Integrating Symbolic Execution, Debugging and Verification

Integration von symbolischer Programmausführung, Debugging und Verifikation
Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte Dissertation
zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
vorgelegt von Martin Hentschel M.Sc. geboren in Walsrode, Deutschland

Tag der Einreichung: 21. Januar 2016
Tag der Prüfung: 8. März 2016

1. Referent: Prof. Dr. Reiner Hähnle
2. Referent: Prof. Dr. K. Rustan M. Leino

Erscheinungsort: Darmstadt
Erscheinungsjahr: 2016

Darmstädter Dissertation — D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Software Engineering

Integrating Symbolic Execution, Debugging and Verification
Integration von symbolischer Programmausführung, Debugging und Verifikation

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Martin Hentschel M.Sc. aus Walsrode, Deutschland

1. Referent: Prof. Dr. Reiner Hähnle
2. Referent: Prof. Dr. K. Rustan M. Leino

Tag der Einreichung: 21. Januar 2016
Tag der Prüfung: 8. März 2016

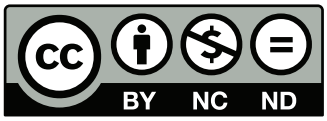
Erscheinungsort: Darmstadt
Erscheinungsjahr: 2016

Darmstädter Dissertation — D 17

Wissenschaftlicher Werdegang
Doktorand am Fachgebiet Software Engineering der Technischen Universität Darmstadt
von Januar 2012 bis März 2016

Studiengang Informatik: Master of Science (M.Sc.)
Carl-Friedrich-Gauß-Fakultät der Technischen Universität Braunschweig
von Oktober 2009 bis November 2011

Studiengang Informatik: Bachelor of Science (B.Sc.)
Fachhochschule Braunschweig/Wolfenbüttel
von März 2006 bis April 2009



Abstract

In modern software development, almost all activities are centered around an integrated development environment (IDE). Besides the main use cases to write, execute and debug source code, an IDE serves also as front-end for other tools involved in the development process such as a version control system or an application lifecycle management.

Independent from the applied development process, the techniques to ensure correct software are always the same. The general goal is to find defects as soon as possible, because the sooner a defect is found, the easier and cheaper it is to fix. In the first place, the programming language helps to prevent some kinds of defects. Once something is written, it is effective to review it not only to find defects, but also to increase its quality. Also tools which statically analyze the source code help to find defects automatically. In addition, testing is used to ensure that selected usage scenarios behave as expected. However, a test can only show the presence of a failure and not its absence. To ensure that a program is correct, it needs to be proven that the program complies to a formal specification describing the desired behavior. This is done by formal verification tools. Finally, whenever a failure is observed, debugging takes place to locate the defect.

This thesis extends the software development tool suite by an interactive debugger based on symbolic execution, a technique to explore all feasible execution paths up to a given depth simultaneously. Such a tool can not only be used for classical debugging activities, but also during code reviews or in order to present results of an analysis based on symbolic execution. The contribution is an extension of symbolic execution to explore the full program behavior even in presence of loops and recursive method calls. This is achieved by integrating specifications in form of loop invariants and methods contracts into a symbolic execution engine. How such a symbolic execution engine based on verification proofs can be realized is presented as well.

In addition, the presented Symbolic Execution Debugger (SED) makes the Eclipse platform ready for debuggers based on symbolic execution. Its functionality goes beyond that of traditional interactive debuggers. For instance, debugging can start directly at any method or statement and all program execution paths are explored simultaneously. To support program comprehension, program execution paths as well as intermediate states are visualized. By default, the SED comes with a symbolic execution engine implemented on top of the KeY verification system. Statistical evidence that the SED increases effectiveness of code reviews is gained from a controlled experiment.

Another novelty of the SED is that arbitrary verification proofs can be inspected. Whereas traditional user interfaces of verification tools present proof states in a mathematical fashion, the SED analyzes the full proof and presents different aspects of it using specialized views. A controlled experiment gives statistical evidence that proof understanding tasks are more effective using the SED by comparing its user interface with the original one of KeY.

The SED allows one to interact with the underlying prover by adapting code and specifications in an auto-active flavor, which creates the need to manage proofs directly within an IDE. A presented concept achieves this, by integrating a semi-automatic verification tool into an IDE. It includes several optimizations to reduce the overall proof time and can be realized without changing the verification tool. An optimal user experience is achieved only if all aspects of verification are directly supported within the IDE. Thus a thorough integration of KeY into Eclipse is presented, which for instance includes in addition to the proof management capabilities to edit JML specifications and to setup the needed infrastructure for verification with KeY.

