chapter 9. In this chap-
ter we are going to discuss how the KeY system is used to verify real world
JAVA CARD programs. The basis and running example for this chapter is the
demonstrative JAVA CARD electronic purse application *Demoney* provided by
Trusted Logic S.A. The purpose of this chapter is not to present full specifi-
cations for *Demoney* and a thorough verification procedure, or give a tutorial
on using the KeY system verification facilities (⇒ Chap. 10). Rather, we will
discuss general problems associated with JAVA CARD verification attempts:
provide advice on what to specify about JAVA CARD applets, how to specify it
using different specification languages, and give the reader hints necessary to
perform efficient verification of his or her own JAVA CARD applets. Verifica-
tions that we are going to discuss were performed on an absolutely unmodified
source code of *Demoney*, thus, the context of this chapter is slightly different
from the rest of the book—we do not consider the use of formal methods
during the development process, instead we show how one can verify preex-
isting, legacy JAVA CARD code, in other words, perform *post-hoc* verification.
We also assume that, to a certain extent, the reader is familiar with JAVA
CARD technology, described by Chen [2000] (⇒ Chap. 9).

In the next section we present the *Demoney* application in more detail,
in Section 14.3 we discuss different specification techniques and languages
available in the KeY system, their advantages and disadvantages in the con-
text of *Demoney* verification. Section 14.4 discusses modular verification and
the use of method contracts in proofs, which is necessary to make the veri-
fication process scalable. Then in Section 14.5 different properties related to
JAVA CARD applets are presented, example specifications are given based on
Demoney and verification is discussed. Finally, Section 14.6 summarises this
chapter.

14.2 Demoney

The case study that we are going to use throughout this chapter is a Java Card electronic purse application *Demoney*. While *Demoney* does not have all of the features of a purse application actually used in production (for example, no really sophisticated security measures are implemented), it was developed some years ago by *Trusted Logic S.A.* as a realistic demonstration application (yet still accessible to non-specialists) that includes some major complexities of a commercial program. In particular *Demoney* is partly optimised for memory consumption, which, as noted by Hähnle and Mostowski [2005], is one of the major obstacles in verification. The *Demoney* source code is at present not publicly available, so there are certain limits to the level of the technical detail in the presented examples. The implementation of *Demoney* follows the extensive, informal specification given by Marlet and Mesnil [2002], here we give a brief overview of the complexity of the *Demoney* source code.

The *Demoney* implementation consists of 9 Java classes and the total of 150 kB of code, large parts of which are comments. The main *Demoney* class with the actual Java Card applet consists of over 3 000 lines of code (100 kB) and contains 42 methods. On top of that *Demoney* uses the Java Card API, which itself contains over 50 classes. Of course not all of the API classes and methods are used in *Demoney*, but it is important to remember that the size of the API in principle influences the verification complexity—every call to an API method in the program has to be treated appropriately in the proof, either by in-lining the referenced method's implementation or by using a suitable method contract. We discuss this issue in detail in Section 14.4.

14.3 OCL, JML, and Dynamic Logic

The KeY system offers three ways of specifying the behaviour of the program: in OCL, JML, and directly in Java Card DL. The KeY system's OCL interface is described in detail in Section 5.2 and JML interface in Section 5.3. Using Java Card DL to specify and verify the behaviour of the system means writing the correctness formulae directly in Java Card DL in the form of a `.key` file and loading it directly to the KeY prover ($\Rightarrow$ Chap. 10).

Java Card applets are very specific kinds of Java programs, usually exhibiting the following features:

- Java Card applets hardly ever make use of class inheritance or interfaces,
- in general, Java Card implementations do not support the `int` primitive data type, instead `byte` and `short` types are widely used giving raise to overflow issues,
- Java Card applets most often store all their data in applet's primitive type attributes or primitive type arrays as contiguous byte segments, and

not in dedicated class structures, and thus there are hardly ever any classes other than the main applet class,

- JAVA CARD applets use exceptions as the only means to communicate errors to the card terminal,
- due to limited resources of a smart card and lack of garbage collection (in general) JAVA CARD applets (i) hardly ever do any memory allocation and (ii) have to deal with many different kinds of runtime exceptions,
- each JAVA CARD memory location can be persistent or transient which affects the behaviour of the JAVA CARD transaction mechanism ($\Rightarrow$ Sect. 9.3).

Because of those features, each of the three ways to specify JAVA CARD applet behaviour have strong and weak aspects, which we outline next.

As for OCL, it seems that in the context of JAVA CARD applet verification OCL does not offer substantially more expressiveness than JML. Moreover, some things are comparatively more difficult (or impossible if only pure OCL is considered, without custom extensions for the KeY system) to express in OCL than in JML or JAVA CARD DL. For example, it is not easy to write expressions involving different integer types or primitive arrays that a JAVA CARD applet deals with. Also, complex exceptional behaviour is difficult to express in OCL, as described by Larsson and Mostowski [2004]. On top of that OCL is not aware (and neither is JML) of the JAVA CARD transaction mechanism and different persistency types of objects, for example, it is not possible to specify that a program or a method makes only conditional updates to object attributes ($\Rightarrow$ Sect. 9.5.2).

So, considering also that JAVA CARD programs usually have a flat class structure and hardly ever make use of advanced software engineering concepts, like design patterns, we are not going to discuss further the use of OCL ($\Rightarrow$ Sect. 5.4) in the context of this chapter.

JML is a specification language that has been designed specifically for JAVA, thus, it handles features specific to JAVA much better. Moreover, it seems that JML has been accepted as a standard specification language in the JAVA CARD verification community, which means there are plenty of references to problems related to specifying JAVA CARD programs in JML ($\Rightarrow$ Sect. 14.6.1), in particular, a comprehensive library of JAVA CARD API specifications tested in practice are available [Poll et al., 2000]. The one thing that JML does not fully support are properties related to the JAVA CARD transaction mechanism. While some of the properties related to object persistency can be expressed through references to the JAVA CARD API, some properties are not expressible at all, like strong invariants or conditional updates ($\Rightarrow$ Sect. 9.4).

One disadvantage of JML is its strict, visible state semantics semantics ($\Rightarrow$ Chap. 8). In the context of this chapter it means that verification of properties based on full JML semantics can be very difficult and impractical. For example, JML requires that all invariants of all instances of all classes in

the project have to be preserved by all methods. Thus, a proof obligation generated from a JML specification for a project of realistic size can be quite big and complex. On the other hand, many useful properties about Java Card applets can be proved using a much lighter correctness semantics. To achieve this in practice, it is necessary to construct proof obligations manually, by writing appropriate Java Card DL formulae in a file that is later fed into the prover.[1] Constructing such proof obligations "manually" is almost as easy as writing JML specification, moreover, it also brings certain advantages into the specification and verification process. Most importantly, Java Card DL provides full support for specifying strong invariants and behaviour related to the Java Card transaction mechanism. Secondly, Java Card DL expressions can be optimised "up-front" for a given proof obligation and certain properties can be expressed in a more direct way by accessing "low level" Java Card DL features (for example, directly referring to implicit fields). This has serious impact on verification efficiency and automation. In addition, to further simplify the proof, the user can include his or her own problem specific logic rules. We will elaborate more on these issues later on in the chapter.[2]

The only disadvantages of using Java Card DL as a specification language are related to user friendliness. For a single method to be verified, the user has to write a .key file that includes the full specification for the method, including necessary parts of the class invariant and invariants for all the objects accessed by the method. Such a file also needs to contain method specifications (contracts), that are going to be used in the proof, directly expressed in Java Card DL. To make the process of writing .key files easier, the KeY system provides a file inclusion mechanism that prevents repetition of Java Card DL formulae over different files. Finally, JML is more widespread than Java Card DL: JML is probably easier to read for an average Java programmer and using Java Card DL as a specification language limits the usability of the specifications to the KeY system.

In the remainder of this chapter we will be presenting and discussing specifications both in Java Card DL and in JML, where applicable. Due to impracticality of the (full) JML approach in the context of this chapter, comments about verification and verification benchmarks are based on Java Card DL proof obligations only, which use the simplified observational correctness semantics.

---

[1] A new proof obligation generation mechanism ($\Rightarrow$ Chap. 8) allows to also produce lightweight proof obligations for JML following observed-state correctness semantics ($\Rightarrow$ Sect. 8.2). The manually constructed Java Card DL proof obligations that we discuss in this chapter adhere to such observational semantics.

[2] It should be stressed that constructing proof obligations manually can easily introduce trivially verifiable proof obligations, or at least proof obligations that are weaker than intended. On the other hand, it is also easy to write a JML specification that is trivially satisfied due to specification inconsistency, however, manually constructed proof obligations are more prone to problems of this kind.

14.4 Modular Verification

No matter which language is used to specify the behaviour of the JAVA CARD program or method, one needs to have support to handle calls to other methods inside the program, in particular, calls to the JAVA CARD API. What it means is that the prover has to have access to either a reference implementation of the API classes and methods, or a set of specifications for those classes and methods that can be used in the proof (method contracts).

Using the reference implementation to deal with API calls poses certain difficulties and inefficiencies in the verification process. First of all, the reference implementation has to be available. Considering that many of the JAVA CARD API methods are native, this is not always possible. Secondly, even if the reference implementation is available (either provided by a JAVA CARD manufacturer, or self-developed specifically for verification), using such an implementation in the proof is not always efficient. Take for example JAVA CARD's API static method `arrayCompare`, which simply compares contents of two arrays. The implementation of this method involves a loop that iterates over the elements of the two arrays. It is known that dealing with loops in program verification is difficult ($\Rightarrow$ Chap. 3, 11), and in-lining the implementation of `arrayCompare` into the program to be verified certainly poses an extra verification challenge. On the other hand, in most cases it is enough to specify `arrayCompare` with a simple postcondition saying that the method returns a byte value -1, 0, or 1 given that the method parameters satisfy a precondition that guarantees exception free execution.

Another example of introducing inefficiency into the proof is repeated in-lining the (possibly very simple) implementation of the same method. Such a situation can be easily triggered by a repeated call in the program to be verified, but also by a heavily branching proof, that is, when the same method call has to be analysed many times in different proof branches.

It is clear that to avoid such inefficiencies, deal with methods without implementation, and make the verification process highly scalable, the second approach of using class specifications and method contracts in the proof has to be taken.

This gives raise to modular verification ($\Rightarrow$ Sect. 8.5). It is a feature of JML to support modular verification ($\Rightarrow$ Sect. 5.3). In the context of this chapter we will also use method contracts written directly in JAVA CARD DL.

Again, there are certain advantages of using JAVA CARD DL instead of JML. For example, one can use Dynamic Logic function symbols in method specifications, which can be a very efficient replacement for JML's model fields and methods, and it is possible to describe transaction specific behaviour of a method in JAVA CARD DL, including strong invariants.

14.4.1 KeY Built-in Methods

Whenever a call to a JAVA CARD API method is encountered in the program to be verified, the KeY system has to have at least the signature of the method available, even if the method is going to be handled by applying a method contract.

Moreover, because of the way JAVA CARD transactions are handled in the KeY system ($\Rightarrow$ Sect. 9.5, 9.9) some of the JAVA CARD API methods cannot be fully specified, notably the methods `beginTransaction`, `commitTransaction`, `abortTransaction`, `isTransient`, and the `makeTransientArray` method family from class `JCSystem`. Their implementation has to be given and used in the proof explicitly. Also, to handle transaction properly, the JAVA CARD API has to contain signatures of the methods representing the "internal" transaction triggering statements that the JAVA CARD DL calculus uses—`jvmBeginTransaction`, `jvmCommitTransaction`, etc. Such a customised skeleton of the JAVA CARD API, which also contains reference implementations for the most commonly used methods[3] can be found on the KeY and book webpages. A full JAVA CARD API written in JML that can be adapted to the needs of the KeY system can be found at `www.cs.ru.nl/E.Poll/publications/jc211_specs.html`. An attempt to produce a complete OCL specification for the JAVA CARD API was also given by Larsson and Mostowski [2004], however, as we said already, we are not going to refer to OCL in this chapter.

Apart from the JAVA CARD API methods that are related to transactions, there are a few other methods that are treated in a special way during the proof. The methods in question are `arrayCompare`, `arrayCopy`, `arrayCopyNonAtomic`, `arrayFillNonAtomic`, `makeShort`, `setShort`, and `getShort` from the `Util` class. Although in principle it is possible to give a reasonable contract specification for each of these methods, it is really difficult to give a highly accurate one that would reflect all of the specificities and complexities that some of these methods entail, for example, implicit transactions or data representation in card's memory (big or small endian) [Sun, 2003b,c]. A solution to this is to provide a relatively simple implementation for each of these methods in terms of an "internal" (or low-level) API method. Such an internal method is then in turn treated by a specialised taclet that accurately reflects the method's behaviour. This is usually done by introducing a dedicated update into the sequent[4] possibly involving special function symbols. Take, for example, the implementation of `getShort`:

---

[3] The reference implementation that we talk about here and use throughout this chapter is for JAVA CARD API 2.2.1. It is partly based on Sun's reference implementation of an older API version (2.1.1) and partly self-developed. The implementation of the API methods is in a way naïve—it tries to reflect accurately the official API specification [Sun, 2003b], but it does not take into account low-level security issues, possible hardware failures, or JAVA CARD object ownership.

[4] Introducing an update directly into the formula can also in many cases simplify the proof as additional, complex update substitutions can be avoided.

```
                ── Java ─────────────────────────────────────

public static short getShort(byte[] bArray, short bOff) {
  if(bArray == null) throw new NullPointerException();
  if(bOff < 0 || (short)(bOff + 1) >= bArray.length)
    throw new ArrayIndexOutOfBoundsException();
  // bArray[bOff] and bArray[(short)(bOff+1)] exception-free,
  // call "special" method
  return de.uka.ilkd.key.javacard.KeYJCSystem.
    jvmMakeShort(bArray[bOff], bArray[(short)(bOff+1)]);
}
                                      ──────────── Java ──
```

The method produces a short value represented by two byte values stored in
`bArray`. A corresponding taclet that executes method `jvmMakeShort` simply
introduces the following update into the sequent:

$$\{\texttt{res} := jvmMakeShort(\texttt{bArray[bOff]}, \texttt{bArray[bOff+1]})\} \;.$$

Here `res` is the result variable for method `jvmMakeShort` and *jvmMakeShort*
is a function symbol that can be later treated with dedicated rules that
represent the semantics of composing and decomposing `shorts` into `bytes`.

  Of course, it is still possible (if necessary) to give a contract specifica-
tion for method `getShort`, verify its implementation by using the customised
taclet for `jvmMakeShort`, and then use the contract for `getShort` in the proof
of a program that makes calls to `getShort`. But in most cases, specifying
`getShort` (or other special methods) is not necessary, in-lining its implemen-
tation directly gives expected results with reasonable efficiency.

14.5 Properties

In the remainder of this chapter we discuss different properties that one
can specify about Java Card applets and prove with the KeY system. We
illustrate the properties with examples based on *Demoney* and discuss the
verification issues in detail. In general, Java Card applet properties can be
divided into two classes: functional properties and security properties. We
discuss them in the following sections.

14.5.1 Functional Properties

Functional properties are concerned with the actual functionality of the pro-
gram, that is, what parts of the program state are changed by a method,
how they are changed, and what is the return value of a method. Take, for
example, the method `keyNum2keySet` in class `Demoney` with the following
signature:

—— Java ——

```
private DESKey[] keyNum2keySet(byte keyNum);
```

—————————————————————————————— Java ——

Informally, the method returns a set of `DESKeys` based on the parameter `keyNum`. A `keyNum` can be one of the `byte` constants `DEBIT_KEY_NUM`, `CREDIT_KEY_NUM`, or `ADMIN_KEY_NUM`. When `keyNum` is not equal to either of the three values the method throws a `CardRuntimeException`. The method does not change any instance attributes of the `Demoney` applet class. Assuming that in *Demoney* the method is always called with a correct argument and never gives raise to an exception, the behaviour can be completely specified in JML in the following way:

—— Java + JML ——

```
/*@ public normal_behavior
   requires keyNum == DemoneyIO.DEBIT_KEY_NUM ||
            keyNum == DemoneyIO.CREDIT_KEY_NUM ||
            keyNum == DemoneyIO.ADMIN_KEY_NUM;
   ensures  keyNum == DemoneyIO.DEBIT_KEY_NUM ==>
               \result == keySets[0];
   ensures  keyNum == DemoneyIO.CREDIT_KEY_NUM ==>
               \result == keySets[1];
   ensures  keyNum == DemoneyIO.ADMIN_KEY_NUM ==>
               \result == keySets[2];
   assignable \nothing;
@*/
private DESKey[] keyNum2keySet(byte keyNum) {...}
```

—————————————————————————— Java + JML ——

The method `keyNum2keySet` accesses the attribute `keySets` of type `Object[]` to extract the actual key set to be returned. Thus, we also need an invariant that describes valid contents of the `keySets` array for the method `keyNum2keySet` to terminate properly without throwing exceptions:

—— Java + JML ——

```
/*@ public invariant keySets != null && keySets.length == 3 &&
      (\forall int i; i >= 0 && i < keySets.length;
            keySets[i] instanceof javacard.security.DESKey[]);
@*/
/*@spec_public@*/ private Object[] keySets;
```

—————————————————————————— Java + JML ——

This is enough information to prove the full functional property of `keyNum2-keySet`, including the fact that the method does not throw anyexception,

which also qualifies as a security property (to be discussed shortly). But before we discuss the verification of `keyNum2keySet`, let us first have a look at how the same specification is written in JAVA CARD DL. First of all, we need to specify where the source code for the verified program is located and declare program variables that are going to be used in the proof obligation:

—— KeY ——

```
\javaSource "demoney/";

\programVariables {
  fr.trustedlogic.demo.demoney.Demoney self;
  byte keyNum;
  javacard.security.DESKey[] result;
}
```

—— KeY ——

Then we can specify the proof obligation directly as a JAVA CARD DL formula. To ensure the termination of `keyNum2keySet` we need to assume its precondition and the part of the class invariant specifying the contents of `keySets`:

—— KeY ——

```
\problem {
// Class invariant:
    self.keySets != null
  & self.keySets.<created> = TRUE
  & self.keySets.length = 3
  & \forall int i; (i >= 0 & i < self.keySets.length ->
      javacard.security.DESKey[]::instance(
        self.keySets[i]) = TRUE)
// keyNum2keySet precondition:
  & (keyNum = DemoneyIO.DEBIT_KEY_NUM |
     keyNum = DemoneyIO.CREDIT_KEY_NUM |
     keyNum = DemoneyIO.ADMIN_KEY_NUM)
-> \<{
     result = self.keyNum2keySet(keyNum)
       @fr.trustedlogic.demo.demoney.Demoney;
     }\>
// keyNum2keySet postcondition:
     ((keyNum = DemoneyIO.DEBIT_KEY_NUM ->
         result = self.keySets[0]) &
      (keyNum = DemoneyIO.CREDIT_KEY_NUM ->
         result = self.keySets[1]) &
      (keyNum = DemoneyIO.ADMIN_KEY_NUM ->
         result = self.keySets[2])
```

```
// Class invariant:
        self.keySets != null
      & self.keySets.<created> = TRUE
      & ...)
}
```
                                                            — KeY —

Even if this proof obligations looks very similar (in fact, syntactically it is practically identical) to the JML specification given above, when verification is considered some differences occur. In case of the JAVA CARD DL specification we have to prove exactly what the proof obligation states, that is, the dynamic logic formula included in the `\problem` section. For JML however, when a proof obligation is generated that follows the full JML semantics, the precondition and the postcondition of the `keyNum2keySet` include all invariants of all the classes from *Demoney* and JAVA CARD API.

The explicit JAVA CARD DL proof obligation is proved fully automatically by the KeY system in less than 30 seconds[5] by first applying the basic JAVA CARD DL strategy and then calling Simplify.

Taclet and Prover Options

For all verification problems like this one, the user has to remember to set the taclet options appropriately (⇒ Sect. 4.4.2). First of all, the `null` pointer checks should be turned on to thoroughly analyse the program for possible abrupt termination resulting from `NullPointerExceptions`.

Secondly, due to the characteristics of the JAVA CARD language, applets are usually subject to overflow issues. To control overflow in the verified program a taclet option for semantics of arithmetic operations has to be set accordingly (⇒ Sect. 12.5). Proving problems with overflow control switched on is substantially more difficult, thus for most of the problems that we discuss in this chapter, natural integer semantics has been used. In Section 14.5.5 we discuss a particular example of a program containing arithmetic expressions causing overflow.

Thirdly, if the JAVA CARD transaction mechanism is involved in the program to be verified the `transactions` option should be set to `transactionsOn` (⇒ Sect. 9.9). If the program does not contain a call to `abortTransaction`, the user can simplify the proof by setting the `transactionAbort` option to `abortOff`. Note, that there can be also implicit aborts in the program, for example, when the program leaves an open transaction. In this case the option needs to be set to `abortOn`, or otherwise the proof will not close. When strong invariants are used in the proofs (the throughout modality) the `throughout` option needs to be set to `toutOn`. Setting the

---

[5] All the benchmarks discussed in this chapter were run on Pentium IV 3.4 GHz Linux system with 1 GB of memory. Although slightly slower, all the proofs discussed in this chapter can be also run on a mid-sized laptop without any problem.

> taclet options in the wrong way does not introduce any unsoundness into the proof in any way, but with inadequate taclet options the proof may not (and most likely will not!) be closable, even for correct proof obligations.
>
> Finally, if the verified source code contains lots of JML annotations which are not intended to be used (because the proof obligations and method contracts are given directly in the `.key` file), the prover can be run with the `no_jmlspecs` option, which will speed up the start up time and save considerable amounts of memory.

The *Demoney* method we have just discussed is quite small and its specification is relatively simple. However, we did not only prove the strictly functional specification of the method but also its termination behaviour, that is, that the method terminates and does so in a non-abrupt fashion—this is what the KeY semantics of the diamond modality in the proof obligation requires. Such a non-abrupt termination property qualifies as a simple security property. More advanced security properties require a program not to throw exceptions of a given sort or allow a program to throw exception *only* of a given sort. In both cases there may be additional conditions. We discuss such properties in the following sections.

14.5.2 Security Properties

As opposed to functional properties, security properties are usually not concerned with state changes caused by a method or the return value of the method. Instead, they concentrate on program behaviour with respect to abrupt termination, preserving sensitive data in good shape, or data confidentiality. A comprehensive set of security properties specific to JAVA CARD applications has been developed in the context of the SecSafe project[6] and published in [Marlet and Métayer, 2001], which we will refer to as the *SecSafe document* in the rest of this chapter. The *SecSafe document* has been developed in cooperation with *Trusted Logic S.A.*, thus, the properties we present here should be considered as *industrial*, that is, properties that are potentially of great interest to the JAVA CARD industry. The properties from the *SecSafe document* that are fully specifiable in JML or JAVA CARD DL and verifiable with the KeY system are discussed in the following sections. Later the other properties, that are not yet supported by the KeY system, are discussed briefly. We start with an overview of the fully supported security properties.

Only ISOExceptions at Top Level
(Section 3.4 of the *SecSafe Document*)

Exceptions of type `ISOException` are used in JAVA CARD to signal error conditions to the smart card terminal. Such an exception results in a specific APDU

---

[6] http://www.doc.ic.ac.uk/~siveroni/secsafe/

(Application Protocol Data Unit) carrying an error code being sent back to the card terminal. To avoid leaking the information about error conditions inside the applet, a well written Java Card applet should only allow exceptions of type `ISOException` to reach the top level of an applet. In other words, `ISOException` is the only kind of exception that is allowed not to be caught inside an applet and propagate to the card terminal through an APDU.

No *X* Exceptions at Top Level
(Section 3.4 of the *SecSafe Document*)

Due to its complexity, the previous property is proposed to be decomposed into simpler sub-properties. Such properties say that certain exceptions should not reach the top level of an applet, including most common `NullPointerException`, `ArrayIndexOutOfBoundsException`, or `Negative-ArraySizeException`. However, when the capabilities of the KeY system are considered, this property is no different than the previous property—in Java Card DL non-abrupt termination required by the diamond modality disallows all kinds of exceptions.

Well Formed Transactions
(Section 3.4 of the *SecSafe Document*)

This property relates to the Java Card transaction mechanism and consists of three parts, which say, respectively: do not start a transaction before committing or aborting the previous one, do not commit or abort a transaction without having started any, and do not let the Java Card Runtime Environment close an open transaction. The Java Card specification allows only one level of transactions, that is, Java Card transactions cannot be nested. The first two parts of this property can be expressed in terms of disallowing Java Card's `TransactionException` at top level, and the third part can be considered together with the next property as they are closely related.

Atomic Updates
(Section 3.5 of the *SecSafe Document*)

In general, this property requires related persistent data in the applet to be updated atomically. This is exactly what the *card tear* properties and strong invariants are about, which were introduced in Chapter 9. In this chapter we show how strong invariants are used in practice.

No Unwanted Overflow
(Section 3.6 of the *SecSafe Document*)

This property simply says that common integer operations should not overflow. Properties of this category can be treated in the KeY system by choosing the appropriate integer semantics ($\Rightarrow$ Chap. 12).