Altogether, a platform for tools based on symbolic execution and related to verification is presented, which offers a seamless integration into an IDE and furthers a usage in combination. Furthermore, many aspects, like the way the SED presents proof attempts to users, help to reduce the barrier of using formal methods.

Zusammenfassung (Abstract in German)

Eine integrierte Entwicklungsumgebung (IDE) unterstützt nahezu alle Aspekte moderner Softwareentwicklung. Neben den klassischen Anwendungsfällen wie das Schreiben, Ausführen und Debuggen von Quellcode, fungiert sie auch als Schnittstelle zu allen anderen Werkzeugen, die im Entwicklungsprozess eingebunden sind. Dies sind z. B. Versionsverwaltungs- oder Anwendungsmanagementsysteme.

Unabhängig vom verwendeten Entwicklungsprozess sind die Techniken, um fehlerfreie Software zu entwickeln, immer die gleichen. Das Ziel ist es Fehler so früh wie möglich zu finden, denn je eher ein Fehler gefunden wird, desto einfacher und günstiger ist er zu beheben. An erster Stelle verhindern Programmiersprachen gewisse Arten von Fehlern. Sobald etwas geschrieben wurde, ist ein Review ein effektives Werkzeug, um Fehler zu finden und die Qualität zu erhöhen. Eine statische Analyse kann gewisse Fehler ebenfalls automatisch finden. Das ausgewählte Szenarien richtig funktionieren, wird durch Tests sichergestellt. Ein Test kann jedoch nur die Gegenwart und nicht die Abwesenheit eines Fehlers zeigen. Um sicherzustellen, dass ein Programm richtig ist, muss gezeigt werden, dass es einer formalen Spezifikation, welche das gewünschte Verhalten beschreibt, entspricht. Dies wird mittels formaler Verifikation erreicht. Sobald ein Fehler gefunden wurde, muss dessen Ursache schließlich durch Debuggen gefunden werden.

Diese Arbeit erweitert das Softwareentwicklungs-Portfolio um einen interaktiven Debugger basierend auf symbolischer Programmausführung, eine Technik die alle möglichen Programmausführungspfade bis zu einer gegebenen Tiefe zeitgleich erkundet. Der resultierende Debugger kann nicht nur für klassische Debugging-Aktivitäten verwendet werden, sondern auch für Reviews oder um Ergebnisse einer Analyse basierend auf symbolischer Programmausführung darzustellen. Im Rahmen dieser Arbeit wird die symbolische Programmausführung erweitert, um das gesamte Programmverhalten auch in Gegenwart von Schleifen und rekursiven Methodenaufrufen zu erkunden. Dies wird durch die Integration von Spezifikationen in Form von Schleifeninvarianten und Methodenverträgen erreicht. Ebenfalls wird gezeigt, wie solch eine symbolische Ausführungseinheit basierend auf Verifikations-Beweisen realisiert werden kann.

Zusätzlich wird der Symbolic Execution Debugger (SED) vorgestellt, welcher die Eclipse Plattform für Debugger basierend auf symbolischer Programmausführung vorbereitet. Die Funktionalität geht dabei über die traditioneller Debugger hinaus, z. B. kann das Debuggen direkt bei jeder beliebigen Methode oder Anweisung beginnen. Ebenso werden alle möglichen Programmausführungspfade zeitgleich erkundet. Um das Programmverstehen zu erhöhen, werden Ausführungspfade und Zustände visualisiert. Standardmäßig stellt der SED eine symbolische Ausführungseinheit basierend auf dem Verifikationssystem KeY zur Verfügung. Ein statistischer Nachweis, dass der SED die Effektivität von Reviews erhöht, wird durch ein kontrolliertes Experiment erbracht.

Eine weitere Neuheit des SED ist, das beliebige Verifikationsbeweise inspiziert werden können. Während traditionelle Benutzeroberflächen von Verifikationswerkzeugen einen Beweis aus mathematischer Sicht präsentieren, analysiert der SED den gesamten Beweis und präsentiert unterschiedliche Aspekte mittels verschiedener Sichten. Ein kontrolliertes Experiment weist durch einen Vergleich mit der originalen Benutzeroberfläche von KeY nach, dass das Beweisverstehen durch den SED erhöht wird.

Der SED ermöglicht mit dem darunterliegenden Beweiser durch Anpassung des Quellcodes und dessen Spezifikationen zu interagieren. Dies schafft das Bedürfnis, Beweise direkt innerhalb der IDE zu verwalten. Dazu wird ein Konzept präsentiert, welches ein semi-automatisches Verifikationswerkzeug in eine IDE integriert. Es beinhaltet mehrere Optimierungen, um die Zeit der Beweisführung zu reduzieren. Zusätzlich kann es ohne Anpassung des Verifikationswerkzeugs realisiert werden. Eine optimale Benutzererfahrung wird jedoch nur erreicht, wenn alle Aspekte der Verifikation direkt von der IDE unterstützt werden. Deshalb wird eine vollständige Integration von KeY in Eclipse vorgestellt, welche beispielsweise zusätzlich zum Beweismanagement auch das Schreiben von JML-Spezifikationen und das Bereitstellen der benötigten Infrastruktur zum Beweisen mit KeY unterstützt.

Insgesamt wird eine Plattform für Werkzeuge basierend auf symbolischer Programmausführung und verwandt zur Verifikation vorgestellt, welche eine nahtlose Integration in eine IDE und eine gemeinsame Nutzung fördert. Viele Aspekte, wie z. B. die Art wie Beweise im SED dem Benutzer dargestellt werden, helfen die Barrieren beim Einsatz formaler Methoden zu reduzieren.

Acknowledgments

Many people made this thesis possible. I would like to spend some words to thank all of them:

Reiner Hähnle was not only a great supervisor, but also a good friend. As supervisor he (i) was always available, (ii) helped to focus on the right things and to take the right decisions and (iii) motivated to publish results at good conferences. As friend he taught me to enjoy good food and wine.

My second reviewer K. Rustan M. Leino. His comments helped a lot to improve this thesis and the presented tools.

Richard Bubel, a good friend and my first contact person for everything related to KeY and work in general. Even he is too modest to admit it, but his support was the most important one for this thesis.

All group members and friends of the Software Engineering group of TU Darmstadt. We had a really good time not only at work, but also in our leisure time. Many ideas were born during discussions and later realized as part of this thesis. This is also true for all my coauthors and the people of the KeY community.

The students who helped in developing the presented tools:

- Stefan Käsdorf did the first implementation of KeY Resources as part of his Bachelor's thesis and continued the work for many semesters as research assistant.
- Marco Drebing implemented breakpoints in the Symbolic Execution Debugger (SED) and the KeYIDE as part of his Bachelor's thesis and completed the work as research assistant.
- Martin Möller realized the grouping of symbolic execution tree nodes in the SED as part of his Bachelor's thesis.
- Marco Drebing, Stefan Käsdorf, Niklas Bunzel and Christoph Schneider did the first implementation of the KeYIDE as part of their Bachelor-Praktikum[1].
- Faris Abraha, Leon Böck, Dominik Helm and Lukas Sommer realized the integration of JPF–SE into the SED as part of their Bachelor-Praktikum[1].
- David Giessing, Thomas Glaser and Moritz Lichter implemented the first version of JML Editing as part of their Bachelor-Praktikum[1].
- Christopher Beckmann, Robert Heimbach, Timm Christian Lippert and Maksim Melnik extended the functionality of JML Editing as part of their Bachelor-Praktikum[1].
- Anna Marie Filighera, Leonard Götz, Viktor Nikolas Pfanschilling and Seena Vellaramkalayil extended the functionality of the KeYIDE and made the SED ready for interactive verification as part of their Bachelor-Praktikum[1].
- Sven Bräutigam and Niklas Bunzel improved the KeYIDE as research assistant.
- Florian Kaffenberger implemented a first prototype of Stubby as part of his study project at the Hochschule Darmstadt (h_da).

All participants of the evaluations for their valuable time and feedback.

Last but not least, I would like to thank my wife Stephanie Hentschel and my family for their support during my time in Darmstadt.