14.5.3 Only ISOExceptions at Top Level

The KeY system provides a uniform framework for allowing and disallowing exceptions of any kind in verified programs—the KeY semantics of the diamond modality requires that all exceptions thrown during the execution of a program are caught, i.e., that the program terminates normally. However, the first property that we want to deal with states that one kind of exception is allowed to be thrown by a Java Card program—an ISOException. In JML this property is easily expressible with the signals clause. In Java Card DL this property is also expressible by a simple transformation of the program inside the modality. Section 3.6.7 describes a general framework for such transformations to deal with abrupt termination in Java Card DL (in fact, such a transformation is exactly what the JML to Java Card DL translation for the signal clauses does).

Let us give a first example of this property based on the *Demoney* method verifyPIN. This method is present (in one form or the other) in many Java Card applets, it is responsible for verifying the correctness of the PIN passed in the APDU. When the PIN is correct the method sets a global flag indicating successful PIN verification and returns. If the PIN is not correct or the maximum number of PIN entry trials has been reached an ISOException with a proper status code (including the number of tries left to enter the correct PIN) is thrown. Let us specify this in JML, without including the description of the status code of the exception:

```
─── Java + JML ─────────────────────────────────
/*@ public behavior
  requires apdu != null && length == DemoneyIO.VERIFY_PIN_LC
    && offset == ISO7816.OFFSET_CDATA
    && apdu._buffer != null
    && offset + length <= apdu._buffer.length
    && apdu._buffer[ISO7816.OFFSET_LC] ==
        DemoneyIO.VERIFY_PIN_LC;
  ensures true;
  signals (ISOException ie);
  signals_only ISOException;
  assignable ISOException._systemInstance._reason[0],
            pin._triesLeft[0], pin._isValidated[0];
@*/
private void verifyPIN(APDU apdu, short offset, byte length)...
─────────────────────────────────────── Java + JML ───
```

The precondition establishes that the apdu object passed as a parameter is well formed and contains the proper data—at least **VERIFY_PIN_LC** bytes of PIN data starting at offset **ISO7816.OFFSET_LC** + 1. The assignable clause states what are the accessible locations that verifyPIN can potentially modify. To verify the correctness of verifyPIN alone, the assignables could be

simplified to \nothing, because the postcondition does not refer to any of the assignable locations. This simplification however would make this specification unusable as a contract in other proofs and would prevent a successful proof of JML's "Assignable Proof Obligation" for verifyPIN.

To be able to prove the correctness of verifyPIN with respect to the presented specification, we need to state a few more properties about the Demoney class and the behaviour of the JAVA CARD API. Method verifyPIN accesses Demoney's private attribute pin of type OwnerPIN. Thus we need to state some facts about the attribute pin:

——— JAVA + JML ———————————————————————

```
/*@ public invariant
    pin != null && pin._maxPINSize == DemoneyIO.VERIFY_PIN_LC;
@*/
/*@spec_public@*/ private OwnerPIN pin;
```

——————————————————————————— JAVA + JML ———

A couple of method calls are made to the pin object inside verifyPIN, thus, the following invariants and method specifications in the OwnerPIN class are also needed:

——— JAVA + JML ———————————————————————

```
/*@ public invariant _maxTries > 0; @*/
/*@spec_public@*/ private byte _maxTries;

/*@ public invariant _pinArray != null
      && _pinArray.length == _maxPINSize
      && JCSystem.isTransient(_pinArray) ==
          JCSystem.NOT_A_TRANSIENT_OBJECT;
@*/
/*@spec_public@*/ private byte[] _pinArray;

/*@ public invariant _maxPINSize > 0; @*/
/*@spec_public@*/ private byte _maxPINSize;

/*@ public invariant _triesLeft != null
      && _triesLeft.length == 1
      && JCSystem.isTransient(_triesLeft) ==
        JCSystem.NOT_A_TRANSIENT_OBJECT;
@*/
/*@spec_public@*/ private byte[] _triesLeft;

// Temporary array to bypass the transaction mechanism
// during updates to _triesLeft:
/*@ public invariant _temps != null
      && _temps.length == 1
```

```
      && JCSystem.isTransient(_temps) ==
            JCSystem.CLEAR_ON_RESET;
@*/
/*@spec_public@*/ private byte[] _temps;

/*@ public invariant _isValidated != null
      && _isValidated.length == 1
      && JCSystem.isTransient(_isValidated) ==
            JCSystem.CLEAR_ON_DESELECT;
@*/
/*@spec_public@*/ private boolean[] _isValidated;

/*@ public normal_behavior
      requires true;
      ensures \result == _triesLeft[0];
      assignable \nothing;
@*/
public byte getTriesRemaining() { ... }

/*@ public normal_behavior
      requires pin != null && offset >= 0
        && length >= 0 && offset + length <= pin.length
        && length <= _pinArray.length;
      ensures true;
      assignable _triesLeft[0], _isValidated[0], _temps[0];
@*/
public boolean check(byte[] pin, short offset, byte length)...
```
—————————————————————————————— Java + JML ——

Note that specifications for both verifyPIN and check have relatively strong
preconditions. For the former the precondition guards occurrence of ISOEx-
ceptions only, for the latter it is ensured that there are no exceptions of any
kind.[7] Such a way of specifying the behaviour could be called *context depen-*
dent, that is, the preconditions are always satisfied in a given context (here,
the *Demoney* applet), and thus a contract resulting from such a specification
is always applicable in the context. On the other hand, a *context independent*
specification covers all possible behaviour of a method. For context indepen-
dent specifications the precondition is usually true and the postcondition is

---

[7] In reality, a JAVA CARD applet can also throw a multiplicity of exceptions related
to hardware or internal failures. We ignore them here, as usually they cannot
be prevented by giving a simple precondition and occurrence of such exceptions
heavily depends on the actual hardware used. Instead we only concentrate on
exceptions caused by programming errors.

very elaborate—it describes all possible outcomes of executing the method, including a long list of all possible exceptions that can occur. The first approach (context independent specifications) considerably simplifies the verification process—generated contracts are sufficient to be used in the proof, but also very simple and easy to prove. Thus, introducing context independent specifications does not bring in any advantages as far as verification is concerned (although it may be good for documentation and "manual" bug finding).

There is one more call that `verifyPIN` makes, namely, a static call to `ISOException.throwIt`. For this call, the following specification suffices:

─── JAVA + JML ───────────────────────────

```
/*@
  public static invariant _systemInstance != null;
@*/
/*@spec_public@*/ private static ISOException _systemInstance;

/*@
  public invariant _reason != null && _reason.length == 1
    && JCSystem.isTransient(_reason) ==
      JCSystem.CLEAR_ON_RESET;;
@*/
/*@spec_public@*/ private short[] _reason;

/*@ public exceptional_behavior
  requires true;
  signals (ISOException ie) ie == _systemInstance;
  signals_only ISOException;
  assignable _systemInstance._reason[0];
@*/
public static void throwIt(short reason) throws ISOException...
```
─────────────────────────────── JAVA + JML ───

┌─ **JAVA CARD Transient Arrays** ─────────────────────────

By now, a careful reader may have noticed that a lot of data in our implementation of the API classes is stored in one element arrays instead of simple primitive data fields. The reason is that array elements are the only global memory locations (that is, ones to which a reference can be kept between smart card sessions) that can be declared to be transient—not preserving its contents between single sessions. The ability to declare transient data is very important for security reasons. For example, one does not want the value of the PIN validation flag to be preserved between card sessions—for each card session the user should be required to present a valid PIN code. Moreover, array elements constitute the only type of

data that can undergo "transaction bypassing"—they can be unconditionally updated despite the fact that a transaction is in progress by calling the methods `arrayCopyNonAtomic` or `arrayFillNonAtomic` from the `Util` class ($\Rightarrow$ Sect. 9.7).

Let us now see how the same specification for `verifyPIN` is formalised in Java Card DL. The proof obligation for the method itself is the following:

```
── KeY ────────────────────────────────────────────────────
\problem {
// verifyPIN precondition
      apdu != null
   & length = DemoneyIO.VERIFY_PIN_LC
   & offset = javacard.framework.ISO7816.OFFSET_CDATA
   & apdu._buffer != null // APDU invariant
   & offset + length <= apdu._buffer.length
   & apdu._buffer[javacard.framework.ISO7816.OFFSET_LC] =
     DemoneyIO.VERIFY_PIN_LC
// Demoney invariant:
   & self.pin != null & self.pin.<created> = TRUE
   & self.pin._maxPINSize = DemoneyIO.VERIFY_PIN_LC
   & // Rest of Demoney invariant
// OwnerPIN invariant:
   & self.pin._pinArray != null
   & self.pin._pinArray.<created> = TRUE
   & self.pin._pinArray.length = self.pin._maxPINSize
   & self.pin._pinArray.<transient> =
       JCSystem.NOT_A_TRANSIENT_OBJECT
   & self.pin._triesLeft != null
   & self.pin._triesLeft.<created> = TRUE
   & self.pin._triesLeft.length = 1
   & self.pin._triesLeft.<transient> =
       JCSystem.NOT_A_TRANSIENT_OBJECT
   // Similarly for _isValidated and _temps plus
   // other parts of OwnerPIN invariant
// ISOException invariant:
   & ISOException._systemInstance != null
   & ISOException._systemInstance.<created> = TRUE
   & ISOException._systemInstance._reason != null
   & ISOException._systemInstance._reason.<created> = TRUE
   & ISOException._systemInstance._reason.length = 1
   & ISOException._systemInstance._reason.<transient> =
       JCSystem.CLEAR_ON_DESELECT
-> \<{ try{ self.verifyPIN(apdu, offset, length)@
            fr.trustedlogic.demo.demoney.Demoney;
```

```
        } catch (javacard.framework.ISOException ie) {}
    }\> ( self.pin != null & self.pin.<created> = TRUE
         & ... ) // Rest of Demoney Invariant
}
```

<div style="text-align: right">— KeY —</div>

And, as in case of JML, we also need a specification of the API methods that
`verifyPIN` makes calls to. In the `.key` file such specifications are given in
the `\contracts` section preceding the `\problem` section. For this particular
proof obligation the following contracts suffice:

— KeY —

```
\contracts {

// Name of the taclet that will be generated
OwnerPIN_getTriesRemaining {
  // Program variables used in the contract
  \programVariables {
     javacard.framework.OwnerPIN ownerPIN;
     byte result;
  }
  // OwnerPIN invariant:
    ownerPIN._triesLeft != null
  & ownerPIN._triesLeft.<created> = TRUE
  & ownerPIN._triesLeft.length = 1
  & ... // Rest of the invariant
  -> \<{
       result = ownerPIN.
         getTriesRemaining()@javacard.framework.OwnerPIN;
     }\>
     // getTriesRemaining postcondition:
     (result = ownerPIN._triesLeft[0]
     // OwnerPIN invariant:
     & ownerPIN._triesLeft != null
     & ownerPIN._triesLeft.<created> = TRUE
     & ...)
  \modifies{}
};

OwnerPIN_check {
  \programVariables {
     javacard.framework.OwnerPIN ownerPIN;
     byte[] pin; short offset; short length;
     byte result;
  }
```

```
  // OwnerPIN invariant:
    ownerPIN._pinArray != null & ...
  // check precondition:
    pin != null & offset >= 0 & length >= 0 &
    offset + length <= pin.length &
    length <= ownerPIN._pinArray.length
  -> \<{
        result = ownerPIN.check(
          pin, offset, length)@javacard.framework.OwnerPIN;
    }\>
    // check postcondition:
    (true
    // OwnerPIN invariant:
    & ownerPIN._triesLeft != null
    & ownerPIN._triesLeft.<created> = TRUE
    & ...)
  \modifies{ownerPIN._triesLeft[0], ownerPIN._isValidated[0],
    ownerPIN._temps[0] }
};

ISOException_throwIt {
  \programVariables { short reason; }

  // ISOException invariant:
    ISOException._systemInstance != null
  & ISOException._systemInstance.<created> = TRUE
  & ...
  -> \<{
      #catchAll(javacard.framework.ISOException ie) {
        javacard.framework.ISOException.throwIt(
          reason)@javacard.framework.ISOException;
      }
    }\>
      (// throwIt exceptional postcondition:
       ie != null & ie = ISOException._systemInstance
       & ie._reason[0] = reason &
       // ISOException invariant:
       ISOException._systemInstance != null & ...)
  \modifies{ISOException._systemInstance._reason[0]}
};

}
```

Note the special construct `#catchAll` in the specification of `throwIt`, which indicates that a method can throw an exception. In this case the formulation of the postcondition makes the exception obligatory (`ie != null & ...`). This is similar to the JML's `signals` clause, but in contrast to JML, `#catchAll` indicates that only an exception of the given type (here `ISOException`) can be thrown. To express the same in JML, the `signals_only` clause has to be added to the specification.

As in case of JML, these method specifications are *context* specifications— the preconditions are always satisfied in the *Demoney* applet context. The proof obligation establishing the correctness of `verifyPIN` is proven automatically in less than a minute. The strategy to use is the basic JAVA CARD DL with contracts option switched on. A call to Simplify may also be necessary to close a few remaining goals after the execution of the strategy is finished. All side proofs for the used contracts are also automatically provable in less than a minute total.

The examples we have presented so far could be proved fully automatically by the KeY system. This, however, is not always going to be the case; in many cases treatment of loops require certain amount of user interaction. In the KeY system loops are either treated with induction or loop invariant rules, both of which require user interaction. Induction proofs are discussed in detail in Chapter 11 and loop invariant rules in Chapter 3. Here we are going to discuss an example of a `while` loop in *Demoney* that can be efficiently treated with a while invariant rule with support for variants to prove absence of exceptions other than `ISOException` in the method `setTLVs`. We are not going to concentrate much on the thorough specification of `setTLVs` and other involved methods, but rather focus on the treatment of the `while` rule itself. The interesting part of the implementation of `setTLVs` is the following:

─── JAVA ───

```java
private void setTLVs(byte[] buffer, short offset, short length)
{
  ...
  short stopOffset = (short)(offset + length);

  while( offset < stopOffset )  {
    try {
      offset = setTLV(buffer, offset, ...);
    } catch(ArrayIndexOutOfBoundsException e) {
      ISOException.throwIt(ISO_7816.SW_WRONG_DATA);
    }
  }
  ...
}
```

─── JAVA ───

As it can be deduced from this code, `setTLVs` iterates over the input array `buffer` with a `while` loop, and calls another method, `setTLV`, to perform some operations on the elements of the input array `buffer`. Method `setTLV` returns an offset pointing to the elements in `buffer` to be processed in the next loop iteration. Method `setTLV` itself does not check if the input array `buffer` contains consistent data, it just tries to process it, and in case the data is in fact inconsistent, an `ArrayIndexOutOfBoundsException` is thrown. What we want to establish, is that the top level method `setTLVs` does not throw exceptions other than `ISOException`. So, what we need to prove in reality is total correctness of `setTLVs`. This has to be done efficiently without repeated symbolic execution of the loop body.

For scenarios like this, the while invariant rule with support for variants can be used. Before we discuss how the rule is applied on the example we need to gather a few facts about the loop and the method `setTLV` appearing inside the loop. To apply the while invariant rule we need to specify: (i) a suitable invariant for the loop, that is, a property that is preserved by a single loop body execution, (ii) optionally a set of variables and locations that a single loop body execution can modify, and (iii) a loop variant—a term that strictly decreases with every loop iteration. For the `setTLV` method we need to know what the range of the `offset` return value is so that a proper invariant for the loop can be constructed. The inspection of the implementation of `setTLV` reveals that the value of `offset` stays between `javacard.framework.ISO7816.OFFSET_LC + 1 == 5` and `buffer.length` (both inclusive). The return value of the `offset` is also strictly greater than the value of the `offset` passed to `setTLV`. The method can also throw `ArrayIndexOutOfBoundsException` or `ISOException`. Thus, our specification for `setTLV` needs to at least include the following facts:

—— Java + JML ——————————————————————————

```
/*@ public behavior
  requires buffer != null && offset >= 5 &&
    offset < buffer.length;
  ensures \result <= buffer.length && \result > \old(offset);
  signals (ISOException ie);
  signals (ArrayIndexOutOfBoundsException ae);
  signals_only ISOException, ArrayIndexOutOfBoundsException;
  assignable ...;
@*/
private short setTLV(byte[] buffer, short offset, ...) { ... }
```
———————————————————————————— Java + JML ——

Since we are only interested in the termination behaviour we do not specify what the method does with the data provided in the `buffer` array.

Now we are ready to give the specification for the `while` loop. A good choice for an invariant is to say that `offset` stays between 5 and `buffer.len-gth`. The only variable that the loop modifies is `offset`, and the term that is strictly decreasing with every loop iteration is `buffer.length - offset`.[8]

Having all this information we can now explain how the actual correctness proof is performed for the top level method `setTLVs` (of course, apart from what we discussed, `setTLVs` as well as `setTLV` need to be specified to a sufficient degree). The first thing to do is to choose the basic JAVA CARD DL strategy with contracts option switched on and loops option switched off. The proof is started and when the `while` loop is reached the prover stops and waits for user interaction. At this point we apply the "decreasing variant" version of the while invariant taclet. When a taclet instantiation window pops up, we need to provide the following information:

- loop variant: `buffer.length - offset`,
- modifies: `{offset}`,
- loop invariant: `offset >= 5 & offset <= buffer.length`.

This information can be also provided in the code in form of JML annotations in the following way:[9]

—— JAVA + JML ——————————————————————————————
```
/*@
   decreases buffer.length - offset;
   assignable offset;
   loop_invariant offset >= 5 && offset <= buffer.length;
@*/
while( offset < stopOffset )  { ... }
```
—————————————————————————————————— JAVA + JML ——

In this case the prover can extract this information and perform the right instantiations to apply the `while_invariant_with_variant_dec` taclet. Next we continue with executing the strategy. When the execution of the strategy stops again, all the open goals should be closable by calling Simplify. Excluding the manual interaction, the whole proof (for an equivalent JAVA CARD DL proof obligation) takes less than a minute to finish. If the JML annotation for the loop is used, the amount of interaction is negligible, and the whole process can be considered automatic. Of course, in all the cases the user still needs to come up with a suitable loop invariant.

---

[8] Based on the loop condition, the first guess for the strictly decreasing term would be `stopOffset - offset`. However, closer inspection of the code (and failed proof attempts!) reveals that it actually should be `buffer.length - offset`.

[9] The keywords used are not compliant with the current (mutating) JML standard. KeY, however, understands and treats appropriately these keywords.

Transactions and JAVA CARD DL Contracts

The real `setTLVs` method in *Demoney* also involves a transaction—before the `while` loop is executed a call to `beginTransaction` is made, and after the loop is finished the transaction is finalised with a call to `commit-Transaction`. The practical consequences are the following. Firstly, the transaction support has to be activated in the prover. Secondly, the execution of `setTLVs` can cause an implicit transaction abort by throwing an `ISOException` inside the loop body. Thus, the `transactionAbort` option needs to be set to `abortOn` as well.

However, a much more important consequence of the possible abort is the necessity to provide two separate method contracts for methods that are called within a transaction—for the abort case and for the commit case (⇒ Sect. 9.9.5). A "commit" method contract is exactly the same as the regular contract. An "abort" method contract specifies the effect of the method on the shadowed object locations (⇒ Sect. 9.5.2).Such a contract is only expressible in JAVA CARD DL. For each persistent object location that a normal contract would refer to (in the example below, contents of the *Demoney* instance array `purchaseAgentAIDBuffer` that `setTLV` modifies), an "abort" contract refers to shadowed locations denoted with a prime symbol '. Moreover, instead of referring to the `diamond` (resp. `box`) modality, an "abort" contract refers to `diamond_tra` (resp. `box_tra`) modality. For example, if a regular contract for method `setTLV` is the following (expressed in JAVA CARD DL):

--- KeY ---

```
Demoney_setTLV {

  \programVariables {
    fr.trustedlogic.demo.demoney.Demoney demoney; ... }

  buffer != null & buffer.<created> = TRUE
& offset > 0 & ...
& demoney.purchaseAgentAIDBuffer != null & ...
-> \diamond{
        #catchAll(Exception e) {
           result = demoney.setTLV(buffer, offset, ...)
             @fr.trustedlogic.demo.demoney.Demoney;
        }
  }\endmodality
    ((e = null & result > offset@pre & ...) |
     (e != null &
     (javacard.framework.ISOException::instance(e) = TRUE
       | ...))))
  \modifies{demoney.purchaseAgentAIDBuffer[*], ...}
};
```

--- KeY ---

Then the corresponding abort contract is the following:

```
── KeY ──────────────────────────────────────────────
Demoney_setTLV_tra {

  \programVariables {
    fr.trustedlogic.demo.demoney.Demoney demoney; ... }

  buffer != null & buffer.<created> = TRUE
& offset > 0 & ...
& demoney.purchaseAgentAIDBuffer' != null & ...
-> \diamond_tra{
        #catchAll(Exception e) {
            result = demoney.setTLV(buffer, offset, ...)
              @fr.trustedlogic.demo.demoney.Demoney;
        }
  }\endmodality
    ((e = null & result > offset@pre & ...) |
     (e != null &
     (javacard.framework.ISOException::instance(e) = TRUE
       | ...)))
  \modifies{demoney.purchaseAgentAIDBuffer'[*]', ...}
};
                                          ── KeY ──
```

Here `diamond_tra` is used instead of `diamond` and persistent object locations (attributes and array elements) are shadowed with a prime symbol.

An abort contract cannot be automatically generated from a regular contract, because the shadowing depends on the persistency type of the expressions involved. For example, if an array is transient, then the prime symbol should not be applied to the corresponding array access operator `[]`. Whether an array is persistent or transient cannot be easily statically detected, thus the user has to provide this information "manually" by means of a properly formulated abort contract.

Finally, whenever user interaction with the prover is required in the scope of any `_tra` modality, the user has to remember to refer to any persistent object location with a prime (or two primes if it is a second consecutive transaction, three primes if it is a third transaction, etc. (⇒ Sect. 9.5.2). For example, when applying the while invariant rule in the scope of `diamond_tra`, if the loop body modifies `obj.attr`, the user has to give `{obj.attr'}` for `#modifies` in the taclet instantiation window. An alternative is to tell the prover to use the current transaction counter, that evaluates to the right number of primes. This is achieved with `^(KeYJCSystem.<transactionCounter>)`, for example, `{obj.attr^(KeYJCSystem.<transactionCounter>)}`. When used in the scope of a "shadowed" contract the prime symbol is in fact a shorthand notation for `^(KeYJCSystem.<transactionCounter>)`.

This concludes the discussion on how to deal with `ISOExceptions` in Java Card programs. In the KeY system, the presence or absence of all kinds of exceptions can be specified and verified in the same uniform way as presented above. One more property we should mention at this point is the well-formed transactions property from the *SecSafe document*. The first part of this property says the transactions should be properly nested, that is, a transaction should not be opened if there is already one in progress, and a transaction should not be closed if none was started. In a Java Card applet the violation of this property will always give raise to a `TransactionException`. Thus, to ensure this part of the transaction well-formedness property, it is enough to specify and verify that a program does not throw a `TransactionException` at top level,[10] which is a special case of the property we have discussed in this section. The second part of the transaction well-formedness property says that the programmer should not leave an open transaction to be closed by Java Card Runtime Environment. To establish this, one more detail in the specification is needed and we discuss that in the following section, as it closely relates to atomicity and *card tear* properties.

14.5.4 Atomicity and Transactions

The atomicity property requires *related* persistent data in the applet to be updated atomically. Strong invariants ($\Rightarrow$ Chap. 9) are used to specify consistency of data at all times, so that in case an abrupt termination occurs (for example, by a card *tear*), the data (in particular, related data) stay consistent. Hence, strong invariants seem to be the right technique to deal with consistency properties related to atomic updates. Java Card transaction mechanism is the facility that the programmer can use to ensure atomicity of arbitrary blocks of a Java Card program, so it seems natural to also deal with transaction properties in the context of atomicity properties. Thus, we will also discuss the second part of the transaction well-formedness property (do not let the JCRE close an open transaction) in this section.

JML does not support strong invariants, and the ability to reason about transactions (and also object persistency) is limited to what can be expressed with Java Card API calls related to transactions and object persistency (one possible solution to this problem was proposed by Hubbers and Poll [2004a]). Therefore, in the context of this section, we limit ourselves to expressing the properties only in Java Card DL.

---

[10] In reality, `TransactionException` can also be thrown due to the transaction commit buffer exhaustion, but in our Java Card model we assume that to be an internal smart card problem (lack of memory), and we stated already that these are ignored. The downside of ignoring this problem is that Java code that "overuses" the transaction mechanism will not be detected. Proper modelling of memory consumption, which will resolve this problem, is a subject of future research ($\Rightarrow$ Sect. 14.5.6).

To discuss the atomicity property we use the example of the perform-
Transaction method in *Demoney*. One of the responsibilities of the *Demoney*
applet is to record information about each purchase made with the electronic
purse in the log file. Among other things, the current balance after the pur-
chase is recorded in a new log entry. As the *SecSafe document* points out
accurately, when atomic consistency properties are considered, one has to
be able to say what it means for the data to be related. In principle, such
relations cannot be deduced automatically, because of the specific logics and
semantics of the considered applet. In our example we want to state that
the current balance of the purse is always the same as the one recorded in
the most recent log entry. Debiting the purse balance and updating the log
file in an atomic fashion is exactly what the performTransaction method
is responsible for, and, not surprisingly, it uses the JAVA CARD transaction
mechanism to ensure atomic update of the involved data.