[1] The Bachelor-Praktikum is a mandatory lecture at the TU Darmstadt in which a group of students realizes a project.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Definitions

List of Propositions

1 Introduction

In the literature, a number of software development processes are suggested to develop software with a high quality. But in practice, there is not *the* golden way to achieve this goal. Instead, each organization and each project implements its own development process. Without customizable tool support, this would not be possible. Typically used tools are (i) an integrated development environment (IDE) in which the source code is written, compiled and debugged, (ii) a version control system to manage different versions of the source code and related documents, and (iii) an application lifecycle management to keep track of requirements, specifications, found bugs, etc. The strength of an IDE is, that it can serve as front-end for all other involved tools in addition to the mentioned use cases.

Independent from the applied development process, the techniques to ensure correct software are always the same. The general goal is to find defects as soon as possible, because the sooner a defect is found, the easier and cheaper it is to fix (see [127]). In the first place, the programming language helps to prevent some kinds of defects. A typed language such as Java for instance prevents one from type incompatible operations, e.g. dividing an integer by a string.

Letting somebody else look at what you have written in a review helps not only to find defects, but also to increase code quality (e.g. design or readability) and to share knowledge. Although the effectiveness of a review like an inspection [47, 48] is well known, tool support is rare. Systems such as Gerrit[1] help to track the discussion during the review, but do not target the reviewing activity directly. In addition to a human review, tools such as FindBugs [14] statically analyze the source code to find common defects. Usually there is no guarantee that all defects are reported or that a reported defect is indeed a defect.

Once a failure is observed, the defect in the source code needs to be found. This activity is called debugging. The tool of choice is a debugger, which allows one to control execution, to inspect the current state and thus to comprehend each performed step. A systematic introduction into debugging and how to trace a failure back to the defect is given by Zeller [137]. However, the challenge to find suitable input values resulting in an execution exhibiting the failure remains.

Testing is widely used in practice. A test case, ideally automatically executable, describes one scenario under which the software is tested. Consequently, a test can only show the presence of failures but not their absence (see [42]). To ensure that a program is correct, it needs to be proven that the program complies to a formal specification describing the desired behavior. This is done by formal verification tools such as KeY [5]. In practice, functional correctness is rarely proven not only because it is difficult to obtain the needed specifications, but also because in particular interactive verification tools integrate purely into mainstream IDEs. Complicating is also the fact that in order to use a formal verification tool detailed knowledge about the logic and the implemented proof search strategy is needed.

Symbolic execution [28, 25, 84, 85] is a technique used by test case generation and formal verification tools to explore systematically *all* possible execution paths of a program for *all* possible input data. The result is a so called *symbolic execution tree* representing the full program behavior up to a given depth.

From the beginning, debugging (e.g. [85]) is one of the use cases of symbolic execution. The inspected artefacts and the degree of user control during the debugging activity varies between the different use cases. In the context of verification, a counterexample violating a verified property is usually presented to the user. Depending on the verification technique, user interaction might be completely avoided or is sometimes needed to guide the prover to find a proof. Test case generation tools can generate test cases fully automatically and once a test fails, a traditional debugger can be used to comprehend program execution during the test. But symbolic execution can be also used to realize an interactive debugger with functionality known from traditional debuggers. This includes step-wise execution, the use of breakpoints to suspend execution at specified points of interest and the inspection of the memory when execution is suspended. Such a system was realized the first time by King [85] in the 70s. Years later,

[1] www.gerritcodereview.com

Hähnle et al. [63] took up the idea in 2010 and realized a prototypical, so called, *visual symbolic state debugger* based on the KeY system and Eclipse. Such an interactive debugger based on symbolic execution has not only the potential to improve the debugging and reviewing process, but also to decrease the barrier of using formal methods.

KeY is a semi-automatic formal verification tool which allows one to verify the correctness of Java programs specified by JML annotations. The program to verify is symbolically executed by rules part of the underlying calculus. The strength of KeY is that it allows one to reason about Java programs without any abstraction. A witness for this is the recent work by De Gouw et al. [39] who found a defect in Java's default sorting algorithm and verified its absence in the fixed version with KeY. The KeY project started in 1998 with the goal to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. KeY is released as stand-alone application in which SMT solvers can be plugged in. However, before this thesis KeY was not integrated into mainstream IDEs like Eclipse[2] which is an obstacle for its use in practice.

In the following, Section 1.1 describes the state of the art before the contributions of this thesis are listed in Section 1.2. An overview of the author's publications and their relation to this thesis is given in Section 1.3. Finally, Section 1.4 presents the structure of this thesis.