In JAVA CARD DL, to express a strong invariant property, the through-
out modality is used (⇒ Chap. 9). Thus, the proof obligation to ensure our
example atomicity property for performTransaction reads:

—— KeY ——

```
\problem {
// No transaction in progress when the method is called:
   JCSystem._transactionDepth = 0 &
// performTransaction precondition - method arguments:
   apduBuffer != null &
   apduBuffer.<created> = TRUE & apduBuffer.length >= 45 &
   apduBuffer[ISO7816.OFFSET_LC] =
     DemoneyIO.COMPLETE_TRANSACTION_LC &
   offsetTransCtx =
     DemoneyIO.COMPLETE_TRANSACTION_OFF_TRANS_CTX &
// Demoney & CyclicFile invariants:
   self.logFile != null & self.logFile.<created> = TRUE &
   self.logFile.records != null &
   self.logFile.records.<created> = TRUE &
   self.logFile.recordLength = DemoneyIO.LEN_LOG_RECORD &
   self.logFile.records.length > 0 &
   self.logFile.nextRecordIndex >= 0 &
   self.logFile.nextRecordIndex < self.logFile.records.length &
// performTransaction precondition - the log entry to be
// filled in by performTransaction is properly allocated:
   jbyte[]::instance(
     self.logFile.records[self.logFile.nextRecordIndex]
   ) = TRUE &
   ((jbyte[])self.logFile.records[
     self.logFile.nextRecordIndex]).length
       = DemoneyIO.LEN_LOG_RECORD &
```

```
// Strong invariant (balance in the recent log entry is equal
// to current balance) holds before performTransaction
// is called. #recentIndex is equivalent to:
// self.logFile.nextRecordIndex - 1 %
//      self.logFile.records.length,
// see explanations below.
   jvmMakeShort(
     ((jbyte[])self.logFile.records[#recentIndex])
       [DemoneyIO.LOG_RECORD_OFF_NEW_BALANCE],
     ((jbyte[])self.logFile.records[#recentIndex])
       [DemoneyIO.LOG_RECORD_OFF_NEW_BALANCE + 1]) =
   self.balance
->
 \[[{
    self.performTransaction(amount,apduBuffer,offsetTransCtx)
      @fr.trustedlogic.demo.demoney.Demoney;
 }\]]
// Strong invariant holds throughout the execution
// of performTransaction
   jvmMakeShort(
     ((jbyte[])self.logFile.records[#recentIndex])
       [DemoneyIO.LOG_RECORD_OFF_NEW_BALANCE],
     ((jbyte[])self.logFile.records[#recentIndex])
       [DemoneyIO.LOG_RECORD_OFF_NEW_BALANCE + 1]) =
   self.balance
}
```

─── KeY ─────

This proof obligation seems to be more complex than what has been presented so far, and indeed some explanations are due. First of all, since the method utilises the JAVA CARD transaction mechanism to ensure atomicity, we need to assume that there is no transaction in progress when performTransaction is called. Secondly, a precondition establishing correctness of the method parameters is needed—that the apduBuffer array contains expected data of proper size. Then we also need to assume that the log record data structure (which is basically a two-dimensional byte array) is properly allocated.

The rest of the assumptions in the proof obligation relate to the strong invariant itself and the parts of the log record that are affected by performTransaction and referred to in the strong invariant. The first complication in expressing the strong invariant is due to the way the log record data structure in *Demoney* is defined—it is a two-dimensional byte array, where the first index points to a given log entry, and the second index points to the actual record data. Since JAVA CARD only allows one-dimensional arrays, a workaround in the *Demoney* code has been introduced, namely, first a one-dimensional array of objects is allocated:

```Java
Object[] records = new Object[...];
```

and then each entry in this array is associated with a byte array:

```Java
records[i] = new byte[...];
```

Because of this, the `records` array lacks static type information. This results in type casts in the *Demoney* code, and necessity to introduce corresponding type casts in the proof obligation above.

The second complication in expressing the strong invariant is caused by the fact that in the `CyclicFile` class only the index to the next record to be used is kept, and in the specification we need to refer to the most recently used record. Since the records are stored in a cyclic fashion (that is, each new record overwrites the oldest one) the next record index is advanced in the following way:

```Java
nextRecordIndex = (short)((short)(nextRecordIndex + 1) %
                         records.length);
```

To construct an index to the most recently used log entry, in JAVA we would have to write something like:

```Java
recentIndex = (short)((short)(nextRecordIndex - 1) %
                  records.length);
```

Thus, we need to give an equivalent expression in our proof obligation:

```KeY
jmod(self.logFile.nextRecordIndex - 1,
  self.logFile.records.length)
```

In the proof obligation above this expression has been abbreviated with `#recentIndex` for clarity. The symbolic execution of the statement that advances the `nextRecordIndex` and the form of the expression referring to the most recent log entry will result in quite complex integer expressions in the proof. To help the prover resolve such expressions easily, without requiring user interaction, we need to provide the following integer simplification taclet:

```
──── KeY ────────────────────────────────────────────

\rules(intRules:aritmethicSemanticsIgnoringOF) {
 modulo_index_prop {
   \schemaVar \term int #index, #size;
   \find(jmod(jmod(#index + 1, #size) - 1, #size))
   \replacewith(#index)
   \heuristics(simplify_int)
 };
}
──────────────────────────────────────────── KeY ────
```

This taclet simply states that the following mathematical fact in integer rings in always true:

$$index = (((index + 1) \mod size) - 1) \mod size .$$

The rule is only valid in natural arithmetic integer semantics (thus a taclet option indication right after the keyword **\rules**) and obviously needs to be proven correct as well ($\Rightarrow$ Sect. 4.5).

Finally, the strong invariant also contains a function symbol jvmMake-Short. The value of the balance, which is of type short, is stored in two elements of a byte array in the log record. The function symbol jvmMake-Short is used to express that two byte values (function parameters) are composed to form a short value (function result). A proper implementation of setShort, getShort, and makeShort, together with a couple of taclets to execute the "built-in" methods ($\Rightarrow$ Sect. 14.4.1), ensure that the strong invariant formulation above is sufficient to prove the desired property.

The correctness proof for performTransaction is most efficiently done without using any method contracts. Most of the method calls are to special methods ($\Rightarrow$ Sect. 14.4.1), and these are usually more efficiently treated by specialised taclets. The only other method that could be specified with a contract is appendRecordData from class CyclicFile. However, because of resulting complex update substitutions, using a contract for this method gives a proof with a few open goals that require non-trivial user interaction. Without using the contract the performTransaction proof obligation is proved almost automatically (two simple interactions and a call to Simplify) in just above two minutes.

This, however, does not conclude the verification of performTransaction. Two more issues to take care of are termination and well-formed transactions. The throughout modality is partial in its nature, thus, proving that the method performTransaction preserves the strong invariant does not mean that the method terminates or does anything useful. In fact, it can be the case that if the precondition is not strong enough the symbolic execution of performTransaction terminates shortly with an exception, which trivially satisfies the strong invariant. So, in the worst case, it is possible that we have

proved the preservation of a strong invariant, when no actual change to the data involved in the strong invariant has been made. Therefore, it is crucial to also prove that the method terminates in a non-abrupt fashion with the same preconditions as used for the proof of the strong invariant.

We also have not yet discussed the second part of the transactions well-formedness property, saying that the program should not leave an open transaction to be closed by the JAVA CARD Runtime Environment. The information about open transactions can be extracted from the JAVA CARD API through the static attribute _transactionDepth of the class JCSystem. To state that a program does not leave an open transaction it is enough to put the following expression in the postcondition:

——— KeY ———

```
JCSystem._transactionDepth = 0
```

——— KeY ———

It was also necessary to include this expression in the precondition of per-formTransaction to ensure that the proper nesting of transactions is not violated.

The termination property of performTransaction and transaction well-formedness can be expressed in one proof obligation, as follows:

——— KeY ———

```
\problem {
// No transaction in progress:
   JCSystem._transactionDepth = 0 &
// performTransaction precondition - method arguments:
   ...
// Demoney invariants:
   ...
// performTransaction precondition - the log entry to be
// filled in by performTransaction is properly allocated:
   ...
->
 \<{
    fr.trustedlogic.demo.demoney.Demoney()::
     self.performTransaction(amount,apduBuffer,offsetTransCtx);
 }\> JCSystem._transactionDepth = 0
}
```

——— KeY ———

Since this proof obligation does not involve the strong invariant or throughout modality, this specification can be also expressed in JML. We leave this as an exercise for the reader.

14.5.5 No Unwanted Overflow

Finally, we deal with a property purely related to integer arithmetic. The informal formulation of the property is really simple—additions, subtractions, multiplications and negations must not overflow. Formal treatment of such properties is a bit more complex though, the whole Chapter 12 is devoted to this, here we only summarise. To deal with all possible issues related to integer arithmetic, in particular overflow, the KeY system uses three different semantics of arithmetic operations. The first semantics treats the integer numbers in the idealised way, that is, the integer types are assumed to be infinite and, thus, not overflowing. The second semantics bounds all the integer types and prohibits any kind of overflow. The third semantics is that of JAVA, that is, all the arithmetic operations are performed as in the JVM, in particular they are allowed to overflow and the effects of overflow are accurately modelled.

  To illustrate how the different semantics can be used to deal with the no unwanted overflow property, we use an example remotely related to *Demoney*, that is, it is not based on the actual *Demoney* code, but it is quoted in the *SecSafe document*. First let us look at a proof obligation with a badly formed program with respect to overflow:

```
——— KeY ———
\problem {
  inShort(balance) & inShort(maxBalance) & inShort(credit) &
  balance > 0 & maxBalance > 0 & credit > 0 ->
  \<{ try {
      if ((short)(balance + credit) > maxBalance)
        ISOException.throwIt(SW_CREDIT_TOO_HIGH);
      else
        balance += credit;
    }catch(javacard.framework.ISOException e){}
  }\> balance > 0
}
                                              ——— KeY ———
```

The problem in this program is that the `(short)(balance + credit)` operation can overflow making the condition inside the `if` statement false resulting in a `balance` being less than 0 after this program is executed. When processed by the KeY system with the idealised integer semantics switched on, this proof obligation is proved quickly. When the arithmetic semantics with overflow control is used this proof obligation is not provable. The fix to the program to avoid overflow is to change the `if` condition in the following way:

```
─ JAVA ─────────────────────────────────────────
    if (balance > (short)(maxBalance - credit))
─────────────────────────────────────── JAVA ─
```

Here, since both `maxBalance` and `credit` are strictly positive, the result of
subtraction will always stay within limits of the `short` integer type. A border-
line case is when `maxBalance` is 1 and `credit` is 32767—the result is then
−32766, which is greater than the minimum allowed `short` value −32768.
This proof obligation is provable with both kinds of integer semantics. For
further discussion about handling integer arithmetic we refer the reader to
Chapter 12.

14.5.6 Other Properties

There are other security properties mentioned in the *SecSafe document*, which
are either not (yet) fully supported in the KeY system, or are beyond the
scope of this book. We briefly discuss those properties here.

Memory Allocation

Due to restricted resources of a smart card, one of the requirements on a
properly designed JAVA CARD applet is the constrained memory usage. This
includes bounded dynamic memory allocation and no memory allocation in
certain life stages of the applet. This seems like a problem strictly related to
syntax-oriented static analysis, because in general there is no need for precise
analysis of the control flow. However, some cases would require precise analy-
sis anyhow, for example, if memory allocation is performed inside a loop, the
precise loop bound has to be known. Either way, we believe that this prop-
erty in general can be formalised and proved with the KeY system as well,
provided some simple extensions to the JAVA CARD DL are added. The main
idea is the following. To model certain aspects of the JAVA virtual machine, in
particular object creation, the JAVA CARD DL refers to a set of implicit fields
defined for every object (⇒ Sect. 3.6.6). For example, each type of object
contains an implicit reference `<next>`, which points to the object of the same
type that was created next after this one—the JAVA CARD DL rules that
handle object creation are responsible for updating the state of the `<next>`
reference in the proof. There is no obstacle to introduce a new static implicit
field to our JAVA model that would keep track of the amount of allocated
memory or the possibility to allocate memory. There are, of course, certain
technical details with respect to the accuracy of the memory consumption es-
timations, for example, an object may consume different amounts of memory
depending on the actual JVM used. Thus, keeping precise record of the allo-
cated memory may not be possible and only close approximations could be
achieved with this solution. Relating to this problem, an approach to perform

memory consumption analysis is presented in Barthe et al. [2005], based on
the JACK tool [Burdy et al, 2003] and semi-automatic generation of JML
annotations that describe memory usage of an applet.

Conditional Execution Points

This property says that certain program points must only be executed if
a given condition holds. This is not a difficult task, for example, JML and
JAVA itself (version 1.4 onwards) provide means to annotate any program
point with a boolean condition (the `@assert` clause in JML and the `assert`
statement in JAVA). ESC/JAVA2 can easily check (to a certain limit) JML
`@assert` clauses. A thorough treatment of such properties in the KeY system
would require us to introduce a generalisation of strong invariants which
are supposed to hold after every program statement. For the generalised
case, such a property would differ depending on the actual program point.
Thus, there are no theoretical obstacles here, and the future versions of the
KeY system will certainly provide support for this property (and also JML's
`@assert` clause).

Information Privacy and Manipulation of Plain Text Secret

Those two properties fall into the category of data security properties. As
it has been shown in [Darvas et al, 2005, Pan, 2005], formalising and prov-
ing such confidentiality properties can be achieved with interactive theorem
proving using the KeY system. This is still a subject of ongoing research and
detailed discussion is beyond the scope of this book. In particular, no exper-
iments on a real JAVA CARD code that we could present has been performed.

14.6 Lessons

The results presented in this chapter prove that the KeY system is highly
capable of performing serious verification of unmodified/legacy JAVA CARD
code with respect to industry related properties. However, to achieve this
level of KeY system's functionality, the following lessons we have learned
have to be kept in mind.

   The performance results presented here would suggest that the time spent
on verification could be considered not to be an issue. As this might be true for
verification itself, one has to remember that most of the time is actually spent
on developing the correct specifications—the prerequisite for verification. In
the context of this work, writing specifications was an iterative, trial and error
process. What is in particular difficult, is to find the right preconditions for
the correct execution of a method, in particular when a preexisting or legacy
code like the *Demoney* applet is considered. Although it has not been used in

the context of this work, the KeY system supports automatic construction of preconditions, based on failed proof attempts. This is described in detail by Platzer [2004a]. The basic idea behind computing the specification is to try to prove a total correctness proof obligation. In case it fails, all the open proof goals are collected and the necessary preconditions that would be needed to close those goals are calculated. There are two disadvantages to this technique: (1) for the proof to terminate the preconditions that guard the loop bounds cannot be omitted, so there is no way to calculate preconditions for loops, they have to be given beforehand, (2) proofs have to be performed the same way for computing the specification as it is done when one simply tries to prove the obligation, so computing the specification is in fact a front-end for analysing failed proof attempts in an organised fashion. Moreover, the specifications produced can be equally hard to read as is analysing the failed proof attempt manually. More ideas about specification engineering can be found in Chapter 5, which is fully devoted to this subject.

The second important thing to keep in mind is the partial correctness issue in the context of strong invariants. As we pointed out, the throughout modality used to write proof obligations for strong invariants is partial. Thus, in many cases it is important to verify the termination property apart from the strong invariant preservation property to ensure that the strong invariant is not satisfied trivially.

The examples presented in this chapter clearly show that highly automatic and very efficient verification can be achieved with the KeY system. In large part this is because the KeY system and the Java Card DL are designed in a way not to bother the user with the workings of the calculus and the proof system. However, we have realised that some verification decisions can further support automation and affect the performance. For example, we gave an example of a method (`appendRecordData`) which is more efficiently handled when no contract for the method is used, but instead the method's implementation is in-lined into the proof. Apart from isolated examples like this, proof modularisation by using method contracts is vital for successful, scalable, and efficient verification. Although the KeY system provides powerful support for such specifications both in JML and Java Card DL, it is still up to the user to construct the specifications, which, as we pointed out already, is tedious work, and do so in a way that no unnecessary complications are introduced into the proof (that is, try to construct *context* specification).

To further support the verification process, we introduced a small number of additional simplification rules for arithmetic expressions in the context of the `performTransaction` method. Such rules considerably simplify the proof, but introducing them, although being relatively easy, requires a little bit more than the basic understanding of Java Card DL. Moreover, each introduced rule has to be proven sound ($\Rightarrow$ Sect. 4.5). Yet another place, where the prover had to be supported by the user, was providing the `while` loop specification in the `setTLVs` method. General discussion on how to (semi-)automatically

calculate or provide such specifications is beyond the scope of this chapter, but a good starting point is Chapter 11 that deals with induction proofs in the KeY system. Finally, the size of the proofs that involve JAVA CARD transaction statements can be greatly reduced by choosing the `abortOff` taclet option in cases where there is no call to `abortTransaction` in the verified code (explicit or implicit).

We have shown that it is possible to express similar properties in both JML and directly in JAVA CARD DL. Both of the approaches have their advantages and disadvantages ($\Rightarrow$ Sect. 14.3). It seems that in the context of this chapter, the JAVA CARD DL approach is more practical. JAVA CARD DL proof obligations can be customised for a given security property and fine tuned to ease verification. The new proof obligation generation mechanism ($\Rightarrow$ Chap. 8) will also provide the possibility to customise JML proof obligations in a similar way. Nevertheless certain properties, like strong invariants and other transaction related properties, are only expressible in JAVA CARD DL.

14.6.1 Related Work

To finish this chapter we should briefly discuss some other work reporting on serious verification attempts of JAVA CARD software on the source code level in the context similar to ours. Jacobs et al. [2004] discuss verification of a commercial JAVA CARD applet with different verification tools. The use of the following JAVA and JAVA CARD verification tools is discussed: ESC/JAVA2 (successor of ESC/JAVA—Flanagan et al. [2002]), KRAKATOA [Marché et al., 2004], JIVE [Meyer and Poetzsch-Heffter, 2000], and LOOP [Jacobs and Poll, 2003]. The security property under consideration, one of the properties we discussed, is that only `ISOExceptions` are thrown at the top level of the applet. Jacobs et al. detected subtle bugs in the applet with respect to a possible uncaught `ArrayIndexOutOfBoundsException` (with LOOP and JIVE tools), as well as full verification (no exceptions other than `ISOException`, satisfied postcondition, and preserved class invariant) of single methods with the KRAKATOA tool. The paper admits that expertise and considerable user interaction with the back-end theorem provers (PVS and COQ) were required. The paper also discusses precondition generation, the same problem we encountered in our work. One of the solutions proposed by Jacobs et al. is to use ESC/JAVA2 to construct preconditions. In short, the tool is run interactively on an unspecified applet, which results in warnings about possible exceptions. Such warnings are removed step by step by adding appropriate expressions to the precondition. Alternatively the weakest precondition calculus of the JIVE system could be used by running the proof "backwards", that is, by starting with a postcondition and calculating the necessary preconditions. This however, has not been presented in the paper and to our understanding the approach has certain limitations.

Breunesse et al. [2002] present a case study in verifying the correctness of a JAVA class representing limited precision decimal numbers (`Decimal` class). The verification tool used there is the LOOP tool. In this case the main goal was to fully verify a very complex functional behaviour of the `Decimal` class. During the process non-trivial bugs have been found and the `Decimal` class was reimplemented.

Finally, Pavlova et al. [2004] discuss specification and verification of some properties from the *SecSafe document*. The two main properties that are discussed are no `ISOExceptions` at top level and well-formed transactions. The paper discusses a mechanism to automatically augment JAVA CARD programs with JML annotations to enforce those properties and mentions successful verification of the properties with the JACK, JIVE, KRAKATOA, LOOP, and ESC/JAVA tools.

15

The Schorr-Waite-Algorithm

by
Richard Bubel

The Schorr-Waite graph marking algorithm named after its inventors Schorr
and Waite [1967] has become an unofficial benchmark for the verification of
programs dealing with linked data structures.

It has been originally designed with a LISP garbage collector as applica-
tion field in mind and thus, its main characteristic is low additional memory
consumption. The original design claimed only two markers per data object
and, more important, only three auxiliary pointers at all during the algo-
rithm's runtime. It is the latter point, where most other graph marking al-
gorithms lose against Schorr-Waite and need to allocate (often implicitly as
part of the method stack) additional memory linear in the number of nodes
in the worst case. These resources are used to log the taken path for later
backtracking when a circle is detected or a sink reached.
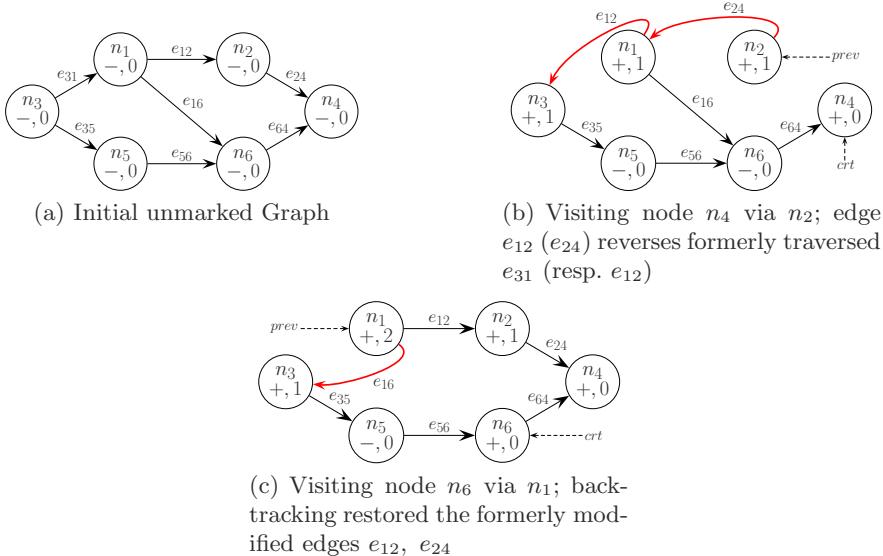
Schorr and Waite's trick is to keep track of the path by reversing traversed
edges offset by one and restoring them afterwards in the backtracking phase
of the algorithm. A detailed description including the Java implementation
to be verified is given in Section 15.1.

Formal treatment of Schorr-Waite is challenging as reachability issues are
involved. Transitive closure resp. reachability is beyond pure first-order logic
and some extra effort has to be spent to deal with this kind of problems (see
[Beckert and Trentelman, 2005] for a detailed discussion). On the other side,
the algorithm is small and simple enough to serve as a testbed for different
approaches. We introduce a notion of reachability as part of Sect. 15.2 and
come back to it for the actual verification, which makes up most of Sect. 15.3.

15.1 The Algorithm in Detail

15.1.1 In Theory

As usual a *directed graph* G is defined as a set of vertices V and edges $E \subseteq V \times V$. The directed edge $s \rightharpoonup t \in E$ connects source node $s \in V$ with target

(a) Initial unmarked Graph

(b) Visiting node n_4 via n_2; edge e_{12} (e_{24}) reverses formerly traversed e_{31} (resp. e_{12})

(c) Visiting node n_6 via n_1; backtracking restored the formerly modified edges e_{12}, e_{24}

Fig. 15.1. Illustration of a Schorr-Waite run: curved edges have been modified to encode the taken path; pointers crt, $prev$ refer to the current resp. the previous node

node $t \in V$, but not vice versa. We call node t a direct *successor* of node s (resp. s a direct *predecessor* of t).

For sake of simplicity, we require that each edge e is labelled with a unique natural number $l(e)$ where $l : E \to \mathbb{N}$. The labelling allows us to put an order on all outgoing edges $e_i := s \rightharpoonup_i t_i, i \in \{1, \ldots, n\}$ of a node s, which complies with the natural number ordering $\leq$ of the corresponding labels $l(e_i)$.

When speaking of visiting all children (of a node s) from left-to-right, we mean in fact that all direct successors of s are accessed via its outgoing edges in ascending order of their labels. We refer to the target node of the edge with the i-th smallest label of all outgoing edges of node s as the node's i-th child.

In addition, each node is augmented with a flag `visited` and an integer field `usedEdge`, which is used to store the number of the most recently visited child via this node (or equivalently the corresponding edge label).

In the subsequent four additional pointers are required:

- `current` and `previous`, whose intended purpose is to refer to the currently respective previously visited node and
- the two helpers `next` and `old`.

Given a directed graph G as for example shown in Fig. 15.1 and a designated node s, here: n_3, Schorr-Waite explores G starting at node s applying a left depth-first strategy:

1. Outgoing from the currently visited node `current` the leftmost not yet visited child `next` is selected and the taken edge e redirected to target the node referenced by pointer `previous`. The `usedEdge` field of `current` is used to keep the label $l(e)$ of the reversed edge in order to restore edge e later in the backtracking phase (step 2). Afterwards `previous` is altered to point to our current node, while pointer `current` moves onto node `next`. Finally, the new `current` node becomes marked as visited. Continue with step 1.

2. If all children of the node referred to by `current` have already been visited or it is a childless node and $current \neq s$, then a backward step is performed. Therefore the edge via which `current` has been accessed and remembered in the `usedEdge` field of node `previous` during step 1, is restored: this means to redirect it to its original target `current`, but not before rescuing its current target using pointer `old`. Now pointer `current` can be reset to the node referenced by `previous` and—last but not least—`previous` is moved back to node `old`. Continue with step 1.

After all reachable nodes have been visited the algorithm terminates when after a backtracking step the starting node s is reached. At this time the original graph structure has been also restored.

15.1.2 In Practice

The design of our JAVA implementation to be verified is illustrated in Fig. 15.2. The graph nodes are modelled as instances of class `HeapObject`, where each instance contains a `children` array, whose i-th component contains the node's i-th child.
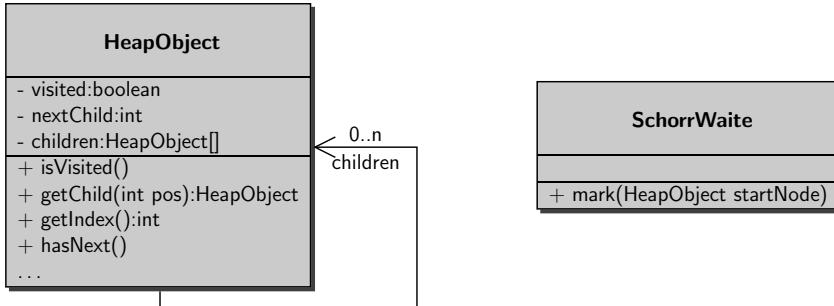
Fig. 15.2. Class diagram showing the involved participants

All `HeapObject` instances provide a rudimentary iterate facility to access their children. Therefore, they implement an integer index field, which contains the array index of the child to be visited next. Method `hasNext` tests if the index field has reached the end of the array and therewith all children of the node

have been accessed. The index field is used to realise the `usedEdge` field of the previously given description. In fact, `usedEdge` is equal to the value stored in the index field minus one.

The JAVA implementation of the algorithm itself is realised as method `mark` of class `Schorr-Waite` shown in Fig. 15.3. Invoking `mark` with a non-null start node handed over as argument starts the graph traversal. The method assumes that before it is invoked, all `HeapObject` instances have no marks set.

```java
public void mark(HeapObject startNode) {
    HeapObject current  = start;
    HeapObject previous = null;
    HeapObject next     = null;
5   HeapObject old      = null;

    startNode.setMark(true);

    while (current != startNode || startNode.hasNext()) {
10      if (current.hasNext()) {
            final int nextChild = current.getIndex();
            next = current.getChild(nextChild);
            if (next != null && !next.isMarked()) {
                // forward scan
15              current.setChild(nextChild, previous);
                current.incIndex();
                previous = current;
                current  = next;
                current.setMark(true);
20          } else {
                // already visited or no child at this slot
                // proceed to next child
                current.incIndex();
            }
25      } else {
            // backward
            final int ref2restore = previous.getIndex() - 1;
            old = previous.getChild(ref2restore);
            previous.setChild(ref2restore, current);
30          current  = previous;
            previous = old;
        }
    } }
```

Fig. 15.3. Core of the Schorr-Waite algorithm

The first lines of method `mark` initialise the required pointers `current`, `previous`, `next` and `old`. The starting node *s* of the previous section is handed over as the method's argument referred to by parameter `startNode`. It becomes also the first node `current` points to. All other pointers are set to `null` at first. Before the while loop is entered, the starting node `startNode` is marked.

After these preparations the graph will be traversed in left depth-first order as long as pointer `current` has not yet returned to the starting node `startNode` or if node `startNode` still has children that need to be checked.

If node `current` has a child left, a forward step is performed (lines 10–25):

line 12 the `nextChild`th component of node `current`'s children array (that is the `current`'s next not already accessed child) is assigned to variable `next`

lines 14–19 these lines are entered in case that the `HeapObject` instance referred to by `next` has not already been visited, which is tested by method `isMarked` in the conditionals guard. First the taken edge, i.e., the `nextChild`th component of `current`'s children array, has to be redirected to the node variable `previous` refers to (line 15). In the succeeding lines, field `nextChild` of node `current` is updated, pointer `previous` is moved toward the node `current` refers to, and finally, node `next` becomes the new `current` node. At the end the new `current` node is marked as visited with help of method `setMark`.

line 23 is only executed in case that node `next` has been already marked in a previous step

Lines 26–32 are executed if node `current` has no remaining children to be visited. In this case a backward step has to be performed:

line 28 the `nextChild-1`th child of node `previous`, which stores the penultimate node is memorised in `old`

line 29 the `nextChild-1`th children array component of node `previous` is restored, i.e., redirected to node `current`

lines 30–31 finally, `current` becomes `previous` and `previous` is set to the node stored in `old`.

15.2 Specifying Schorr-Waite

A correct implementation of Schorr-Waite must guarantee at least these requirements:

First, the algorithm invoked on an arbitrary node of a finite graph must terminate. And second, in its final state it is ensured that

1. When initially invoked on an unmarked graph, a node has been marked if and only if it has been reachable from the starting node.

2. The graph structure has not been modified, i.e., after the algorithm terminates all components of the children arrays contain their original value again.

We concentrate on the specification and verification of the first requirement. The second one is a relative straightforward extension to the first one and can be specified and proven in a similar manner.

15.2.1 Specifying Reachability Properties

Reasoning about linked or recursive data structures requires some notion of reachability of objects. Therefore, we define a reachable predicate that allows to express that an object is reachable from another one via a specified set of fields *Acc*.

In order to express these properties we have to define a variant of non-rigid predicate (function) symbols equipped with a list of locations/accessor expressions that are allowed to be used in order to navigate between objects. We call them *non-rigid symbols with explicit dependencies*.

A New Class of Symbols

A first step in the definition of these symbols, is to define a notion of accessor expressions. You might want to look ahead to Example 15.9 to get an idea what we are aiming at.

Definition 15.1 (Accessor expression). *The set AE of* accessor expressions *is inductively defined as follows:*

1. $(\cdot)_C.\mathtt{a}@(C)$ *is an accessor expression for all attributes $\mathtt{a}$ declared in a class type C,*
2. $f(*,\ldots,*,\overbrace{a}^{i},*,\ldots,*)$ *is an accessor expression for any arbitrary expression $a \in AE$, n-ary function symbol f, $1 \le i \le n$ and the sort of a is compatible with the i-th argument sort of f.*

Note 15.2. Any accessor expression *acc* contains exactly one placeholder $(\cdot)_C$. We will write *s.acc* when we mean to replace $(\cdot)_C$ by a term s with $sort(s) \le C$. The suffix $@(C)$ to the attribute $\mathtt{a}$ only serves to disambiguate attributes with the same name in different classes.

Example 15.3. Let `List` denote a class type declaring an attribute `next` of the same type. Further, let `ASTNode` be another class type declaring an attribute `children` of array type `ASTNode[]`. Then both $(\cdot)_{\mathtt{List}}.\mathtt{next}$ and $(\cdot)_{\mathtt{ASTNode}}.\mathtt{children[*]}$ are accessor expressions. When there is only one obvious choice for the placeholder $(\cdot)_C$ we omit it. We will thus write these two accessor expressions as `next` and `children[*]`.

Definition 15.4 (Syntax and signature: non-rigid symbols with explicit dependencies). *The non-rigid predicate/function symbol $p[acc\_list;]$:* $T_1 \times \cdots \times T_n$, *resp.,* $f[acc\_list;] : T_1 \times \cdots \times T_n \to T$, *where acc_list is a semicolon separated list of accessor expressions are called* non-rigid symbols with explicit dependencies *.*

Terms and formulas are defined as before($\Rightarrow$ Sect. 3.2) with the only difference that the corresponding sets of function and predicate symbols may contain these newly introduced symbols.

Definition 15.5 (Semantics). *Let $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$ denote a JAVA CARD DL Kripke structure. Then for any two states S_1 and $S_2 \in \mathcal{S}$, the predicate $p[acc\_list;](t_1, \ldots, t_n)$ evaluates to the same truth value, if S_1 and S_2 coincide on the interpretation of all accessor expressions, i.e., for any $acc \in acc\_list$ with placeholder $(\cdot)_C$, and for all $u, v \in Term_\Sigma^C$ with $val_{S_1}(u) = val_{S_2}(v)$ the following holds: $val_{S_1}(u.acc) = val_{S_2}(v.acc)$. In case of nested accessor expressions, both states have to coincide on the constituents and in case of an array expression on the length attribute as well. Analogous for function symbols.*

The Reachable Predicate

The reachable predicate is a representative of a non-rigid predicate with explicit dependencies. Its syntax is defined as follows:

Definition 15.6 (Reachable predicate, syntax). *The ternary non-rigid predicate* `reach[`*acc_list;*`](`T`,`T`,int)` *is called* reachable predicate, *where acc_list is a semicolon separated list of accessor expressions, whose usage is allowed in navigation expressions.*

Example 15.7. Let o, u and s, t be program variables of type `List` resp. `ASTNode` and n be an arbitrary integer constant, then `reach[`*next;*`](o,o,0)`, `reach[`*next;*`](o,u,n)` and `reach[`*children[*$*$*];*`](s,t,n)` are syntactical correct JAVA CARD DL formulas.

The semantics of the reachable predicate needs to be defined. The definition has to adhere to the constraint given in Def. 15.5.

Definition 15.8 (Reachable predicate, semantics). *The reachable predicate* `reach[`*acc_list;*`](o,u,n)` *is valid in a state s iff. $s \models \exists p_1, \ldots, p_m; (o.a_1.\ldots.a_n \doteq u)$ with accessor expressions $a_i \in acc\_list$ and a logic variable p_j for each $* \in \{a_1, \ldots, a_n\}$ of the corresponding type.*

Example 15.9. Let terms o, u denote two objects of type T and term n a positive integer. The formula `reach[`*next;*`](o,u,n)` is valid in a state s iff $s \models o.\underbrace{next.\cdots.next}_{n} \doteq u$. Let t_1, t_2 be terms of type `ASTNode` then `reach[`*children[*$*$*];*`](t_1,t_2,2)` is valid in state s iff $s \models \exists p_1 \exists p_2; (t_1.children[p_1].children[p_2] \doteq t_2)$.

Furthermore `reach[`*acc_list;*`](o,u,0)` will always be equivalent to $o = u$.

The taclet reachableDefinition representing the semantics definition of the reachable predicate in its instance for $acc\_list = children[*]$; can be written as follows:

```
——— KeY ———————————————————————————————
 reachableDefinition {
   \find(reach[children[*];](t1, t2, n))
   \varcond(\notFreeIn(k, t1, t2, n))
   \replacewith(t1 = t2 & n = 0 |
       (t1 != null & n > 0 &
        \exists k;( k>=0 & k<t1.children@(HeapObject)).length &
         reach[children[*];*.children.length;]
            (t1.children@(HeapObject)[k], t2, n-1))) )
 };
                                              ——— KeY ———
```

The given rule defines reachable recursively, but is well-founded. Instead of reach[children[*];] the slightly shorter form reach[children[*];] will from now on be used in order to keep the formulas readable.

If n is negative, it will evaluate to false due to n >= 0. If n is equal to zero then t1 = t2 must hold. This is the only case where t1 may be null. If n > 0 then it must hold that t2 is reachable in n − 1 steps from an object stored in the children array of t1.

Together with induction over the natural numbers, rule reachableDefinition suffices to express the required reachable properties. But it would not be very convenient and, therefore, a number of further taclets exists covering common situations directly

```
——— KeY ———————————————————————————————
 reachableDefinitionBase {
     \find(reach[children[*];](t1, t2, 0))
     \replacewith(t1 = t2)
 };

 reachableDefinitionFalse {
     \assumes (n < 0 ==>)
     \find(reach[children[*];](t1, t2, n)) \sameUpdateLevel
     \replacewith(false)
 };
                                              ——— KeY ———
```

Encoding the Backtracking Path

For the specification of the loop invariant it turns out to be useful to define a relation $onPath$, which describes the backtracking path. We formalise the re-

lation as the characteristic function of the set of nodes lying on the backtracking path in means of an auxiliary non-rigid predicate with explicit dependencies onPath[$*$.children[$*$]; $*$.nextChild] : $HeapObject \times HeapObject \times int$.

For sake of shortness and readability, we will skip the accessor list for the $onPath$ predicate from now on. Formally, the non-rigid predicate $onPath$ is defined as follows: Let σ denote a state, x, y terms of type HeapObject and n an integer term, then:

$$\sigma \models onPath(x, y, n)$$
$$\text{iff.}$$
$$n >= 0 \text{ and there exist terms } x = u_0, \ldots, u_n = y, \text{ such that}$$
$$\text{for all } 0 \le i < n :$$
$$\sigma \models u_{i+1} \doteq u_i.children[u_i.nextChild - 1]$$
$$and$$
$$\sigma \models !\, u_i \doteq null$$

Notice that $onPath(x, y, 0)$ is equivalent to $x \doteq y$. The semantical definition of $onPath$ is reflected by the calculus in form of a recursive definition:

——— KeY ———

```
onPathDefinition {
  \find(onPath(t1,t2,step))
  \replacewith(
    step >= 0 &
    ((t1 = t2 & step = 0) |
     (t1 != null & t1.nextChild@(HeapObject) > 0 &
      t1.nextChild@(HeapObject) <
            t1.children@(HeapObject).length &
      onPath(t1.children@(HeapObject)
                  [t1.nextChild@(HeapObject)-1], t2, step-1))
   ))
 };
```

——— KeY ———

The recursion is well founded and required to formalise the existential statement given in the semantical definition. For convenience reasons, we use additional taclets which can be derived directly from the rule onPathDefinition:

——— KeY ———

```
  onPathBase {
        \find ( onPath(t1, t2, 0) )
        \replacewith( t1 = t2 )
  };
  onPathNull {
        \find ( onPath(null, t2, n) )
        \replacewith( n = 0 & t2 = null )
  };
```

```
onPathNegative {
      \assumes (n < 0 ==> )
      \find ( onPath(t1, t2, n) )   \sameUpdateLevel
      \replacewith( false )
};
```
— KeY —

With this work done, we can now express the property that a node x is on the backtracking path by:

$$\exists\ int\ n\ onPath(previous, x, n);\ \ |\ \ \texttt{x} = \texttt{current}$$

where **previous** and **current** are the reference variables declared in method **mark**. Note, that we have included the **current** node to belong to the backtracking path.

15.2.2 Specification in Java Card DL

Pre- and Postconditions

The proof obligation of method **mark** to be proven valid is listed in Fig. 15.4. In previous chapters we considered proof obligations in OCL or JML. Since the reachable concepts are available in neither of them we resort to using Dynamic Logic formulas directly ($\Rightarrow$ Chap. 14). The proof obligation is composed of three components:

1. invariant of class **HeapObject** (lines 1–12),
2. the precondition proper (lines 14–19) and
3. the postcondition (lines 22–26) to be ensured to hold after the method has been executed.

The instance invariant of class **HeapObject** gives the following guarantees:

Line 3 that field **children** is always a non null array reference. Consequently, a node representing a sink refers to a zero-length array instead to **null**.
Lines 4–5 that the value of field **nextChild** ranges from 0 to the number of children.
Lines 7–9 that arrays referenced by **children** are not shared among different **HeapObject** instances.
Lines 10–12 that the components of the **children** array are not null.

In addition, a caller of method **mark** has to ensure that the start node, which is passed through as an argument (**startNode**), is not **null** (line 15) and that all markers of all nodes have been reset to their initial values indicating that they have not yet been visited (line 16). We simplified the specification

```
  ─── KeY ──────────────────────────────────────────────────
    // Invariant of class HeapObject
2     \forall HeapObject ho;(!(ho = null) ->
        !(ho.children = null)              &
4       ho.nextChild >= 0                  &
        ho.nextChild <= ho.children.length &
6       ho.children.length >= 0 ) &
      \forall HeapObject ho1;
8       \forall HeapObject ho2;(!(ho1 = ho2) ->
            !(ho1.children = ho2.children)) &
10    \forall HeapObject ho;\forall int i;
          (!(ho = null) & 0 <= i & i < ho.children.length ->
12           !(ho.children[i] = null))
    // contract for method 'mark'
14  // precondition
    !(startNode = null) &
16    \forall HeapObject ho; (ho.visited = FALSE & ho.nextChild = 0)
    // keep old values
18    \forall HeapObject ho; \forall int i;
        (children_pre(ho,i) = ho.children[i])
20  ->
    \[{ sw.mark(startNode); }\]
22  // postcondition
      (\forall HeapObject ho;(ho != null &
24        \exists int n; (n>=0 &
            reach[children[*];](startNode, ho, n)) <->
26          ho.visited = TRUE))
  ──────────────────────────────────────────────── KeY ───
```

Fig. 15.4. The JAVA CARD DL proof obligation for verifying Schorr-Waite

slightly by requiring that the markers of *all* nodes even of not yet created ones have been set to their initial value.

By proving the proof obligation, we can ensure that all and *only* nodes reachable from the starting node startNode have been marked as visited (line 22–26). Note, that the postcondition does not specify reservedness of the class invariants. This makes the proof easier and, in fact, one would often decompose these kind of proof obligations in order to keep a proof feasible.

For later use it will be convenient to refer to the "old" content of the children arrays. As JAVA CARD DL is not a high-level specification language, there is no construct like @pre in OCL or \old in JML. Instead we use in line 19 the trick to remember old values of the OCL/DL translation as described in Sect. 5.2.

Invariants

The most critical part of the specification is the loop invariant as most of the later verification depends on a sufficiently strong invariant. The **while** invariant rule used in KeY ($\Rightarrow$ Sect. 3.7.1) takes change information into consideration and allows to reduce the complexity of the invariant.

The loop's assignable set is

```
{current, previous, next, old,
 *.children@(HeapObject)[*],
 *.visited@(HeapObject),
 *.nextChild@(HeapObject) }
```

In the first line all possibly altered method-local pointers are enumerated. The remaining lines denote all the fields of nodes that are likely to be changed. The assignable set is a conservative approximation in principal it would be sufficient to restrict to fields of nodes reachable from the starting node.

The loop invariant is listed in Fig. 15.5. Its core part on which we will concentrate on is the subformula in lines 20–40. We come later back to the filtering condition stated in the lines 20–25. For the next few paragraphs, assume that the first condition (lines 21–22), and consequently, the following equations `lCur=current`, `lPrv=previous` and `bnd=current.nextChild` hold.

To write a good invariant means to find the right balance between being strong enough to allow to prove the methods postconditions, but not too strong in order to keep the *preserves loop invariant* proof branch as simple as possible.

For the moment let the graph to be marked be similar to the one shown in Fig. 15.6. In both sub-figures the algorithm is currently at node c and prepares for its move onwards to node n. The other children $d_0 \ldots d_k$ (with $k = c.\texttt{nextChild}-1$) have already been accessed via node c. The backtracking path is highlighted using solid curved edges.

In order to get a rough idea, how a possible invariant could look like, we concentrate first on the case illustrated by the left part of Figure 15.6(a).

The subgraph S spanned by the current node's children d_0 to d_k is a promising candidate to look at for a loop invariant. One is tempted to state that all nodes belonging to S have already been marked as visited and in fact, that is what we express in lines 27–31. The proof plan in mind is that if this invariant is preserved by the loop, then when the loop terminates we are back at the start node *and* all of its children have been accessed. Thus we can yield directly from the loop invariant that all nodes in the subgraph spanned by its children, which is nearly the complete graph excluding just the start node itself, have been marked.

But unfortunately the proposed invariant is not preserved by the loop, due to situations like the one illustrated in Fig. 15.6(b). In such a scenario the spanned subgraph contains a node u', which is only reachable via paths crossing the backtracking path, i.e., all paths share at least one node (here: u) with the backtracking path. In this case we cannot assume, that the complete

---
— KeY —

```
      current != null
    & current.visited = TRUE
    & (previous  = null -> startNode = current)
    & (previous != null -> previous.nextChild > 0)
 5  & \forall HeapObject ho;(
          (\exists int n; onPath(previous, ho, n)) ->
              (ho = null | ho.visited = TRUE))
    & \forall HeapObject ho;
          (ho != null ->
10             ho.nextChild <= ho.children.length & ho.nextChild >= 0)
    & \forall HeapObject ho;\forall int i;
          (ho != null & 0 <= i & i<ho.nextChild ->
              (ho.children[i] != null |
               (ho = startNode & current != startNode &
15                i = ho.nextChild - 1))) & ...
    & \forall HeapObject ho;
          \forall int i; ((ho != null & i >= ho.nextChild & i >= 0 &
              i < ho.children.length) ->
                ho.children[i] = children_pre(ho,i))
20  & \forall HeapObject lCur;\forall HeapObject lPrv;\forall int bnd;
      ((lCur != null & ( ( lCur = current & lPrv = previous
                          & bnd = current.nextChild)
                        | (lPrv = lCur.children[lCur.next-1]
                           & \exists int d; onPath(lPrv, lCur, d)
25                         & bnd = current.nextChild - 1)))
      ->
       \forall HeapObject ho1;(
         \forall int n; (ho1 != null & n >= 0 &
           \exists int idx; (0 <= idx & idx < bnd &
30           reach[children[*];](lCur.children[idx], ho1, n)))
         -> (ho1.visited = TRUE |
             (\exists HeapObject ho2; (ho2 != null &
               ho2.children != null
             & (\exists int d; (d >= 0 & onPath(lPrv, ho2, d))
35               | ho2 = lCur)
             & \exists int j; (ho2.nextChild <= j
                 & j < ho2.children.length
                 & \exists int l; (l >= 0
                     reach[children[*];](ho2.children[j], ho1, l))
40      ))))))

      Assignable Clause
      { current, previous, next, old, *.children@(HeapObject)[*],
        *.visited@(HeapObject), *.nextChild@(HeapObject) }
```
                                                              — KeY —

Fig. 15.5. Loop invariant and assignable clause

subgraph reachable from u has been visited. As the algorithm as described in 15.1.1 will stop at u and perform a backtracking step instead of exploring its unvisited children. Literally spoken, the backtracking path plays the role of demarcation line that bounds subgraph S.

To cope with these kinds of situations the stated property has to be weakened. This is achieved by introducing a disjunction (lines 33–40) stating now that all nodes of the spanned subgraph have been visited *or* there is at least one path to the node crossing the backtracking path. Notice, that this property is weaker than necessary as we do not require a node to be visited, if there is at least one path sharing a node with the backtracking path. This weakening does not hurt us, as in the use case the backtracking path is empty, turning the second part of the disjunction to false and allowing us to draw the conclusion that all nodes of the subgraph spanned by the children of the starting node have been marked as visited.

In order to reestablish the invariant in case of backward step, one has to state the explained property not only relative to the currently examined node (i.e., `lCur=current`), but for the other nodes on the backtracking path too. Therefore the second part of the filtering condition (lines 23– 25) is required. Some further (technical) invariant details:

lines 5–7 in addition to the nodes specified by the invariant's core part as marked, also the nodes of the backtracking path have been marked. This property is essential
- to solve some aliasing problems occurring during the proof like that the `current` node does not coincide with the former `previous` node
- to reason that the complete graph has been visited. Remember the core part allows only to draw the conclusion that the subgraph spanned by the children has been marked visited, but excludes the `current` node

lines 9–16 express a kind of "preserve instance invariant" statement for class `HeapObject`. Note that the loop will violate the invariant that **null** is not referenced by the children array components. The weakened version of this invariant can be found in lines 13-16.

lines 16–19 completes the former part of the invariant by stating that the components of the children with an index greater or equal to `nextChild` remain unchanged, allowing to use parts of the `HeapObject`'s invariant stated in the precondition (e.g., that the stored values are not **null**).

15.3 Verification of Schorr-Waite Within KeY

In the subsequent sections we will roughly outline the correctness proof of Schorr-Waite. We will step only into the technical details for some of the more interesting proof steps. The interested reader may download the complete proof from the book website and load it with the accompanying KeY version.

(a) Subgraph spanned by children 0 to $nextChild - 1$ do not contain a node only reachable by through nodes on the backtracking path

(b) Subgraph spanned by children 0 to $nextChild - 1$ contains a node where all paths have to cross the backtracking path
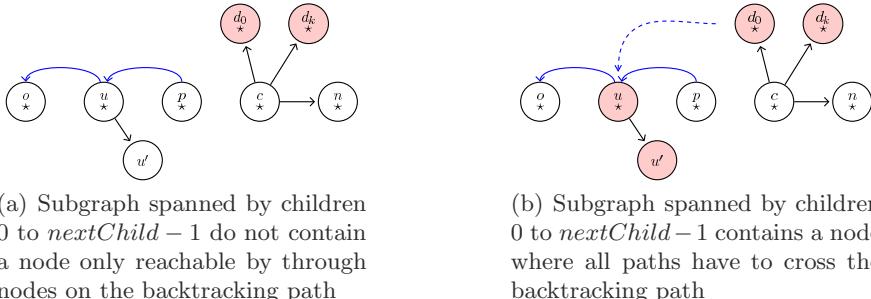
Fig. 15.6. Loop invariant: core part

15.3.1 Replacing Arguments of Non-rigid Functions Behind Updates

In several branches of the proof, we face situations similar to the following:

```
—— KeY ————————————————————————————
  {current:=startNode} reach[..](current, x, n)
==>
  {current:=startNode} reach[..](startNode, x, n)
                                              —— KeY ——
```

The sequent is clearly universally valid. But in order to close this proof goal, the first arguments of both occurrences of the reachable predicate need to be unified. Inserting the reachable definition will not succeed, as the definition itself is recursive and the value of `n` unknown. Furthermore, we have to operate behind updates, restricting the kind of applicable taclets.

In order to close this sequent the non-rigid arguments of the formulas `reach[..](fst, snd, thrd)` have to be replaced by new rigid constant symbols ci and defining equations `{current:=startNode}`ci `= fst` have to be added to the sequent's antecedent.

The replacement is performed by successive application of rule pullOut on the first arguments of both sides:

```
—— KeY ————————————————————————————
 pullOut { \find ( t ) \sameUpdateLevel
           \varcond ( \new(sk, \dependingOn(t)) )
           \replacewith (sk)
           \add ( t = sk ==>)
 };
                                              —— KeY ——
```

The result is the following sequent:

```
─── KeY ───────────────────────────────────────────
  {current:=startNode} (c1 = current),
  {current:=startNode} (c2 = startNode),
  {current:=startNode} reach[..](c1, x, n)
==>
  {current:=startNode} reach[..](c2, x, n)
───────────────────────────────────────── KeY ───
```

After a few further simplification steps, we obtain:

```
─── KeY ───────────────────────────────────────────
  c1 = startNode, c1 = c2,
  {current:=startNode} reach[..](c1, x, n))
==>
  {current:=startNode} reach[..](c2, x, n))
───────────────────────────────────────── KeY ───
```

The equation `c1 = c2` contains only rigid elements and is thus applicable also in the scope of updates - in fact behind any modality.

Applying this equation on the first argument of the third formula in the antecedent `{current:=startNode} reach[..](c1, x, n))` establishes an axiom where two equal formulas occur in the ante- and succedent.

15.3.2 The Proof

Invariant Initially Valid

This branch closes almost automatically (with help from Simplify for some universal quantifier instantiations). Only one interactive step remains for the invariant part, that ensures that all nodes on the backtracking path have been marked visited (lines 5-7). In the initial case, only the starting node, which has been marked visited in the statement before the loop is entered, is part of the backtracking path. To show that no other node is on the backtracking path, we insert the *onPath* predicate definition and leave the remaining steps for the strategies.

Use Case

As the method to verify ends when the loop terminates, this proof branch is of normal complexity. Most of the steps are performed automatically by the strategies. Nevertheless some interaction with the prover are necessary. Besides usual universal quantifications, which would be possible to perform also automatically (i.e., a heuristic approach should succeed), there is one step that will reoccur in the *preserves loop invariant*, which is of particular interest.

In this branch only normal termination of the loop is considered. Abrupt termination as uncaught exceptions or `return` or `break` statements are treated in the preserves invariant branch. The task is to prove that when the

- loop condition evaluates to `false` and
- loop invariant is valid

then

- the method's postcondition is satisfied, i.e., all reachable nodes have been visited.

The plan is to use the core part of the invariant (Fig. 15.5), lines 20–40).
    The postcondition to prove looked like

```
──── KeY ────
\forall HeapObject x; \forall int n;
  (x != null & reach[children[*];](startNode, x, n)
    -> x.visited = TRUE)
                                                    ──── KeY ────
```

In order to prove the post condition we have to show that an arbitrary chosen non-null instance `x_0` of type `HeapObject` reachable from the starting node `startNode` within `n_0` steps, is marked reachable.
    After some steps, this part of the proof goal is presented[1] as

```
──── KeY ────
( !(x_0 = null) & n_0 >= 0 &
   n_0 <= -1 + startNode.children.length &
{\for HeapObject h; h.nextChild := anonNextChild(h) ||
 startNode.visited              := TRUE             ||
 \for HeapObject h; h.visited   := anonVisited(h)   ||
 current                        := startNode        ||
 previous                       := anonPrevious(sw) ||
 \for (int i; HeapObject h)
   \if (i >= 0 & i <= -1 + h.children.length)
        h.children[i]           := anonChildren(h.children, i)}
 reach[children[*];](startNode, x_0, n_0)) ->
     anonVisited(x_0) = TRUE
                                                    ──── KeY ────
```

The first line corresponds with the afore stated side conditions. Following is a quantified update describing the state after leaving the methods. The functions symbols `anon*` are the anonymous functions introduced by the while invariant rule application. They describe the value of the location after the while loop, for example, `anonVisited(h)` is the value of `h.visited` when leaving the loop and so on.

---

[1] Names are slightly beautified.

Preserves Loop Invariant

Although this proof branch requires most interactions, the necessary techniques have been already introduced in the preceding paragraphs. The greatest difficulty is to keep track of the current loop invariant part that has to be proven.

In several subgoals the definitions of the *reachable* and *onPath* predicate have to be inserted—often in combination with the pullOut taclet to make the predicates' arguments rigid, as described in the *Use Case* paragraph. The remaining steps have been mostly simple quantifier instantiations.

15.4 Related Work

There is a variety of literature available about verification of the Schorr-Waite algorithm. We briefly describe a (representative) selection of them.

Broy and Pepper

The Schorr-Waite algorithm has been treated by [Broy and Pepper, 1982]. In this paper, the authors start with the construction of an algebraical data type modelling a binary graph. They continue with the definition of the reflexive and transitive closure relation R^* of the graph. Then a function B is developed, *proven* to compute the set of all reachable graph nodes from a distinguished node x, i.e. $R^*(x)$. The function B turns out to realise the well-known depth-first traversal algorithm for (binary) graphs.

An extended graph structure is build upon the binary graph data type. In addition to the binary graph it provides two distinguished nodes (representing the current and previous node). Also two additional basic functions *ex* and *rot* are defined, which exchange the current and previous node resp. perform a rotation operation (forward step). By composition of these elementary graph operations a function is constructed that computes and returns a tuple consisting of a set of nodes and an extended graph structure. It is proven that the returned node set is the same as computed by the former function B and that the returned extended graph structure is the same on which the function has operated.

Afterwards the functional algorithm is refined to a procedural version.

Mehta and Nipkow

The authors of [Mehta and Nipkow, 2003] verify the correctness of a Schorr-Waite implementation (for binary graphs) using higher order logics. The program is written in a simple imperative programming language designed by the authors themselves. The operational semantics of the programming language has been modelled in Isabelle/HOL and a Hoare style calculus has been derived from the semantics.

The main difference to our approach is the explicit modelling of heaps and the distinction between addresses and references. On top of these definitions a reachability relation (and some auxiliary relations) is defined as above.

The program is then specified using Hoare logic by annotating the program with assertions and a loop invariant making use of the former defined relations. From these annotations, verification conditions are generated, which have to be proven by Isabelle/HOL.

Abrial

The approach described in [Abrial, 2003] uses the B language and methodology to construct a correct implementation of the Schorr-Waite algorithm. Therefore the author starts with a high-level mathematical abstraction in B of a graph marking algorithm and then successively refines the abstraction towards an implementation of an (improved) version of Schorr-Waite. Each refinement step is accompanied by several proof obligations that need to be proven to ensure the correctness of the refinement step.

Yang

In [Yang, 2001] the author uses a relatively new kind of logic called Separation Logics, which is a variant of bunched implication logics. For verification they use a Hoare like calculus. The advantage of this logic is the possibility to express that two heaps are distinct and in particular the existence/possibility of a frame introduction rule. In short, the frame introduction rule allows to embed a property shown for a local memory area in a global context with other memory cells.

The frame rule allows to show that if $\{P\}C\{Q\}$ is valid for a local piece of code C then one can embed this knowledge in a broader context $\{P * H\}C\{Q * H\}$ as long as the part of the heap H talks about is not altered by C (separate heaps). Without this frame rule one would have to consider H when proving $\{P\}C\{Q\}$, which makes correctness proves very tedious, in particular when the property shall be used in different *separate* contexts H_i.

Hubert and Marché

In [Hubert and Marché, 2005] the authors follow an approach very similar to the one presented in this chapter. They used a weakest precondition calculus for C implemented in the CADUCEUS tool to verify a C implementation of Schorr-Waite working on a bigraph. In the same manner as described here, they specified the loop invariant with help of an inductively defined reachable predicate using a higher order logic.

Appendices

A

Predefined Operators in Java Card DL

by
Steffen Schlager

This appendix lists syntax and semantics of all predefined function and predicate symbols of Java Card DL.

A.1 Syntax

A.1.1 Built-in Rigid Function Symbols

These symbols are contained in the set FSym_r^0 ($\Rightarrow$ Def. 3.4).

function symbol	typing and informal semantics
$+$	`integer`, `integer` $\rightarrow$ `integer` addition
$-$	`integer`, `integer` $\rightarrow$ `integer` subtraction
$*$	`integer`, `integer` $\rightarrow$ `integer` multiplication
$/$	`integer`, `integer` $\rightarrow$ `integer` Euclidian division
$\%$	`integer`, `integer` $\rightarrow$ `integer` remainder for $/$
$-$	`integer` $\rightarrow$ `integer` unary minus
jdiv	`integer`, `integer` $\rightarrow$ `integer` division rounding towards 0
jmod	`integer`, `integer` $\rightarrow$ `integer` remainder for *jdiv*
$\ldots, -1, 0, 1, 2, \ldots$	`integer` integer numbers
FALSE	`boolean` constant for truth value false

function symbol	typing and informal semantics
TRUE	boolean
	constant for truth value true
null	Null
	constant for null element
(A) (for any $A \in \mathcal{T}$)	$\top \rightarrow A$
	cast to type A
A::get for any $A \in \mathcal{T}_d \setminus \{\text{Null}\}$	integer
	gives an object of dynamic type A
max_byte	integer
max_short	integer
max_int	integer
max_long	integer
max_char	integer
	maximum number of
	respective Java Card type
min_byte	integer
min_short	integer
min_int	integer
min_long	integer
min_char	integer
	minimum number of
	respective Java Card type

A.1.2 Built-in Rigid Function Symbols whose Semantics Depends on the Chosen Integer Semantics

These symbols are contained in the set FSym_r^0 ($\Rightarrow$ Def. 3.4). Their semantics depends on the chosen integer semantics ($\Rightarrow$ Sect. 12.5).

function symbol	typing and informal semantics
unaryMinusJint	integer
unaryMinusJlong	integer
	unary minus
addJint	integer, integer $\rightarrow$ integer
addJlong	integer, integer $\rightarrow$ integer
	addition
subJint	integer, integer $\rightarrow$ integer
subJlong	integer, integer $\rightarrow$ integer
	subtraction
mulJint	integer, integer $\rightarrow$ integer
mulJlong	integer, integer $\rightarrow$ integer
	multiplication

function symbol	typing and informal semantics
divJint	integer, integer → integer
divJlong	integer, integer → integer
	division
modJint	integer, integer → integer
modJlong	integer, integer → integer
	modulo
shiftrightJint	integer, integer → integer
shiftrightJlong	integer, integer → integer
	binary shift-right
unsignedshiftrightJint	integer, integer → integer
unsignedshiftrightJlong	integer, integer → integer
	unsigned binary shift-right
shiftleftJint	integer, integer → integer
shiftleftJlong	integer, integer → integer
	binary shift-left
orJint	integer, integer → integer
orJlong	integer, integer → integer
	binary or
xorJint	integer, integer → integer
xorJlong	integer, integer → integer
	binary xor
andJint	integer, integer → integer
andJlong	integer, integer → integer
	binary and
negJint	integer
negJlong	integer
	binary negation
moduloByte	integer
moduloShort	integer
moduloInt	integer
moduloLong	integer
moduloChar	integer
	computation of overflow

A.1.3 Built-in Non-Rigid Function Symbols

These symbols are contained in the set FSym^0_{nr} ($\Rightarrow$ Def. 3.4).

function symbol	typing and informal semantics
[ ]	⊤, integer → ⊤
	array access in the logic
length	⊤ → integer
	length of an array

function symbol	typing and informal semantics
`A.<nextToCreate>` (for any $A \in \mathcal{T}_d$)	`integer` index of next object to be created
`<created>`	Object → boolean indicates whether object is created

A.1.4 Built-in Rigid Predicate Symbols

These symbols are contained in the set PSym_r^0 ($\Rightarrow$ Def. 3.4).

predicate symbol	typing and informal semantics
$<$	`integer, integer` less than
$\leq$	`integer, integer` less than or equal
$>$	`integer, integer` greater than
$\geq$	`integer, integer` greater than or equal
$\doteq$	$\top, \top$ equality
$!\doteq$	$\top, \top$ inequality
$quanUpdateLeq$	$\top, \top$ ordering predicate
$\sqsubseteq A$ (for any $A \in \mathcal{T}$)	$\top$ type predicate for A
$arrayStoreValid$	$\top, \top$ holds iff an array store operation is valid for the given arguments

A.1.5 Built-in Rigid Predicate Symbols whose Semantics Depends on the Chosen Integer Semantics

These symbols are contained in the set PSym_r^0 ($\Rightarrow$ Def. 3.4). Their semantics depends on the chosen integer semantics ($\Rightarrow$ Sect. 12.5).

predicate symbol	typing and informal semantics
inByte	`integer`
inShort	`integer`
inInt	`integer`
inLong	`integer`
inChar	`integer`
	holds iff argument in range
	of respective JAVA CARD type

A.1.6 Built-in Non-rigid Predicate Symbols Contained in PSym^0_{nr}

This symbol is contained in the set PSym^0_{nr} ($\Rightarrow$ Def. 3.4).

predicate symbol	typing and informal semantics
inReachableState	()
	characterises JAVA CARD-reachable states

A.2 Semantics

A.2.1 Semantics of Built-in Rigid Function Symbols

function symbol	semantics
$+$	$\mathcal{I}_0(+)(x, y) = x + y$
$-$	$\mathcal{I}_0(-)(x, y) = x - y$
$*$	$\mathcal{I}_0(*)(x, y) = x * y$
$/$	$\mathcal{I}_0(/)(x, y) = \begin{cases} z \text{ such that} & \text{if } y \neq 0 \\ 0 \leq x - y * z < |y| & \\ \text{some arbitrary but} & \text{otherwise} \\ \text{fixed } d \in \mathcal{D}^{\texttt{integer}} \end{cases}$
$\%$	$\mathcal{I}_0(\%)(x, y) = \begin{cases} x \% y := x - y * (x/y) & \text{if } y \neq 0 \\ \text{some arbitrary but} & \text{otherwise} \\ \text{fixed } d \in \mathcal{D}^{\texttt{integer}} \end{cases}$
$-$	$\mathcal{I}_0(-)(x) = -x$

function symbol	semantics
jdiv	$\mathcal{I}_0(jdiv)(x,y) =$ $$\begin{cases} \lvert x\rvert/\lvert y\rvert & \text{if } x \geq 0, y > 0 \text{ or} \\ & x \leq 0, y < 0 \\ -\lvert x\rvert/\lvert y\rvert & \text{if } x < 0, y > 0 \text{ or} \\ & x > 0, y < 0 \\ \text{some arbitrary but} & \text{if } y = 0 \\ \text{fixed } d \in \mathcal{D}^{\texttt{integer}} \end{cases}$$
jmod	$\mathcal{I}_0(jmod)(x,y) =$ $$\begin{cases} x - y * \mathcal{I}_0(jdiv(x,y)) & \text{if } y \neq 0 \\ \text{some arbitrary but} & \text{otherwise} \\ \text{fixed } d \in \mathcal{D}^{\texttt{integer}} \end{cases}$$
$\ldots, -1, 0, 1, \ldots$	$\mathcal{I}_0(i) = i$ for $i \in \{\ldots, -1, 0, 1, \ldots\}$
FALSE	$\mathcal{I}_0(\text{FALSE}) = \mathit{ff}$
TRUE	$\mathcal{I}_0(\text{TRUE}) = \mathit{tt}$
null	$\mathcal{I}_0(null) = null$
(A)	$\mathcal{I}_0((A))(x) =$ $$\begin{cases} x & \text{if } \delta_0(x) \sqsubseteq A \\ \text{some arbitrary} & \text{otherwise} \\ \text{but fixed } d \in \mathcal{D}^A \end{cases}$$
A::get	see Def. 3.53
max_byte	$\mathcal{I}_0(max\_byte)(x) = 2^7 - 1$
min_byte	$\mathcal{I}_0(min\_byte)(x) = -2^7$
max_short	$\mathcal{I}_0(max\_short)(x) = 2^{15} - 1$
min_short	$\mathcal{I}_0(min\_short)(x) = -2^{15}$
max_int	$\mathcal{I}_0(max\_int)(x) = 2^{31} - 1$
min_int	$\mathcal{I}_0(min\_int)(x) = -2^{31}$
max_long	$\mathcal{I}_0(max\_long)(x) = 2^{63} - 1$
min_long	$\mathcal{I}_0(min\_long)(x) = -2^{63}$
max_char	$\mathcal{I}_0(max\_char)(x) = 2^{16} - 1$
min_char	$\mathcal{I}_0(min\_char)(x) = 0$
moduloByte	$\mathcal{I}_0(moduloByte)(x)$ $= (x + 2^7)\%2^8 - 2^7$
moduloShort	$\mathcal{I}_0(moduloShort)(x)$ $= (x + 2^{15})\%2^{16} - 2^{15}$
moduloInt	$\mathcal{I}_0(moduloInt)(x)$ $= (x + 2^{31})\%2^{32} - 2^{31}$
moduloLong	$\mathcal{I}_0(moduloLong)(x)$ $= (x + 2^{63})\%2^{64} - 2^{63}$
moduloChar	$\mathcal{I}_0(moduloChar)(x) = x\%2^{16}$

function symbol	semantics
unaryMinusJint	$\mathcal{I}_0(unaryMinusJint)(x) = roundInt(-x)$
unaryMinusJlong	$\mathcal{I}_0(unaryMinusJlong)(x) = roundLong(-x)$
addJint	$\mathcal{I}_0(addJint)(x, y) = roundInt(x + y)$
addJlong	$\mathcal{I}_0(addJlong)(x, y) = roundLong(x + y)$
subJint	$\mathcal{I}_0(subJint)(x, y) = roundInt(x - y)$
subJlong	$\mathcal{I}_0(subJlong)(x, y) = roundLong(x - y)$
mulJint	$\mathcal{I}_0(mulJint)(x, y) = roundInt(x * y)$
mulJlong	$\mathcal{I}_0(mulJlong)(x, y) = roundLong(x * y)$
divJint	$\mathcal{I}_0(divJint)(x, y) = roundInt(jdiv(x, y))$
divJlong	$\mathcal{I}_0(divJlong)(x, y) = roundLong(jdiv(x, y))$
modJint	$\mathcal{I}_0(modJint)(x, y) = roundInt(jmod(x, y))$
modJlong	$\mathcal{I}_0(modJlong)(x, y) = roundLong(jmod(x, y))$
	where $roundInt = \mathcal{I}_0(moduloInt)$ and
	$roundLong = \mathcal{I}_0(moduloLong)$
shiftrightJint	$\mathcal{I}_0(shiftrightJint)(x, y) = x \verb|>>| y$
shiftrightJlong	$\mathcal{I}_0(shiftrightJlong)(x, y) = x \verb|>>| y$
unsignedshiftrightJint	$\mathcal{I}_0(unsignedshiftrightJint)(x, y) = x \verb|>>>| y$
unsignedshiftrightJlong	$\mathcal{I}_0(unsignedshiftrightJlong)(x, y) = x \verb|>>>| y$
shiftleftJint	$\mathcal{I}_0(shiftleftJint)(x, y) = x \verb|<<| y$
shiftleftJlong	$\mathcal{I}_0(shiftleftJlong)(x, y) = x \verb|<<| y$
orJint	$\mathcal{I}_0(orJint)(x, y) = x \mid y$
orJlong	$\mathcal{I}_0(orJlong)(x, y) = x \mid y$
xorJint	$\mathcal{I}_0(xorJint)(x, y) = x \verb|^| y$
xorJlong	$\mathcal{I}_0(xorJlong)(x, y) = x \verb|^| y$
andJint	$\mathcal{I}_0(andJint)(x, y) = x \verb|&| y$
andJlong	$\mathcal{I}_0(andJlong)(x, y) = x \verb|&| y$
negJint	$\mathcal{I}_0(negJint)(x) = \verb|~|x$
negJlong	$\mathcal{I}_0(negJlong)(x) = \verb|~|x$
	these functions have the same semantics as the corresponding JAVA bit-operators [Gosling et al., 2000, § 15.15.5]

A.2.2 Semantics of Built-in Predicate Symbols

function symbol	semantics
inByte	$\mathcal{I}_0(inByte) = \{z \in \mathbb{Z} \mid -2^7 \le z \le 2^7 - 1\}$
inShort	$\mathcal{I}_0(inShort) = \{z \in \mathbb{Z} \mid -2^{15} \le z \le 2^{15} - 1\}$
inInt	$\mathcal{I}_0(inInt) = \{z \in \mathbb{Z} \mid -2^{31} \le z \le 2^{31} - 1\}$
inLong	$\mathcal{I}_0(inLong) = \{z \in \mathbb{Z} \mid -2^{63} \le z \le 2^{63} - 1\}$
inChar	$\mathcal{I}_0(inChar) = \{z \in \mathbb{Z} \mid 0 \le z \le 2^{16} - 1\}$
$<$	$\mathcal{I}_0(<) = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid x < y\}$
$<=$	$\mathcal{I}_0(\le) = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid x \le y\}$
$>$	$\mathcal{I}_0(>) = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid x > y\}$

function symbol	semantics
$>=$	$\mathcal{I}_0(\geq) = \{(x,y) \in \mathbb{Z} \times \mathbb{Z} \mid x \geq y\}$
$\doteq$	$\mathcal{I}_0(\doteq) = \{(x,y) \in \mathcal{D} \times \mathcal{D} \mid x = y\}$
$quanUpdateLeq$	$\mathcal{I}_0(quanUpdateLeq) = \{(x,y) \in \top \times \top \mid x \prec y\}$
$\sqsubseteq A$	$\mathcal{I}_0(\sqsubseteq A)(x) = \mathcal{D}^A$
$arrayStoreValid$	$\mathcal{I}_0(arrayStoreValid) =$ $\quad \{(x,y) \in \mathcal{D} \times \mathcal{D} \mid A \sqsubseteq B \text{ where}$ $\quad\quad\quad\quad B[\,] = \delta_0(x) \text{ and } A = \delta_0(y)\}$
$inReachableState$	holds in exactly those states that are reachable by a JAVA CARD program

B

The KeY Syntax

by
Wojciech Mostowski

The KeY system accepts different kinds of inputs related to Java Card DL. From the user point of view these inputs can be divided into the following categories:

- system rule files,
- user defined rule files,
- user problem files/proofs with optional user defined rules,
- Java Card DL terms and formulae required by the interaction component of the KeY system.

From the system's perspective the division is similar, but on top of this, the distinction between schematic mode and term (normal) mode is very important:

- in schematic mode schema variables can be defined and used (usually in definition of rules/taclets) and concrete terms or formulae are forbidden,
- in normal mode schema variables and all other schematic constructs are forbidden, while concrete terms and formulae are allowed.

Additionally, most of terms and formulae constructs can appear in both schematic and normal mode, but take slightly different form depending on the mode.

In either case, all inputs the KeY system accepts follow the same syntax—the KeY syntax (or, as sometimes it is sometimes referred to, `.key` file syntax).

On the implementation level, the parsing of the KeY input is done on two levels. One parser (called term, taclet, or problem parser) is used to parse all the input up to modalities, and a second parser (schematic Java parser) is used to parse all Java program blocks inside the modalities. Thus, on occasion, slightly different conventions may apply when input material inside the modality is considered as compared to input outside of the modality.

Finally, note that the syntax described here reflects the syntax of the KeY system snapshot available before the book was printed. The KeY syn-

tax undergoes minor changes during system development, thus the publicly available KeY system may differ slightly in its input syntax.

B.1 Notation, Keywords, Identifiers, Numbers, Strings

Expressions in the `type-writer` font are KeY syntax tokens or identifiers. Keywords are annotated with **bold type-writer** font. In the KeY system the convention is to use an escape character, backslash \, to mark (almost) all keywords, and some KeY specific operators, like modalities. This is necessary to avoid collisions between KeY system keywords/operators and possible JAVA identifiers/operators.

An identifier in the KeY system can be one of the following:

───── KeY Syntax ───

`lettersdigits_#`	starts with a letter, can contain letters, digits, underscore, and hash characters
`identifier@pre`	like above, to allow OCL names in JAVA CARD DL
`$identifier`	like the first one, for special OCL purposes, an identifier can start with a single dollar character
`<letters>`	identifier enclosed in <>, used to annotate implicit attributes, only letters allowed
`\letters_`	If not a reserved keyword, a sequence of letters and underscores starting with a backslash is also an identifier
`singledigit`	In special cases, when used as a function symbol, e.g., `1(...)`, a single digit is also an identifier

── KeY Syntax ─────

A keyword is a reserved identifier that starts with a backslash \ and contains only letters and underscores. An exception from this rule are keywords **true**, **false**, and modality symbols. Some examples of identifiers and the list of all keywords:

───── KeY Syntax ───

Identifiers:
```
varName #varName operation@pre $forAll
<transient> \non_keyword_ident
```

Keywords and keyword-escaped symbols:
   \sorts \generic \extends \oneof \object \inter

   **\schemaVariables \schemaVar \modalOperator \operator
   \program \formula \term \variables \skolemTerm \location
   \function**

   \programVariables

```
\varcond \typeof \elemTypeof \new \not \same \compatible
\sub \strict \staticMethodReference \notFreeIn
\static \notSameLiteral \isReferenceArray \isReference
\dependingOn \dependingOnMod \isQuery \hasSort
\isLocalVariable \isInReachableState \isAbstractOrInterface
\containerType

\bind \forall \exists \subst \ifEx \for
\if \then \else

\include \includeLDTs \javaSource \withOptions
\optionsDecl \settings

\sameUpdateLevel \inSequentState \closegoal
\heuristicsDecl \noninteractive \displayname
\helptext \replacewith \addrules \addprogvars
\heuristics \recursive \find \add \assumes
\inType

\predicates \functions \nonRigid

\rules \problem \proof \contracts \modifies

\< \> \[ \] \[[ \]] \diamond \box \throughout
\modality \endmodality
\diamond_trc \diamond_tra \diamond_susp
\box_trc \box_tra \box_susp
\throughout_trc \throughout_tra \throughout_susp
true false
```
———————————————————————————————— KeY Syntax ——

An integer number in KeY can be given in a decimal or hexadecimal form
with infinite precision. An integer constant can have a negation sign:

—— KeY Syntax ————————————————————————————————————
Decimal integers:
```
    1 2 -3 10 -20 12345678901234567890123456789
```
Hexadecimal integers:
```
    0x01 -0xA 0xFFAABBCC0090ffaa
```
———————————————————————————————— KeY Syntax ——

The KeY system can also recognise strings and character constants in its in-
put. Strings and characters in KeY are practically the same as strings and
characters in JAVA, with the same special characters and character quot-
ing rules:

───── KeY Syntax ─────────────────────────────────────

```
"A␣string␣with␣a␣line␣break␣at␣the␣end.\n"
'A' '0' '\t' '\r' '0x0020'
```

───────────────────────────────── KeY Syntax ─────

Finally, in the following expressions in ⟨*italics*⟩ represent parsing rules in regular expression form with operators ::= (definition), | (alternative), ? (zero or one occurrence), ∗ (zero or more occurrences), + (one or more occurrences), and () grouping. Whenever necessary, explanations are given to explain intuitive meaning of the rules. Identifiers are denoted with ⟨*identifier*⟩, numbers with ⟨*number*⟩, strings and characters with ⟨*string*⟩ and ⟨*character*⟩ respectively.

B.2 Terms and Formulae

We start with describing what are the KeY syntax rules to construct a valid JAVA CARD DL term or formula. Note, that on the syntax level, terms are hardly distinguishable from formulae, that is, from the parser point of view, a formula is a term with a special top-level sort (a "formula" sort). In general, many of the following rules for the KeY syntax are only applicable if the involved expressions have the right sort. That is, on the implementation level, apart from the syntax checks also semantic checks are performed when expressions are parsed.

B.2.1 Logic Operators

The logic operators for building terms or formulae are the following:

───── KeY Syntax ─────────────────────────────────────

⟨*formula*⟩ ::= ⟨*term*⟩

⟨*term*⟩ ::= ⟨*term₁*⟩ (**<->** ⟨*term₁*⟩)∗

⟨*term₁*⟩ ::= ⟨*term₂*⟩ (**->** ⟨*term₁*⟩)?

⟨*term₂*⟩ ::= ⟨*term₃*⟩ (**|** ⟨*term₃*⟩)∗

⟨*term₃*⟩ ::= ⟨*term₄*⟩ (**&** ⟨*term₄*⟩)∗

⟨*term₄*⟩ ::=
      ! ⟨*term₄*⟩
   | ⟨*modality Term*⟩
   | ⟨*quantifier Term*⟩
   | ⟨*equality Term*⟩

───────────────────────────────── KeY Syntax ─────

Intuitively, it means that the negation ! is the strongest operator and is right associative, then comes left associative conjunction &, then left associative disjunction |, then *right* associative implication ->, and finally left associative equivalence <->. Possible modalities are the following:

───── KeY Syntax ─────────────────────────────────────

$\langle modalityTerm\rangle$ ::=
    $\langle modalityBlock\rangle$ $\langle term_4\rangle$

$\langle modalityBlock\rangle$ ::=
    \< $\langle javaBlock\rangle$ **\>**
  | **\[** $\langle javaBlock\rangle$ **\]**
  | **\[[** $\langle javaBlock\rangle$ **\]]**
  | **\diamond** $\langle javaBlock\rangle$ **\endmodality**
  | **\box** $\langle javaBlock\rangle$ **\endmodality**
  | **\throughout** $\langle javaBlock\rangle$ **\endmodality**
  | **\diamond_trc** $\langle javaBlock\rangle$ **\endmodality**
  | **\box_trc** $\langle javaBlock\rangle$ **\endmodality**
  | **\throughout_trc** $\langle javaBlock\rangle$ **\endmodality**
  | **\diamond_tra** $\langle javaBlock\rangle$ **\endmodality**
  | **\box_tra** $\langle javaBlock\rangle$ **\endmodality**
  | **\throughout_tra** $\langle javaBlock\rangle$ **\endmodality**
  | **\diamond_susp** $\langle javaBlock\rangle$ **\endmodality**
  | **\box_susp** $\langle javaBlock\rangle$ **\endmodality**
  | **\throughout_susp** $\langle javaBlock\rangle$ **\endmodality**
  | **\modality{**$\langle modalityName\rangle$**}** $\langle javaBlock\rangle$ **\endmodality**

$\langle modalityName\rangle$ ::= $\langle identifier\rangle$

─────────────────────────────────────── KeY Syntax ─────

In the last alternative, $\langle modalityName\rangle$ can be either a concrete modality (**diamond**, **box**, **diamond_trc**, etc.), or a schema variable representing a set of modalities if the expression is parsed in the schematic mode.

    As mentioned earlier, JAVA blocks inside modalities are parsed separately, we describe the corresponding syntax in Section B.5.

    Next, a quantifier takes the following form:

───── KeY Syntax ─────────────────────────────────────

$\langle quantifierTerm\rangle$ ::=
    \forall $\langle variableBinding\rangle$ $\langle term_4\rangle$
  | **\exists** $\langle variableBinding\rangle$ $\langle term_4\rangle$

─────────────────────────────────────── KeY Syntax ─────

A variable binding takes the following form:

——— KeY Syntax ———

⟨*variableBinding*⟩ ::=
    ⟨*singleVariableBinding*⟩
  | ⟨*multipleVariableBinding*⟩

——————————————————————————————————— KeY Syntax ———

Then, depending on the parsing mode, variable binding can take the following forms, in normal mode:

——— KeY Syntax ———

⟨*singleVariableBinding*⟩ ::= ⟨*sortExp*⟩ ⟨*varName*⟩ ;

⟨*multipleVariableBinding*⟩ ::=
  (⟨*sortExp*⟩ ⟨*varName*⟩ (; ⟨*sortExp*⟩ ⟨*varName*⟩)+)

⟨*varName*⟩ ::= ⟨*identifier*⟩

——————————————————————————————————— KeY Syntax ———

And in the schematic mode:

——— KeY Syntax ———

⟨*singleVariableBinding*⟩ ::=
  ⟨*schemaVarName*⟩ ;

⟨*multipleVariableBinding*⟩ ::=
  (⟨*schemaVarName*⟩ (; ⟨*schemaVarName*⟩)+)

⟨*schemaVarName*⟩ ::= ⟨*identifier*⟩

——————————————————————————————————— KeY Syntax ———

In the former, ⟨*varName*⟩ is any valid KeY identifier and ⟨*sortExp*⟩ is a valid sort name as explained shortly, in the latter ⟨*schemaVarName*⟩ is also any valid KeY identifier associated with a proper schema variable (⇒ Sect. 4.1). A sort expression takes the following form:

——— KeY Syntax ———

⟨*sortExp*⟩ ::=
    ⟨*sortName*⟩ (([]) ∗ | ({})∗)
  | ⟨*intersectionSort*⟩

⟨*intersectionSort*⟩ ::=
    \inter(⟨*sortName*⟩ ,
      (⟨*intersectionSort*⟩ | ⟨*sortName*⟩))

——————————————————————————————————— KeY Syntax ———

A sort name $\langle sortName \rangle$ is any valid KeY sort, including fully qualified sorts that reflect JAVA types. The suffix [] denotes "array of" sort, and the suffix {} denotes "set of" sort. The keyword **\inter** is used to construct intersection sorts. Examples of valid quantifiers and sort expressions are the following:

—— KeY Syntax ——————————————

```
\forall (int i; int j) true
\exists java.lang.Object{} o_set; true
\inter(Sort1, \inter(Sort2, Sort3))
```

—————————————————————— KeY Syntax ——

In the remainder of this appendix $\langle variableBinding \rangle$ and $\langle sortExp \rangle$ are going to be referenced often.

Finally, an $\langle equalityTerm \rangle$ expresses (in-)equality between two atomic terms:

—— KeY Syntax ——————————————

$\langle equalityTerm \rangle$ ::=
    $\langle atomicTerm_1 \rangle$ (= $\langle atomicTerm_1 \rangle$)?
  | $\langle atomicTerm_1 \rangle$ (!= $\langle atomicTerm_1 \rangle$)?

—————————————————————— KeY Syntax ——

The inequality operator != is simply a syntactic sugar: a != b is the same as !a = b (note that = binds stronger than !).

B.2.2 Atomic Terms

Atomic terms are build in the following way. The top-level atomic term is:

—— KeY Syntax ——————————————

$\langle atomicTerm_1 \rangle$ ::=
  $\langle atomicTerm_2 \rangle$ ($\langle intRelation \rangle$ $\langle atomicTerm_2 \rangle$)?

$\langle intRelation \rangle$ ::=
  < | <= | > | >=

—————————————————————— KeY Syntax ——

The rule $\langle intRelation \rangle$ represents a possible integer comparison relation in the infix form. Of course, such relation can be only used if the sort of $\langle atomicTerm_2 \rangle$ permits this. The infix relation expressions (as well as infix integer binary operators, like +, -, *, etc., see below) are only a short hand notation for corresponding function symbols, like lt, geq, add, or mul ($\Rightarrow$ App. A).

Further definitions for atomic terms are the following:

───── KeY Syntax ───

$\langle atomicTerm_2 \rangle$ ::=
  $\langle atomicTerm_3 \rangle$ ($\langle arithOp_1 \rangle$ $\langle atomicTerm_3 \rangle$)∗

$\langle atomicTerm_3 \rangle$ ::=
  $\langle atomicTerm_4 \rangle$ ($\langle arithOp_2 \rangle$ $\langle atomicTerm_4 \rangle$)∗

$\langle atomicTerm_4 \rangle$ ::=
    - $\langle atomicTerm_4 \rangle$
  | ($\langle sortExp \rangle$) $\langle atomicTerm_4 \rangle$
  | $\langle atomicTerm_5 \rangle$

$\langle arithOp_1 \rangle$ ::= + | -

$\langle arithOp_2 \rangle$ ::= * | / | %

─── KeY Syntax ─────

Intuitively, all binary arithmetic operators are left associative, and *, /, % bind stronger than + and -. Unary minus and sort casts (definition of $\langle atomicTerm_4 \rangle$) are strongest and right associative. Next, the definition for $\langle atomicTerm_5 \rangle$ is the following:

───── KeY Syntax ───

$\langle atomicTerm_5 \rangle$ ::=
    $\langle accessTerm \rangle$
  | $\langle substitutionTerm \rangle$
  | $\langle updateTerm \rangle$

─── KeY Syntax ─────

Access terms are defined in the following way:

───── KeY Syntax ───

$\langle accessTerm \rangle$ ::=
  $\langle primitiveTerm \rangle$ $\langle arrayAttributeQueryAccess \rangle$∗

$\langle primitiveTerm \rangle$ ::=
    $\langle staticQuery \rangle$
  | $\langle staticAttribute \rangle$
  | $\langle functionPredicateTerm \rangle$
  | $\langle variable \rangle$
  | $\langle conditionalTerm \rangle$
  | $\langle specialTerm \rangle$
  | $\langle abbrTerm \rangle$
  | ($\langle term \rangle$)

```
    |   true
    |   false
    |   ⟨number⟩
    |   ⟨character⟩
    |   ⟨string⟩
```

⟨*arrayAttributeQueryAccess*⟩ ::=
      ⟨*arrayAccess*⟩
  | ⟨*attributeAccess*⟩
  | ⟨*queryAccess*⟩

⟨*arrayAccess*⟩ ::=
  [⟨*indexTerm*⟩] ⟨*shadowOp*⟩?

⟨*indexTerm*⟩ ::=
      ⟨*atomicTerm₁*⟩
  | *
  | ⟨*atomicTerm₁*⟩ .. ⟨*atomicTerm₁*⟩

⟨*attributeAccess*⟩ ::= . ⟨*attributeExp*⟩ ⟨*shadowOp*⟩?

⟨*queryAccess*⟩ ::= . ⟨*queryExp*⟩

⟨*staticAttribute*⟩ ::=
  ⟨*typeReference*⟩ . ⟨*attributeExp*⟩ ⟨*shadowOp*⟩?

⟨*staticQuery*⟩ ::= ⟨*typeReference*⟩ . ⟨*queryExp*⟩

⟨*shadowOp*⟩ ::=
      ^(⟨*atomicTerm₁*⟩)
  | '+

———————————————————————————————————— KeY Syntax ——

The ⟨*shadowOp*⟩ rule gives the syntax for *shadowed* array and attribute expressions (⇒ Sect. 9.5.2). The second alternative can only be used in normal term parsing mode, and not in the schematic mode. The last two alternatives of the ⟨*indexTerm*⟩ can only occur when modifies clauses (for example, in contracts (⇒ Sect. B.4.1), or when instantiating modifier sets (⇒ Sect. 3.7.4)) are parsed and denote quantified array expressions. Further details are the following:

—— KeY Syntax ——————————————————————————————

⟨*attributeExp*⟩ ::=
  ⟨*attributeName*⟩ ⟨*classLocator*⟩?

⟨*queryExp*⟩ ::=
  ⟨*queryName*⟩ ⟨*classLocator*⟩? (⟨*argumentList*⟩?)

⟨*classLocator*⟩ ::= @(⟨*typeReference*⟩)

⟨*argumentList*⟩ ::= (⟨*term*⟩ (, ⟨*term*⟩)*)

⟨*attributeName*⟩ ::= ⟨*identifier*⟩

⟨*queryName*⟩ ::= ⟨*identifier*⟩

——————————————————————————————— KeY Syntax ——

Class locator expressions are used to resolve possible collisions between attribute names when JAVA name shadowing occurs. Class locator expressions are obligatory when such a collision takes place, otherwise they are optional. Class locators can only occur in normal term parsing mode, not in the schematic mode. A ⟨*typeReference*⟩ is a fully qualified JAVA type expression, for example:

—— KeY Syntax ———————————————————————————————
```
java.lang.Object
int[]
```
——————————————————————————————— KeY Syntax ——

If there are no ambiguities, the package information can be skipped. An ⟨*attributeName*⟩ is either a concrete attribute name, or a schema variable representing one, again depending on the parsing mode. ⟨*queryName*⟩ is similar to ⟨*attributeName*⟩, however in the current version of the KeY system, query expressions can only appear in normal parsing mode, thus ⟨*queryName*⟩ always represents a concrete method/query name.

    Before we describe what are the lowest level building blocks for terms, we first go back to the definition of ⟨*substitutionTerm*⟩ and ⟨*updateTerm*⟩:

—— KeY Syntax ———————————————————————————————
⟨*substitutionTerm*⟩ ::=
  { **\subst** ⟨*singleVariableBinding*⟩ ⟨*atomicTerm₁*⟩ } ⟨*term₄*⟩

⟨*updateTerm*⟩ ::=
  { ⟨*singleUpdate*⟩ (|| ⟨*singleUpdate*⟩)* } ⟨*term₄*⟩

⟨*singleUpdate*⟩ ::=
    (**\for** ⟨*variableBinding*⟩)? (**\if** (⟨*formula*⟩))?
    ⟨*atomicTerm₁*⟩ := ⟨*atomicTerm₁*⟩
  | * := * ⟨*number*⟩
——————————————————————————————— KeY Syntax ——

The second alternative in the ⟨*singleUpdate*⟩ rule represents an anonymous update (⇒ Sect. 3.7.4). The above definitions mean that, in particular, the following terms are going to be parsed like this:

─── KeY Syntax ───

```
{\subst int i; 2} i = i        is parsed as
({\subst int i; 2} i) = i

{o.a := 1} o.a = o.a           is parsed as
({o.a := 1} o.a) = o.a
```

─────────────────────── KeY Syntax ───

In updates, update quantification **\for** and update guard **\if** are optional.
  Function and predicate expressions are constructed in the following way:

─── KeY Syntax ───

⟨*functionPredicateTerm*⟩ ::=
  ⟨*functionPredicateName*⟩ (⟨*dependencyList*⟩)? ((⟨*argumentList*⟩))?

⟨*functionPredicateName*⟩ ::=
    ⟨*identifier*⟩
  | ⟨*sortExp*⟩ :: ⟨*sortOperator*⟩

⟨*sortOperator*⟩ ::= ⟨*identifier*⟩

⟨*dependencyList*⟩ ::=
  [⟨*dependencies*⟩ (, ⟨*dependencies*⟩)*]

⟨*dependencies*⟩ ::=
  ((⟨*attributeExp*⟩ | ⟨*arrayExp*⟩) ;)+

⟨*arrayExp*⟩ ::= [] (⟨*typeReference*⟩)

─────────────────────── KeY Syntax ───

The dependency list can only be present if the corresponding function or predicate has been declared to be non-rigid. The use and meaning of dependencies in function and predicate expressions have been explained in Section 15.2.1. The second alternative in ⟨*functionPredicateName*⟩ is for predefined functions and predicates that relate to sort operations (built-in sort functions and predicates), ⟨*sortOperator*⟩ can be any of the following identifiers:

─── KeY Syntax ───

```
including excluding includes excludes sum emptySet
isEmpty notEmpty size intersection symmetricDifference
instance union without
```

─────────────────────── KeY Syntax ───

Simple variables, conditional terms, and abbreviations are defined as follows:

─── KeY Syntax ───

$\langle variable \rangle$ ::= $\langle identifier \rangle$

$\langle abbrTerm \rangle$ ::= @ $\langle identifier \rangle$

$\langle conditionalTerm \rangle$ ::=
  (**\if** | **\ifEx** $\langle variableBinding \rangle$)
    ($\langle formula \rangle$)
      \then ($\langle term \rangle$)
      \else ($\langle term \rangle$)

─── KeY Syntax ───

A variable can be a logic or a program variable (normal parsing mode), or a schema variable (schematic mode). An abbreviation expression refers to an identifier that contains a term abbreviation.

A special term is built in the following way:

─── KeY Syntax ───

$\langle specialTerm \rangle$ ::=
    $\langle metaTerm \rangle$
  | **\inType** (
           -? $\langle schemaVariable \rangle$
        | $\langle schemaVariable \rangle$ $\langle arithOp \rangle$ $\langle schemaVariable \rangle$
)

$\langle metaTerm \rangle$ ::=
  $\langle metaOperator \rangle$ (($\langle argumentList \rangle$))?

$\langle arithOp \rangle$ ::= + | - | * | / | %

$\langle schemaVariable \rangle$ ::=
$\langle identifier \rangle$

$\langle metaOperator \rangle$ ::= $\langle identifier \rangle$

─── KeY Syntax ───

Special terms can only occur in the schematic mode. Meta-terms are used to construct taclet meta-operator expressions ($\Rightarrow$ Sect. 4.2.8). Currently, valid meta-operator identifiers are the following:

─── KeY Syntax ───

```
#lengthReference #created #nextToCreate
#traInitialized #transient #transactionCounter #shadowed
#add #sub #mul #div #jdiv #mod #jmod #less #greater
#leq #geq #eq #JavaIntUnaryMinus #JavaLongUnaryMinus
#JavaIntAdd #JavaIntSub #JavaIntMul #JavaIntDiv #JavaIntMod
#JavaIntAnd #JavaIntOr #JavaIntXor #JavaIntComplement
#JavaIntShiftRight #JavaIntShiftLeft
```

```
#JavaIntUnsignedShiftRight #JavaLongAdd #JavaLongSub
#JavaLongMul #JavaLongDiv #JavaLongMod #JavaLongAnd
#JavaLongOr #JavaLongXor #JavaLongComplement
#JavaLongShiftRight #JavaLongShiftLeft
#JavaLongUnsignedShiftRight
#moduloByte #moduloShort #moduloInteger #moduloLong
#whileInvRule #introNewAnonUpdate
#arrayBaseInstanceOf #arrayStoreStaticAnalyse
#expandDynamicType #ResolveQuery #constantvalue
#Universes #allSubtypes
#divideMonomials #divideLCRMonomials
#createInReachableStatePO
```
—— KeY Syntax ——

The ideas behind the **\inType** special operator are described in Chapter 12.
Note that different meta-operators/constructs are used inside modalities for
the schematic JAVA code blocks (⇒ Sect. B.5.5).

Finally, the remaining term building blocks are number ⟨*number*⟩ con-
stants, string ⟨*string*⟩ and character ⟨*character*⟩ constants, grouping with
parenthesis (), and logic constants **true** and **false**.

To sum up this section, here are some examples of properly built terms
and formulae. In normal mode:

—— KeY Syntax ——
```
false -> true
\forall int i; (i + i = 2 * i & i - i = 0)
\forall int i; (add(i, i) = mul(2, i) & sub(i, i) = 0)
\exists java.lang.Object[] o; o != null
{\for int i; \if (i >= 0 & i < a.length)
   a[i].<transient> := 1, o := a} (o[0].<transient> = 1)
StaticClass.staticAttr <= StaticClass.staticQuery()
o.a@(ClassA).b@(ClassB) =
   o.identity@(ClassA)(o.a@(ClassA).b@(ClassB))
java.lang.Object::instance(o) = TRUE
\< {i = 1;} \> i = 1
```
—— KeY Syntax ——

And in schematic mode:

—— KeY Syntax ——
```
\forall #v; (#v + #v = 2 * #v & #v - #v = 0)
\exists #v; #v != null
{\for #v; \if (#v >= 0 & #v < a.length)
  #transient(#a[#v]) := 1} (#transient(#a[#v]) = 1)
AnySort::instance(#v) = TRUE
#o.#a != #se0 + #se1
```
—— KeY Syntax ——

B.3 Rule Files

All rule files (system and user defined) are parsed only in schematic mode.
On the top level, a rule file has the following form:

—— KeY Syntax ——————————————————————

⟨*ruleFile*⟩ ::=
　⟨*libraryIncludeStatement*⟩∗
　⟨*ruleFileDeclarations*⟩∗
　⟨*ruleBlock*⟩∗

———————————————————————— KeY Syntax ——

B.3.1 Library and File Inclusion

The KeY system supports file inclusion on two levels: (low) file level, and
(high) library level. File inclusion statements can appear *anywhere* in the
KeY input, and take the following form:

—— KeY Syntax ——————————————————————

⟨*fileInclusion*⟩ ::=
　\includeFile " ⟨*fileName*⟩ ";

———————————————————————— KeY Syntax ——

The effect of **\includeFile** is that KeY unconditionally redirects its input
to the indicated file ⟨*fileName*⟩. When the indicated file is read in, the parsing
in the current file continues. File inclusion nesting is allowed and its depth is
not limited by the KeY system itself.

　Library file inclusion can be done with the following statements:

—— KeY Syntax ——————————————————————

⟨*libraryIncludeStatement*⟩ ::=
　(**\include** | **\includeLDTs**)
　　⟨*libraryFileName*⟩ (, ⟨*libraryFileName*⟩)∗ ;

———————————————————————— KeY Syntax ——

The major feature of the library inclusion statements is that each library file
is going to be read in once, even if the same library is requested multiple times
(for example, because of circular dependencies). On the implementation level,
when the library files are read in **\includeLDTs** performs slightly different
operations than **\include**.

B.3.2 Rule File Declarations

Each rule file can have the following declarations:

—— KeY Syntax ————————————————————————————

$\langle ruleFileDeclarations \rangle$ ::=
    $\langle ruleSetsDecl \rangle$
  | $\langle optionsDecl \rangle$
  | $\langle sortsDecls \rangle$
  | $\langle schemaVariablesDecl \rangle$
  | $\langle functionsDecl \rangle$
  | $\langle predicatesDecl \rangle$

——————————————————————————————— KeY Syntax ——

Rule sets and options are declared in the following way:

—— KeY Syntax ————————————————————————————

$\langle ruleSetsDecl \rangle$ ::=
  \heuristicsDecl { ($\langle ruleSetName \rangle$;)∗ }

$\langle optionsDecl \rangle$ ::=
  \optionsDecl { ($\langle oneOptionDecl \rangle$;)∗ }

$\langle oneOptionDecl \rangle$ ::=
  $\langle optionName \rangle$: { $\langle optionValue \rangle$ (, $\langle optionValue \rangle$)∗ }

$\langle ruleSetName \rangle$ ::= $\langle identifier \rangle$

$\langle optionName \rangle$ ::= $\langle identifier \rangle$

$\langle optionValue \rangle$ ::= $\langle identifier \rangle$

——————————————————————————————— KeY Syntax ——

Examples of valid rule set and option declarations are:

—— KeY Syntax ————————————————————————————

```
\heuristicsDecl { simplify_int; simplify_prog; }

\optionsDecl {
  nullPointerPolicy:{nullCheck, noNullCheck};
  programRules:{Java, ODL};
}
```

——————————————————————————————— KeY Syntax ——

Sorts are declared in the following way:

――― KeY Syntax ―――

⟨*sortsDecl*⟩ ::=
  \sorts { (⟨*oneSortDecl*⟩ ;)* }

⟨*oneSortDecl*⟩ ::=
    \object ⟨*sortNameList*⟩
  | **\generic** ⟨*sortNameList*⟩
      (**\extends** ⟨*sortNameList*⟩)? (**\oneof** { ⟨*sortNameList*⟩ })?
  | ⟨*intersectionSort*⟩
  | ⟨*sortName*⟩ **\extends** ⟨*sortNameList*⟩
  | ⟨*sortNameList*⟩

⟨*sortNameList*⟩ ::= ⟨*sortName*⟩ (, ⟨*sortName*⟩)*

――― KeY Syntax ―――

The definitions of ⟨*intersectionSort*⟩ and ⟨*sortName*⟩ have been given earlier. Note, that here ⟨*sortName*⟩ may be (only in some places) required to be a simple ⟨*identifier*⟩, and cannot be a fully qualified sort name.

Schema variables are declared in the following way:

――― KeY Syntax ―――

⟨*schemaVariablesDecl*⟩ ::=
  \schemaVariables { (⟨*schemaVarDecl*⟩ ;)* }

⟨*schemaVarDecl*⟩ ::=
    (**\modalOperator** | **\operator** (⟨*sortName*⟩))
        { ⟨*operatorList*⟩ } ⟨*variableList*⟩
  | **\formula** ⟨*schemaModifiers*⟩? ⟨*variableList*⟩
  | **\location** ⟨*schemaModifiers*⟩? ⟨*variableList*⟩
  | **\function** ⟨*schemaModifiers*⟩? ⟨*variableList*⟩
  | **\program** ⟨*schemaModifiers*⟩? ⟨*programSchemaVarSort*⟩ ⟨*variableList*⟩
  | **\term** ⟨*schemaModifiers*⟩? ⟨*sortName*⟩ ⟨*variableList*⟩
  | **\variables** ⟨*sortName*⟩ ⟨*variableList*⟩
  | **\skolemTerm** ⟨*sortName*⟩ ⟨*variableList*⟩

⟨*programSchemaVarSort*⟩ ::= ⟨*identifier*⟩

⟨*schemaModifiers*⟩ ::= [⟨*identifier*⟩ (, ⟨*identifier*⟩)*]

⟨*variableList*⟩ ::= ⟨*identifier*⟩ (, ⟨*identifier*⟩)*

⟨*operatorList*⟩ ::= ⟨*identifier*⟩ (, ⟨*identifier*⟩)*

――― KeY Syntax ―――

Schema variable modifiers can be `list`, `rigid`, or `strict`. The list of currently defined program schema variable sorts is the following:

─── KeY Syntax ───

```
LeftHandSide Variable StaticVariable SimpleExpression
NonSimpleExpression Expression Literal Label
InstanceCreation ArrayCreation ArrayInitializer
SpecialConstructorReference LoopInit Guard ForUpdates
MultipleVariableDeclaration  ArrayPostDeclaration
Switch ImplicitVariable ExplicitVariable
ConstantVariable ImplicitReferenceField VariableInitializer
ImplicitClassInitialized NonSimpleMethodReference
Statement Catch MethodBody NonModelMethodBody Type
NonPrimitiveType MethodName ExecutionContext
ContextStatementBlock <allocate>

JavaBooleanExpression JavaByteExpression
JavaCharExpression JavaShortExpression JavaIntExpression
JavaLongExpression JavaByteShortExpression
JavaByteShortIntExpression AnyJavaTypeExpression
AnyNumberTypeExpression SimpleStringExpression

ImplicitClassInitializationInProgress ImplicitClassErroneous
ImplicitClassPrepared ImplicitNextToCreate ImplicitCreated
ImplicitTraInitialized ImplicitTransactionCounter ArrayLength

makeTransientBooleanArray makeTransientByteArray
makeTransientShortArray makeTransientObjectArray
jvmArrayCopy jvmArrayCopyNonAtomic jvmArrayFillNonAtomic
jvmArrayCompare jvmMakeShort jvmSetShort jvmIsTransient
jvmBeginTransaction jvmCommitTransaction jvmAbortTransaction
jvmSuspendTransaction jvmResumeTransaction
```
─── KeY Syntax ───

Some examples of properly declared schema variables:

─── KeY Syntax ───

```
\schemaVariables {
  \modalOperator {diamond, box, throughout} #puremodal;
  \operator (int) {add, sub, mul, mod, div};
  \formula post, inv, post1;
  \program Type #t, #t2 ;
  \program[list] Catch #cs ;
  \location[list] #modifies;
  \function[list] anon1, anon2, anon3;
  \term[rigid,strict] H h;
  \variables G x;
```
─── KeY Syntax ───

Function and predicate declarations are very similar to each other, the only
difference is that there is no result type specified for predicates:

— KeY Syntax ——————————————————————————————————

$\langle functionsDecl \rangle$ ::=
  \functions $\langle optionSpecs \rangle$? { ($\langle oneFunctionDecl \rangle$;)∗ }

$\langle predicatesDecl \rangle$ ::=
  \predicates $\langle optionSpecs \rangle$? { ($\langle onePredicateDecl \rangle$;)∗ }

$\langle oneFunctionDecl \rangle$ ::=
  (**\nonRigid** ([Location])?)?
      $\langle sortExp \rangle$ $\langle functionPredicateName \rangle$ ($\langle dependencyList \rangle$)?
        $\langle argumentSorts \rangle$?

$\langle onePredicateDecl \rangle$ ::=
  (**\nonRigid** ([Location])?)?
      $\langle functionPredicateName \rangle$ ($\langle dependencyList \rangle$)?
        $\langle argumentSorts \rangle$?

$\langle argumentSorts \rangle$ ::= ($\langle sortExp \rangle$ (, $\langle sortExp \rangle$)∗)

$\langle optionSpecs \rangle$ ::= ($\langle optionSpecList \rangle$)

$\langle optionSpecList \rangle$ ::= $\langle oneOptionSpec \rangle$ (, $\langle oneOptionSpec \rangle$)∗

$\langle oneOptionSpec \rangle$ ::= $\langle optionName \rangle$: $\langle optionValue \rangle$

———————————————————————————————————— KeY Syntax —

In the above $\langle dependencyList \rangle$ can only occur if the function or predicate is
declared to be non-rigid. The option specification tell the the system that
the declared functions or predicates should only be visible when the specified
option is active. Some function and predicate declaration examples:

— KeY Syntax ——————————————————————————————————

```
\functions(intRules:javaSemantics) {
  int unaryMinusJint(int);
}

\predicates {
  \nonRigid Acc(java.lang.Object, any);
}
```

———————————————————————————————————— KeY Syntax —

B.3.3 Rules

Rules (taclets) are defined in **\rules** blocks this way:

──── KeY Syntax ───

$\langle ruleBlock \rangle$ ::=
  \rules $\langle optionSpecs \rangle$? { ($\langle taclet \rangle$;)* }

── KeY Syntax ────

The option specification has the same meaning as for the function and predicate declarations. Each taclet can have additional (per taclet) option specifications and local schema variable declarations. The syntax for a taclet is:

──── KeY Syntax ───

$\langle taclet \rangle$ ::=
  $\langle identifier \rangle$ $\langle optionSpecs \rangle$? {
      (**\schemaVar** $\langle schemaVarDecl \rangle$;)*
      $\langle contextAssumptions \rangle$? $\langle findPattern \rangle$?
      $\langle stateCondition \rangle$? $\langle variableConditions \rangle$?
      ($\langle goalTemplateList \rangle$ | **\closegoal**)
      $\langle tacletModifiers \rangle$*
  }

$\langle contextAssumptions \rangle$ ::= **\assumes** ($\langle schematicSequent \rangle$)

$\langle findPattern \rangle$ ::= **\find** ($\langle termOrSequent \rangle$)

$\langle stateCondition \rangle$ ::= **\inSequentState** | **\sameUpdateLevel**

$\langle variableConditions \rangle$ ::= **\varcond** ($\langle variableConditionList \rangle$)
$\langle variableConditionList \rangle$ ::= $\langle variableCondition \rangle$ (, $\langle variableCondition \rangle$)*

$\langle goalTemplateList \rangle$ ::= $\langle goalTemplate \rangle$ (; $\langle goalTemplate \rangle$)*

$\langle schematicSequent \rangle$ ::= $\langle termList \rangle$? **==>** $\langle termList \rangle$?

$\langle termList \rangle$ ::= $\langle term \rangle$ (, $\langle term \rangle$)*

$\langle termOrSequent \rangle$ ::= $\langle term \rangle$ | $\langle schematicSequent \rangle$

$\langle tacletModifiers \rangle$ ::=
      \heuristics ($\langle identifierList \rangle$)
   | **\noninteractive**
   | **\recursive**
   | **\displayname** $\langle string \rangle$
   | **\helptext** $\langle string \rangle$

$\langle identifierList \rangle$ ::= $\langle identifier \rangle$ (, $\langle identifier \rangle$)*

── KeY Syntax ────

A variable condition can be one of the following:

─── KeY Syntax ───────────────────────────────────────

⟨*variableCondition*⟩ ::=
      \new (⟨*variable*⟩ ,
              ⟨*typeCondExp*⟩
            | **\dependingOn** (⟨*variable*⟩)
            | **\dependingOnMod** (⟨*variable*⟩)
)
   | **\notSameLiteral** (⟨*variable*⟩ , ⟨*variable*⟩)
   | **\notFreeIn** (⟨*variable*⟩ (, ⟨*variable*⟩)+)
   | **\hasSort** (⟨*variable*⟩ , ⟨*sortExp*⟩)
   | **\isQuery** (⟨*variable*⟩)
   | **\isInReachableState** (⟨*variable*⟩)
   | **\isReference** ([non_null])? (⟨*typeCondExp*⟩)
   | **\not**? **\staticMethodReference** (
              ⟨*variable*⟩ , ⟨*variable*⟩ , ⟨*variable*⟩)
   | **\not**? **\isReferenceArray** (⟨*variable*⟩)
   | **\not**? **\static** (⟨*variable*⟩)
   | **\not**? **\isLocalVariable** (⟨*variable*⟩)
   | **\not**? **\isAbstractOrInterface** (⟨*typeCondExp*⟩)
   | **\not**? ⟨*typeComparison*⟩ (⟨*typeCondExp*⟩ , ⟨*typeCondExp*⟩)

⟨*typeCondExp*⟩ ::=
      \typeof (⟨*variable*⟩)
   | **\containerType** (⟨*variable*⟩)
   | ⟨*sortExp*⟩

⟨*typeComparison*⟩ ::=
   \same | **\compatible** | **\strict**? **\sub**
                                              ─── KeY Syntax ───

The goal specification is defined as follows:

─── KeY Syntax ───────────────────────────────────────

⟨*goalTemplate*⟩ ::=
      ⟨*optionSpecs*⟩ { ⟨*oneGoalTemplate*⟩ }
   | ⟨*oneGoalTemplate*⟩

⟨*oneGoalTemplate*⟩ ::=
   ⟨*branchName*⟩?
        ⟨*replaceGoal*⟩ ⟨*addGoal*⟩? ⟨*addRules*⟩? ⟨*addProgramVars*⟩?
      | ⟨*addGoal*⟩ ⟨*addRules*⟩?
      | ⟨*addRules*⟩

⟨*branchName*⟩ ::= ⟨*string*⟩ :

⟨*replaceGoal*⟩ ::= **\replacewith** (⟨*termOrSequent*⟩)

⟨*addGoal*⟩ ::= **\add** (⟨*schematicSequent*⟩)

⟨*addRules*⟩ ::=
  \addrules (⟨*taclet*⟩ (, ⟨*taclet*⟩)*)

⟨*addProgramVars*⟩ ::=
  \addprogvars (⟨*variable*⟩ (, ⟨*variable*⟩)*)

——————————————————————————————— KeY Syntax ——

Some examples of properly formed taclets:

—— KeY Syntax ———————————————————————————————

```
eliminateVariableDeclaration {
  \find (\<{.. #t #v0; ...}\> post)
    \replacewith (\<{..  ...}\> post)
    \addprogvars(#v0)
  \heuristics(simplify_prog, simplify_prog_subset)
  \displayname "eliminateVariableDeclaration"
};

makeInsertEq {
  \find (sr = tr ==>)
  \addrules ( insertEq { \find (sr) \replacewith (tr) } )
  \heuristics (simplify)
  \noninteractive
};

cut {
  "cut:␣#b␣TRUE":  \add (#b ==>);
  "cut:␣#b␣FALSE": \add (==> #b)
};
```

——————————————————————————————— KeY Syntax ——

B.4 User Problem and Proof Files

User problem and proof files are almost the same, the only difference is that
the problem file does not contain a **\proof** section. User problem and proof
files have some additional elements as compared to rules files, and all the
elements of the rule files can be present in a problem/proof file:

────── KeY Syntax ──────────────────────────────────────

$\langle userProblemProofFile \rangle$ ::=
  $\langle proverSettings \rangle$?
  $\langle javaSource \rangle$?
  $\langle libraryIncludeStatement \rangle *$
  $\langle tacletOptionActivation \rangle$?
  $\langle programVariablesDecl \rangle$?
  $\langle ruleFileDeclarations \rangle *$
  $\langle contracts \rangle *$
  $\langle ruleBlock \rangle *$
  $\langle problem \rangle$
  $\langle proof \rangle$?

── KeY Syntax ──

The following simple definitions cover most of the problem/proof file syntax:

────── KeY Syntax ──────────────────────────────

$\langle proverSettings \rangle$ ::= **\settings** { $\langle string \rangle$ }

$\langle javaSource \rangle$ ::= **\javaSource** " $\langle fileName \rangle$ ";

$\langle tacletOptionActivation \rangle$ ::= **\withOptions** $\langle optionSpecList \rangle$;

$\langle programVariablesDecl \rangle$ ::=
  \programVariables { ($\langle programVarDecl \rangle$;)* }

$\langle programVarDecl \rangle$ ::= $\langle typeReference \rangle$ $\langle variableList \rangle$

$\langle problem \rangle$ ::= **\problem** { $\langle formula \rangle$ }

$\langle proof \rangle$ ::= **\proof** { $\langle proofTree \rangle$ }

──────────────────────────────────── KeY Syntax ──

The **\programVariables** section defines program variables local to the the
problem, for example:

────── KeY Syntax ──────────────────────────────

\programVariables { java.lang.Object o; }

\problem { **\<** {o = **new** Object();} **\>** o != null }

──────────────────────────────────── KeY Syntax ──

The parameter to **\settings** is a string containing the description of prover
settings in a category/property list form, similar to this:

—— KeY Syntax ——————————————————————

```
\settings {
"#Proof-Settings-Config-File
#Tue␣Apr␣04␣15:36:57␣CEST␣2006
[General]SuggestiveVarNames=false
[General]OuterRenaming=true
[View]FontIndex=0
[View]ShowWholeTaclet=false
[SimultaneousUpdateSimplifier]DeleteEffectLessLocations=true
[DecisionProcedure]=SIMPLIFY
..." }
```

—————————————————————————— KeY Syntax ——

The **\proof** contains a proof tree in the form of Lisp-like nested lists. Since proof trees are in principle not supposed to be edited by the user manually, we skip the detailed description of the proof tree syntax. An example of a proof tree is the following:

—— KeY Syntax ——————————————————————

```
\proof {
(keyLog "0" (keyUser "woj" ) (keyVersion "0.2184"))

(branch "dummy␣ID"
  (rule "concrete_and_1" (formula "1") (term "0")
     (userinteraction "n"))
  (rule "concrete_and_2" (formula "1") (term "1")
     (userinteraction "n"))
  (rule "concrete_eq_4" (formula "1") (userinteraction "n"))
  (rule "concrete_not_2" (formula "1") (userinteraction "n"))
  (rule "close_by_true" (formula "1") (userinteraction "n"))
)
}
```

—————————————————————————— KeY Syntax ——

B.4.1 Method Contracts

Finally, method contracts expressed directly in JAVA CARD DL take the following form in the KeY syntax:

—— KeY Syntax ——————————————————————

⟨*contracts*⟩ ::=
  \contracts { (⟨*oneContract*⟩ ;)∗ }

⟨*oneContract*⟩ ::=
  ⟨*identifier*⟩ **{**
    ⟨*programVariablesDecl*⟩?
    ⟨*prePostFormula*⟩
    \modifies { ⟨*locationList*⟩ **}**
    (**\heuristics (** ⟨*identifierList*⟩ **))?**
    (**\displayname** ⟨*string*⟩ **)?**
  }

⟨*prePostFormula*⟩ ::=
  ⟨*formula*⟩ **->** ⟨*modalityBlock*⟩ ⟨*formula*⟩

⟨*locationList*⟩ ::= ⟨*oneLocation*⟩ **(** **,** ⟨*oneLocation*⟩ **)∗**

⟨*oneLocation*⟩ ::=
    ∗
  | **(** **\for** ⟨*variableBinding*⟩ **)?** **(** **\if** ⟨*formula*⟩ **)?** ⟨*accessTerm*⟩

————— KeY Syntax —————

Here program variables are declared locally for a contract. The contract formula ⟨*prePostFormula*⟩ has to be in special form (Hoare Triple)—the program blocks appearing inside the modality are limited to single method body reference expression and special exception catching constructs (⇒ Sect. B.5). Also, in contracts, an ⟨*accessTerm*⟩ can contain quantified array expressions (⇒ Sect. B.2.2). An explicit quantification of expressions with **\for** is also possible. An example of a contract is the following:

————— KeY Syntax —————

```
\contracts {
  Demoney_setUndefined {
    \programVariables {
      byte b;
      fr.trustedlogic.demo.demoney.Demoney demoney;
    }
    demoney.definedParamFlags != null ->
    \<{
      demoney.setUndefined(b)
         @fr.trustedlogic.demo.demoney.Demoney;
    }\> demoney.definedParamFlags != null
    \modifies { demoney.definedParamFlags[*] }
    \displayname "setUndefined"
  };
}
```

————— KeY Syntax —————

B.5 Schematic JAVA Syntax

In principle, inside a JAVA CARD DL modality any valid JAVA code block can be placed, that is, any JAVA code block that would be allowed in a JAVA method implementation. On top of that, the KeY system allows extensions to the regular JAVA syntax. We are not going to discuss the JAVA syntax, we assume that in this context it is common knowledge. Similarly to terms, different rules for the code inside a modality apply when the schematic mode is used for parsing the rule files, as explained below. All JAVA blocks that appear inside a modality have to be surrounded with a pair of braces {}. In the following, JAVA keywords that appear in JAVA blocks are marked with **bold**.

B.5.1 Method Calls, Method Bodies, Method Frames

In normal parsing mode the following construct can be used to refer to method's body/implementation:

—— KeY Syntax ——————————————————————

$\langle methodBody\rangle$ ::=
  ($\langle resultLoc\rangle$ =)?
    $\langle staticClassOrObjectRef\rangle$. $\langle methodName\rangle$
      $\langle methodArguments\rangle$ @ $\langle classReference\rangle$;

————————————————————————— KeY Syntax ——

The following are properly constructed method body references:

—— KeY Syntax ——————————————————————

```
o.method()@MyClass;
result = pack.StaticClass.method(o, 2)@pack.StaticClass;
```

————————————————————————— KeY Syntax ——

On the top level, any JAVA block can also be enclosed in a method frame to provide the method execution context. A method frame expression takes the following form:

—— KeY Syntax ——————————————————————

$\langle methodFrame\rangle$ ::=
  method-frame(result -> $\langle resultLoc\rangle$,
                source = $\langle classReference\rangle$,
                this = $\langle variable\rangle$) : {
    $\langle javaBlock\rangle$
  }

————————————————————————— KeY Syntax ——

For example:

```
 KeY Syntax
\<{ method-frame(result->j, source=MyClass, this=c)
    : {
        this.a=10;
        return this.a;
    }
}\> ...
                                             KeY Syntax
```

B.5.2 Exception Catching in Contracts

When a JAVA CARD DL method contract is constructed a method body reference inside the modality can be enclosed in an exception catching construct to allow exceptional specification of a method. The syntax is the following:

```
 KeY Syntax
⟨contractExceptionCatch⟩ ::=
    #catchAll(⟨classReference⟩ ⟨variable⟩) {
      ⟨methodBody⟩
    }
                                             KeY Syntax
```

For example:

```
 KeY Syntax
#catchAll(Exception e) {
  o.method()@MyClass;
}
                                             KeY Syntax
```

B.5.3 Inactive JAVA Block Prefix and Suffix

In the schematic mode several extensions to the JAVA syntax are available. First, the inactive prefix and suffix of the JAVA block (⇒ Sect. 3.6) can be given with .. and ..., following this syntax:

```
 KeY Syntax
⟨inactivePrefixSuffix⟩ ::=
    .. ⟨javaBlock⟩ ...
                                             KeY Syntax
```

The inactive prefix and suffix constructs can be seen as special cases of program schema variables.

B.5.4 Program Schema Variables

Any program part inside the modality can be replaced with a corresponding schema variable, provided a schema variable of a proper kind is provided to match the given element in Java code ($\Rightarrow$ Sect. B.3.2). Additionally some elements of the inactive prefix can also be matched with a schema variable to refer to the execution context data. For example, the following are valid schematic Java blocks:

—— KeY Syntax ——————————————————
```
.. #loc = #se; ...              Assignment
.#ex.. #lb: throw #se; ...      Execution Context & Label
.. #t #v = #exp; ...            Variable declaration
.. #se.#mn(#selist);            Method call
```
———————————————————— KeY Syntax ——

B.5.5 Meta-constructs

When in schematic mode, the KeY system offers a variety of meta-constructs to perform local program transformations and new code introduction related to the corresponding symbolic execution rules of Java code. Such meta-constructs can only be used in the modalities that are part of the **\replacewith** or **\add** taclet goal specifiers. For example the following meta-construct can be used to introduce proper method body reference into the analysed Java code:

—— KeY Syntax ——————————————————
```
.. #method-call(#se.#mn(#selist)); ...
```
———————————————————— KeY Syntax ——

The full list of schematic Java meta-constructs is the following:

—— KeY Syntax ——————————————————
```
#unwind-loop #unpack #switch-to-if #do-break
#evaluate-arguments #replace #resolve-multiple-var-decl
#typeof #length-reference

#method-call #method-call-contract #expand-method-body
#constructor-call #special-constructor-call

#create-object #post-work #array-post-declaration
#init-array-creation #init-array-creation-transient
#init-array-assignments

#static-initialisation #isstatic #static-evaluate
```
———————————————————— KeY Syntax ——

B.5.6 Passive Access in Static Initialisation

Finally, the static initialisation rules ($\Rightarrow$ Sect. 3.6.6) extend JAVA syntax with
the passive (or raw) access operator:

—— KeY Syntax ——————————————————————

$\langle passiveAccessExp \rangle$ ::= @($\langle attributeVariableAccess \rangle$)

——————————————————————— KeY Syntax ——

The passive access operator @ can be used both in normal (only when appro-
priate) and schematic mode.

References

Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Proc. TAPSOFT: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *LNCS*, pages 682–696. Springer, 1997.

Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proc. Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy*, volume 2805, pages 51–74. Springer, September 2003.

Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.

Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool: integrating object oriented design and formal verification. *Software and System Modeling*, 4(1):32–54, 2005a.

Wolfgang Ahrendt, Andreas Roth, and Ralf Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In Geoff Sutcliffe and Andrei 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, December 2005b.

Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Number 2 in Center for Environmental Structure series. Oxford University Press, New York, 1977.

Christopher Alexander. The origins of pattern theory: The future of the theory and the generation of a living world. *IEEE Software*, pages 71–82, September/October 1999.

Paulo Sergio Almeida. Controlling sharing of state in data types. In M. Aksit and S. Matsuoka, editors, *ECOOP '97–Object-Oriented Programming*,

volume 1241 of *LNCS*, pages 32–59. Springer, 1997.

Gustav Andersson. OCL constraints for design patterns in the KeY project. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, June 2005.

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean 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.

Thomas Baar. OCL and graph transformations: A symbiotic alliance to alleviate the frame problem. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops, Montego Bay, Jamaica, Revised Selected Papers*, volume 3844 of *LNCS*, pages 20–31. Springer, 2006.

Thomas Baar, Reiner Hähnle, Theo Sattler, and Peter H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In Kurt Mehlhorn and Gregor Snelting, editors, *Softwaretechnik-Trends*, Informatik Aktuell, pages 389–404. Springer-Verlag, September 2000.

Thomas Baar, Bernhard Beckert, and Peter H. Schmitt. An extension of Dynamic Logic for modelling OCL's @*pre* operator. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Proc. Fourth Andrei Ershov International Conference, Perspectives of System Informatics, Novosibirsk, Russia*, volume 2244 of *LNCS*, pages 47–54. Springer, 2001.

Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-200-14, Microsoft Research, 2000.

Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proc. Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004.

Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas 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, 2000.

Richard Banach and Michael Poppleton. Retrenchment: An Engineering Variation on Refinement. In Didier Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings*, volume 1393 of *LNCS*, pages 129–147. Springer, 1998.

Richard Banach and Michael Poppleton. Sharp retrenchment, modulated refinement and punctured simulation. *Formal Aspects of Computing*, 11: 498–540, 1999.

Michael Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen and Carron Shank-

land, editors, *Proc. Mathematics of Program Construction, 7th International Conference, Stirling, Scotland, UK*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proc. 17th IEEE Computer Security Foundations Workshop, CSFW-17, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004.

Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logic. In Bernhard Beckert and Bernhard Aichernig, editors, *Proc. Software Engineering and Formal Methods Conference, Koblenz, Germany*, pages 86–95. IEEE Computer Society, 2005.

Markus Baum. Proof Visualization. Studienarbeit, Department of Computer Science, University of Karlsruhe, 2006.

Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer, 2001.

Bernhard Beckert and Vladimir Klebanov. Must program verification systems and calculi be verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, pages 34–41, 2006.

Bernhard Beckert and Wojciech Mostowski. A program logic for handling Java Card's transaction mechanism. In Mauro Pezzé, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE), Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer-Verlag, 2003.

Bernhard Beckert and André 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, 2006.

Bernhard Beckert and Bettina Sasse. Handling Java's abrupt termination in a sequent calculus for Dynamic Logic. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 5–14. Technical Report DII 07/01, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, 2001.

Bernhard Beckert and Steffen Schlager. Software verification with integrated data type refinement for integer arithmetic. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, volume 2999 of *LNCS*, pages 207–226. Springer, 2004.

Bernhard Beckert and Steffen Schlager. Refinement and retrenchment for programming language data types. *Formal Aspects of Computing*, 17(4): 423–442, 2005.

Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer, 2001.

Bernhard Beckert and Peter H. Schmitt. Program verification using change information. In *Proceedings, Software Engineering and Formal Methods (SEFM), Brisbane, Australia*, pages 91–99. IEEE Press, 2003.

Bernhard Beckert and Kerry Trentelman. Second-order principles in specification languages for object-oriented programs. In G. Suttcliff and A. Voronkov, editors, *Proceedings, 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Montego Bay, Jamaica*, volume 3835 of *LNCS*. Springer, 2005.

Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: a new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas*, 98(1):17–53, 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.

Bernhard Beckert, Thorsten Bormer, and Vladimir Klebanov. Reusing proofs when program verification systems are modified. In *Proc. Software Certificate Management Workshop (SoftCeMent), Long Beach, USA*, pages 41–46, 2005a.

Bernhard Beckert, Steffen Schlager, and Peter H. Schmitt. An improved rule for while loops in deductive program verification. In Kung-Kiu Lau, editor, *Proceedings, Seventh International Conference on Formal Engineering Methods (ICFEM), Manchester, UK*, volume 3785 of *LNCS*, pages 315–329. Springer, 2005b.

Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer Verlag, second edition, 2003.

Eerke Boiten and John Derrick. IO-Refinement in Z. In A. Evans, D. Duke, and T. Clark, editors, *3rd BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer-Verlag, September 1998.

Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In Cindy Norris and Jr. James B. Fenwick, editors, *Proc. 30th ACM SIGPLAN-SIGACT Symposium on Principles of*

programming languages (POPL), volume 38, 1 of *ACM SIGPLAN Notices*, pages 213–223, New Orleans, Louisiana, January 2003. ACM Press.

Robert Boyer. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003.

Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

Cees-Bart Breunesse. *On JML: Topics in Tool-assisted Verification of Java Programs*. PhD thesis, Radboud University of Nijmegen, 2006.

Cees-Bart Breunesse and Erik Poll. Verifying JML specifications with model fields. In *Proc. Workshop on Formal Techniques for Java-like Programs*, pages 51–60, 2003. Technical Report 408, ETH Zurich.

Cees-Bart Breunesse, Bart Jacobs, and Joachim van den Berg. Specifying and verifying a decimal representation in Java for smart cards. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology (AMAST'02)*, LNCS, pages 304–318. Springer-Verlag, 2002.

Manfred Broy and Peter Pepper. Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite algorithm. *ACM Transactions on Programming Languages and Systems*, 4(3):362–381, 1982. doi: http://doi.acm.org/10.1145/357172.357175.

Achim D. Brucker and Burkhart Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In César Muñoz, Sophiène Tahar, and Víctor Carreño, editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *LNCS*, pages 99–114. Springer-Verlag, Hampton, VA, USA, 2002.

Richard Bubel. Behandlung der Initialisierung von Klassen und Objekten in einer dynamischen Logik für Java Card. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, August 2001.

Richard Bubel and Reiner Hähnle. Integration of informal and formal development of object-oriented safety-critical software — a case study with the KeY system. *Software Tools for Technology Transfer*, 7(3):197–211, June 2005.

Richard Bubel, Andreas Roth, and Philipp Rümmer. Ensuring correctness of lightweight tactics for Java Card dynamic logic. In *Prel. Proc. of Workshop on Logical Frameworks and Meta-Languages (LFM)*, pages 84–105, 2004.

Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proc. Formal Methods Europe, Pisa, Italy*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.

Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

David A. Burke and Kristofer Johannisson. Translating formal software specifications to natural language: A grammar-based approach. In Philippe Blache, Edward Stabler, Joan Busquets, and Richard Moot, editors, *Proc. Logical Aspects of Computational Linguistics (LACL)*, volume 3402 of *LNCS*, pages 51–66. Springer-Verlag, 2005.

Daniel Burrows. Static encapsulation analysis of Featherweight Java. Master's thesis, Graduate School of Pennsylvania State University, 2005.

Rod M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.

Patrice Chalin. Improving JML: For a safer and more effective language. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proc. Formal Methods Europe, Pisa, Italy*, volume 2805 of *LNCS*, pages 440–461. Springer-Verlag, 2003.

Patrice Chalin. JML support for primitive arbitrary precision numeric types: Definition and semantics. *Journal of Object Technology*, 3(6):57–79, 2004.

Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Java Series. Addison-Wesley, June 2000.

Tony Clark and Jos Warmer, editors. *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer-Verlag, 2002.

Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Prc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 48–64, Vancouver, Canada, October 1998.

Robert L. Constable and Michael J. O'Donnell. *A Programming Logic, with an Introduction to the PL/CV Verifier*. Winthrop Publishers, 1978.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada*, volume 41, 6 of *ACM SIGPLAN Notices*, pages 415–426. ACM Press, June 2006.

Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

James O. Coplien. *Software Patterns*. SIGS Management Briefings. SIGS, New York, 1996.

John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: A rational module system for Java and its applications. In *Proc. 18th annual ACM SIGPLAN conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 241–254, Anaheim, California, USA, 2003. ACM Press. doi: http://doi.acm.org/10.1145/949305.949326.

Hans-Joachim Daniels. Multilingual syntax editing for software specifications. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2005.

Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer-Verlag, 2005.

Ewen Denney and Bernd Fischer. Software certification and software certificate management systems. In *Proc. Software Certificate Management Workshop (SoftCeMent), Long Beach, USA*, pages 1–6, 2005.

John Derrick and Eerke Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology. Springer-Verlag, 2001.

David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report #1998-159, Compaq Systems Research Center, Palo Alto, USA, December 1998.

Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, March 1998. ACM Press.

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. 21st International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press, ACM Press, 1999.

Hans-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical logic*. Undergraduate texts in mathematics. Springer Verlag, 1984.

Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, second edition, 2000.

Christian Engel. A Translation from JML to JavaDL. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, February 2005.

European Space Agency. Ariane 501 inquiry board report, July 1996.

Azadeh Farzan, Feng Chen, José Meseguer, and Grigore 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.

Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, second edition, 1996.

Melvin C. Fitting and Richard L. Mendelsohn. *First-Order Modal Logic*. Kluwer Academic Publishers, 1999.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.

Jean Gallier. *Logic for Computer Science.* Harper & Row Publisher, 1986. Revised online version from 2003 available from author's web page at `http://www.cis.upenn.edu/~jean/gbooks/logic.html`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading/MA, 1995.

Tobias Gedell and Reiner Hähnle. Automating verification of loops by parallelization. In Miki Herrmann, editor, *Proc. Intl. Conf. on Logic for Programming Artificial Intelligence and Reasoning, Pnhom Penh, Cambodia*, LNCS. Springer-Verlag, October 2006.

Robert Geisler, Marcus Klar, and Felix Cornelius. InterACT: An interactive theorem prover for algebraic specifications. In Martin Wirsing and Maurice Nivat, editors, *Proc. 5th International Conference on Algebraic Methodology and Software Technology (AMAST), Munich, Germany*, volume 1101 of *LNCS*, pages 563–566. Springer-Verlag, 1996.

Martin Giese. Incremental closure of free variable tableaux. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proc. Intl. Joint Conf. on Automated Reasoning IJCAR, Siena, Italy*, volume 2083 of *LNCS*, pages 545–560. Springer-Verlag, 2001.

Martin Giese. Taclets and the KeY prover. In David Aspinall and Cristoph Lüth, editors, *Proc. User Interfaces for Theorem Provers Workshop, Rome, Italy*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

Martin Giese. A calculus for type predicates and type coercion. In Bernhard Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, Tableaux 2005*, volume 3702 of *LNAI*, pages 123–137. Springer, 2005.

Martin Giese and Daniel Larsson. Simplifying transformations of OCL constraints. In L. Briand, editor, *8th Intl. Conf. on Model Driven Engineering Languages and Systems*, volume 3713 of *LNCS*, pages 309–323. Springer-Verlag, 2005.

Christoph Gladisch. Verification of C with KeY. Diplomarbeit, Department of Computer Science, University of Koblenz-Landau, 2006.

Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38: 173–198, 1931.

Martin Gogolla and Mark Richters. Expressing UML class diagrams properties with OCL. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*, pages 85–114. Springer-Verlag, 2002.

Joseph Goguen and Jose Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

Reuben Louis Goodstein. On the restricted ordinal theorem. *Journal of Symbolic Logic*, 9(2):33–41, 1944.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, 2nd edition, 2000.

Jean Goubault-Larrecq and Ian Mackie. *Proof Theory and Automated Deduction*, volume 6 of *Applied Logic Series.* Kluwer Academic Publishers, May 1997.

David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 366–373. Springer-Verlag, New York, NY, 1995.

Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

Elmar Habermalz. Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe, 2000a.

Elmar Habermalz. *Ein dynamisches automatisierbares interaktives Kalkül für schematische theoriespezifische Regeln.* PhD thesis, Universität Karlsruhe, 2000b.

Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IPGL*, 13(4):415–433, July 2005.

Reiner Hähnle and Wojciech Mostowski. Verification of safety properties in the presence of transactions. In Gilles Barthe and Marieke Huisman, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.

Reiner Hähnle and Peter H. Schmitt. The liberalized δ-rule in free variable semantic tableaux. *Journal of Automated Reasoning*, 13(2):211–222, October 1994.

Reiner Hähnle and Angela Wallenburg. Using a software testing technique to improve theorem proving. In Alex Petrenko and Andreas Ulrich, editors, *Post Conference Proceedings, 3rd International Workshop on Formal Approaches to Testing of Software (FATES), Montréal, Canada*, LNCS. Springer-Verlag, 2003.

Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An authoring tool for informal and formal requirements specifications. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering (FASE), Grenoble, France*, volume 2306 of *LNCS*, pages 233–248. Springer-Verlag, 2002.

David Harel. *First-Order Dynamic Logic.* Springer, 1979.

David 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.

David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* MIT Press, 2000.

John Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS 1869, pages 234–251. Springer, 2000.

John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs), Nice, France*, LNCS 1690, pages 113–130. Springer, 1999.

Jifeng He, Tony Hoare, and Jeff W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming*, volume 213 of *LNCS*, pages 187–196. Springer, 1986.

Maritta Heisel, Wolfgang Reif, and Werner Stephan. Program verification by symbolic execution and induction. In K. Morik, editor, *Proc. 11th German Workshop on Artificial Intelligence*, volume 152 of *Informatik Fachberichte*. Springer-Verlag, 1987.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

Tony Hoare. The ideal of verified software. In Thomas Ball and Robert B. Jones, editors, *Proc. Computer Aided Verification, 18th International Conference (CAV), Seattle, WA, USA*, volume 4144 of *LNCS*, pages 5–16. Springer-Verlag, 2006. URL http://dx.doi.org/10.1007/11817963_4.

John Hogg. Islands: Aliasing protection in object-oriented languages. In Andreas Paepcke, editor, *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 271–285. ACM Press, 1991.

Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003.

Engelbert Hubbers and Erik Poll. Reasoning about card tears and transactions in Java Card. In Michel Wermelinger and Tiziana Margaria, editors, *Proc. Fundamental Approaches to Software Engineering (FASE), Barcelona, Spain*, volume 2984 of *LNCS*, pages 114–128. Springer-Verlag, 2004a.

Engelbert Hubbers and Erik Poll. Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. Dept. of Computer Science NIII-R0438, Radboud University Nijmegen, 2004b.

Thierry Hubert and Claude Marché. A case study of C source code verification: the Schorr-Waite algorithm. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Proc. Third IEEE International Conference on Software Engineering and Formal Methods (SEFM), Koblenz, Germany*, pages 190–199. IEEE Computer Society, 2005.

Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, The Netherlands, 2001.

Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In T. S. E. Maibaum, editor, *Proc. Fundamental Approaches to Software Engineering, Third International Conference, (FASE), Berlin, Germany*, volume 1783 of *LNCS*, pages 208–221, 2000.

Michael Huth and Mark Ryan. *Logic in Computer Science.* Cambridge University Press, second edition, 2004.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

Bart Jacobs. Java's Integral Types in PVS. In E. Najim, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *LNCS*, pages 1–15. Springer, 2003.

Bart Jacobs and Erik Poll. A logic for the Java Modeling Language. In Heinrich Hußmann, editor, *Proc. Fundamental Approaches to Software Engineering, 4th International Conference (FASE), Genova, Italy*, volume 2029 of *LNCS*, pages 284–299. Springer-Verlag, 2001.

Bart Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. In *Software Security – Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4–6, 2003. Revised Papers*, volume 3233 of *LNCS*, pages 134–153. Springer, 2003.

Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java Program Verification Challenges. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *LNCS*, pages 202–219. Springer, Berlin, 2003.

Bart Jacobs, Claude Marché, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. In Charles Rattray, Savitri Maharaj, and Carron Shankland, editors, *Proc. 10th Conf. on Algebraic Methodology and Software Technology (AMAST), Stirling, UK*, volume 3116 of *LNCS*, pages 241–257. Springer-Verlag, July 2004.

Janna Khegai, Bengt Nordström, and Aarne Ranta. Multilingual syntax editing in GF. In Alexander Gelbukh, editor, *Intelligent Text Processing and Computational Linguistics (CICLing-2003)*, volume 2588 of *LNCS*. Springer, 2003.

James C. King. *A program verifier.* PhD thesis, Carnegie-Mellon University, 1969.

James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bull. London. Math. Soc.*, 14:285–293, 1982.

Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28:619–695, 2006.

Thomas Kolbe and Christoph Walther. Reusing proofs. In Anthony G. Cohn, editor, *Proc. 11th European Conference on Artificial Intelligence, Amsterdam, The Netherlands*, pages 80–84. John Wiley and Sons, 1994.

Sascha Konrad, Laura A. Campbell, Betty H. C. Cheng, and Min Deng. A requirements patterns-driven approach to specify systems and check properties. In Thomas Ball and Sriram K. Rajamani, editors, *Proc. Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA*, volume 2648 of *LNCS*, pages 18–33. Springer-Verlag, 2003.

Dexter Kozen and Jerzy Tiuryn. Logics of programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. The MIT Press, 1990.

Daniel Larsson and Reiner Hähnle. Symbolic fault injection. Technical Report 2006-17, Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, 2006.

Daniel Larsson and Wojciech Mostowski. Specifying JAVA CARD API in OCL. In Peter H. Schmitt, editor, *OCL 2.0 Workshop at UML 2003*, volume 102C of *ENTCS*, pages 3–19. Elsevier, November 2004.

Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Iowa State University, Department of Computer Science, 2003. Revised June 2004.

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual*, August 2006. Draft revision 1.197.

K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005. doi: http://dx.doi.org/10.1016/j.ipl.2004.10.015.

Barbara Liskov and John Guttag. *Program development in Java: abstraction, specification, and object-oriented design*. Addison-Wesley, Reading, MA, USA, 2000.

Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

Maria Manzano. *Extensions of First Order Logic*, volume 19 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996. Chapter VI and VII on many-sorted logic.

Claude Marché and Nicolas Rousset. Verification of JAVA CARD applets behavior with respect to transactions and card tears. In *Proc. Software Engineering and Formal Methods (SEFM), Pune, India*. IEEE CS Press, 2006.

Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of Java/Java Card programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

Renaud Marlet and Cédric Mesnil. Demoney: A demonstrative electronic purse – Card specification. Technical Report SECSAFE-TL-007, Trusted Logic S.A., November 2002.

Renaud Marlet and Daniel Le Métayer. Security properties and JAVA CARD specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic S.A., August 2001.

Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In Franz Baader, editor, *Proc. 19th Intern. Conf. on Automated Deduction (CADE), Miami, FL, USA*, volume 2741 of *LNCS*, pages 121–135. Springer-Verlag, 2003.

Erica Melis and Axel Schairer. Similarities and reuse of proofs in formal software verification. In Barry Smyth and Pádraig Cunningham, editors, *Proc. European Workshop on Advances in Case-Based Reasoning (EW-CBR), Dublin, Ireland*, volume 1488 of *LNCS*, pages 76–78, 1998.

Erica Melis and Jon Whittle. Analogy in inductive theorem proving. *J. Autom. Reason.*, 22(2):117–147, 1999. doi: http://dx.doi.org/10.1023/A:1005936130801.

José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning, Second International Joint Conference, IJCAR 2004, Cork, Ireland, Proceedings*, volume 3097 of *LNCS*, pages 1–44. Springer, 2004.

Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10): 40–51, October 1992.

Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In Susanne Graf and Michael I. Schwartzbach, editors, *Proc. 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Berlin, Germany*, volume 1785 of *LNCS*, pages 63–77. Springer-Verlag, 2000.

Carroll Morgan. *Programming from specifications*. International series in computer science. Prentice-Hall, 1990.

Wojciech Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In Maura Cerioli, editor, *Proc. Fundamental Approaches to Software Engineering (FASE), Edinburgh*, volume 3442 of *LNCS*, pages 357–371. Springer-Verlag, April 2005.

Wojciech Mostowski. Formal reasoning about non-atomic JAVA CARD methods in Dynamic Logic. In Jayadev Misra, Tobias Nipkow, and Emil Sekerin-

ski, editors, *Proceedings, Formal Methods (FM) 2006, Hamilton, Ontario, Canada*, volume 4085 of *LNCS*, pages 444–459. Springer, August 2006.

Wojciech Mostowski. Rigorous development of Java Card applications. In T. Clarke, A. Evans, and K. Lano, editors, *Proceedings, Fourth Workshop on Rigorous Object-Oriented Methods, London, U.K.*, March 2002.

Peter Müller. *Modular specification and verification of object-oriented programs*. Springer-Verlag, 2002.

Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62 (3):253–286, October 2006.

Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

Stanislas Nanchen, Hubert Schmid, Peter H. Schmitt, and Robert F. Stärk. The ASMKeY prover. Technical Report 436, Department of Computer Science, ETH Zürich and Institute for Logic, Complexity and Deduction Systems, Universität Karlsruhe, 2003.

Anil Nerode and Richard A. Shore. *Logic for Applications*. Springer (second edition), 1979.

Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In H. Schwichtenberg and K. Spies, editors, *Proceedings, Marktobderdorf Summer School 2003*. IOS Press, 2003.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

OCL 2.0. *Object Constraint Language Specification, version 2.0.* Object Modeling Group, June 2005. OMG document formal/2006-05-01.

David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, June 1999.

Ola Olsson and Angela Wallenburg. Customised induction rules for proving correctness of imperative programs. In Bernhard Beckert and Bernhard Aichernig, editors, *Proceedings, Software Engineering and Formal Methods (SEFM), Koblenz, Germany*, pages 180–189. IEEE Press, 2005.

Frank Ortmeier, Wolfgang Reif, and Gerhard Schellhorn. Formal safety analysis of a radio-based railroad crossing using deductive cause-consequence analysis (DCCA). In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *Proc. 5th European Dependable Computing Conference, Budapest, Hungary*, volume 3463 of *LNCS*, pages 210–224. Springer-Verlag, 2005.

S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *LNCS*, pages 411–414. Springer, July/August 1996.

Jing Pan. A theorem proving approach to analysis of secure information flow using data abstraction. Master's thesis, Chalmers University of Technology and Göteborg University, 2005.

Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. Enforcing high-level security properties for applets. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kalam, editors, *IFIP 18th World Computer Congress 2004, Smart Card Research and Advanced Applications (CARDIS), Toulouse, France*. Kluwer, 2004.

Cees Pierik and Frank S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *Proc. Formal Methods for Open Object-Based Distributed Systems, 6th IFIP WG 6.1 International Conference (FMOODS), Paris, France*, volume 2884 of *LNCS*, pages 64–78. Springer-Verlag, 2003.

André Platzer. Using a program verification calculus for constructing specifications from implementations. Studienarbeit, Universität Karlsruhe, Fakultät für Informatik, 2004a.

André Platzer. An object-oriented dynamic logic with updates. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, September 2004b.

Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Proc. European Symposium on Programming (ESOP), Amsterdam, The Netherlands*, volume 1576 of *LNCS*, 1999.

Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the JAVA CARD API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Academic Publishers, 2000.

Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 17th Annual IEEE Symposium on Foundation of Computer Science, Houston, TX, USA*, pages 109–121. IEEE Computer Society, 1977.

Aarne Ranta. Grammatical Framework: A type-theoretical grammar formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

Wolfgang Reif and Kurt Stenzel. Reuse of proofs in software verification. In Rudrapatna K. Shyamasundar, editor, *Proc. Foundations of Software Technology and Theoretical Computer Science, Bombay, India*, volume 761 of *LNCS*, pages 284–293. Springer-Verlag, 1993.

Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science B.V., 2001.

Andreas Roth. Specification and verification of encapsulation in Java programs. In Martin Steffen and Gianluigi Zavattaro, editors, *Proc. Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 In-*

ternational Conference (FMOODS), Athens, Greece, volume 3535 of LNCS, pages 195–210. Springer-Verlag, 2005.

Andreas Roth. *Specification and Verification of Object-oriented Components.* PhD thesis, Fakultät für Informatik der Universität Karlsruhe, 2006.

Philipp Rümmer. Generating counterexamples for Java Dynamic logic. In Wolfgang Ahrendt, Peter Baumgartner, and Hans de Nivelle, editors, *Preliminary Proceedings of Workshop on Disproving at CADE 20*, pages 32–44, July 2005.

G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif. The Mondex challenge: Machine checked proofs for an electronic purse. Technical report 2006-2, Institut für Informatik, Universität Augsburg, Germany, 2006.

Steffen Schlager. Handling of Integer Arithmetic in the Verification of Java Programs. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, May 2002.

Herbert Schorr and William M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, 1967. doi: http://doi.acm.org/10.1145/363534.363554.

Yoav Shoham. What is the frame problem? In Frank M. Brown, editor, *The Frame Problem in Artificial Intelligence*, pages 5–21. Morgan Kaufmann Publishers: San Mateo, CA, 1987.

Kurt Stenzel. A formally verified calculus for full Java Card. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *Proc. Algebraic Methodology and Software Technology, AMAST 2004, Stirling, Scotland, UK*, volume 3116 of *LNCS*, pages 491–505. Springer-Verlag, 2004.

Kurt Stenzel. *Verification of Java Card Programs.* PhD thesis, Institut für Informatik, Universität Augsburg, Germany, July 2005.

Werner Stephan, Bruno Langenstein, Andreas Nonnengart, and Georg Rock. Verification support environment. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning, Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, volume 2605 of *LNCS*, pages 476–493. Springer-Verlag, 2005.

Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 500–504. Springer, 2002.

Code Conventions for the Java Programming Language. Sun Microsystems, Inc., 2003a. Available at `java.sun.com/docs/codeconv`.

Java Card 2.2.1 Application Programming Interface. Sun Microsystems, Inc., Santa Clara, California, USA, October 2003b.

Java Card 2.2.1 Runtime Environment Specification. Sun Microsystems, Inc., Santa Clara, California, USA, October 2003c.

Java Card 2.2.1 Virtual Machine Specification. Sun Microsystems, Inc., Santa Clara, California, USA, October 2003d.

Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In Michel Wermelinger and Harald Gall, editors, *Proc. 10th European Software Engineering Conference/13th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering, 2005, Lisbon, Portugal*, pages 253–262. ACM Press, 2005.

Kerry Trentelman. Proving correctness of Java Card DL taclets using Bali. In Bernhard Aichernig and Bernhard Beckert, editors, *Proc. 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM), Koblenz, Germany*, pages 160–169, 2005.

Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Genova, Italy*, volume 2031 of *LNCS*, pages 299–312, 2001.

David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001a.

David von Oheimb. Axiomatic semantics for Java^{light}. In Sophia Drosso pou lou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Proc. Formal Techniques for Java Programs, Workshop at ECOOP'00, Cannes, France*, 2000.

David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13:1173–1214, 2001b.

David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proc. Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark*, volume 2391 of *LNCS*, pages 89–105. Springer-Verlag, 2002.

Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, August 2003.

Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999a.

Jos Warmer and Anneke Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, March 1999b.

Benjamin Weiß. Proving encapsulation in KeY with the Universe type system. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2006.

Hongseok Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm, January 2001.

Ernst Zermelo. Beweis dass jede Menge wohlgeordnet werden kann. *Mathematische Annalen*, 59:514–516, 1904.

Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.

List of Symbols

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.