1.1 State of the Art

Modern software verification targets different scenarios with different methods and related tool support. An overview of the state of the art and trends are discussed by Beckert and Hähnle [21].

Functional correctness properties are specified in a declarative manner, usually by using a contract based specification language such as JML [93] for Java programs, Spec# [15] for C# programs, or ACSL [17] and the VCC [34] language for C programs. To prove that a program complies with its specification, it needs to be shown that all possible program executions (in particular for all input values and initial states) behave as specified. Tool support uses deductive verification, meaning that a logic is used for reasoning. The way how possible program executions are explored differs between the tools:

- KeY [5], KIV [110] and VeriFast [74] for instance use symbolic execution, realized by calculus rules interpreting statements according to the semantics of the programming language.
- Dafny [94] and Why [49] for instance use weakest precondition reasoning to generate first order verification conditions which can be solved by external SMT solvers.
- Alternatively, a higher order logic like Isabelle/HOL [104] can be used to model syntax and semantics of the programming language and to express correctness as mathematical theorem.

As verification tools became powerful enough to deal with real programs, Baumann et al. [18] found out that writing specifications is the bottleneck in functional verification. To deal with large systems, each component (e.g. a method) is verified in isolation. This requires to provide so called *auxiliary specifications* describing the functional behavior of involved components (e.g. called methods) and loop invariants to deal with unbounded loops. The size of such specifications is often a multiple of the implementation itself.

Other application areas of deductive verification like the verification of non-functional properties (e.g. resource consumption) or relational properties (e.g. non-interference) require less or avoid auxiliary specifications completely. Also bounded verification avoids auxiliary specifications, because the program behavior is only explored up to a given bound, which is still enough to observe failures within the bound.

Model checking is related to the verification of safety and liveness properties. The program is modeled as a transition system which should never enter critical states (safety) and at some point reach some desired states (liveness). Specifications are usually written in a temporal logic and interpreted according

[2] An Eclipse plug-in which allowed one to start KeY from Eclipse was available for early KeY releases.

to an execution history. To deal with large models, symbolic execution can be used to reduce the state space. This is implemented for instance by the Java PathFinder [129, 10] or Bogor [113, 41].

Whereas verification proves the correctness of a program, testing tries to find failures in the target program and in addition in its execution environment. Test cases can be generated automatically according to a coverage criterion, such as that each statement is executed at least once. Symbolic execution is typically used to explore feasible execution paths until the coverage criterion is fulfilled. This is realized for instance by Pex [123].

A new application area of symbolic execution is program transformation and compilation. Ji et al. [80] use symbolic execution to explore all possible execution paths and then iterate backward over the symbolic execution tree to generate an optimized program step by step.

In modern software development, an integrated development environment (IDE) such as Eclipse[3], IntelliJ IDEA[4], Netbeans[5] or Visual Studio[6] is used to cover all aspects of software development. This includes besides classical functionalities like editing, compilation and debugging to be a front-end for all tools and systems used in the development process. Many verification tools followed the trend and offer an IDE integration. For interactive tools, the IDE integration usually supports editing of related documents and a user interface specific to the verification method. Fully automatic verification tools like Dafny can be integrated like a compiler and report about failed proofs directly while the user writes the source code.

1.2 Contributions

In the last years, KeY became a powerful verification tool which can deal with real world programs. Most of the effort was put in the theory behind and in the development of a user interface, which is optimized for experienced users performing a single proof. However, real world programs are not verified once by a single proof. Several proofs are required to ensure overall system correctness and many of them need to be redone whenever code or specifications change. The contribution of this thesis is a concept to integrate an interactive verification tool into an IDE and its implementation integrating KeY into Eclipse. It supports all aspects of verification in a development process where the source code is modified several times.

In addition, the Symbolic Execution Debugger (SED), a platform for interactive symbolic execution is presented. It allows a user to directly debug any method or statement(s) without setting up an initial state. Being based on symbolic execution, all feasible execution paths are explored at once and the resulting symbolic execution tree is visualized. This turns symbolic execution into a powerful assistant for code reviews. In addition, the SED can be used like a traditional debugger for bug finding or to find execution paths fulfilling a given criteria. Auxiliary specifications are not needed at all, but if available, they can be debugged as well and be applied to guarantee finite execution paths. The SED is a complete rewrite of [5] to decouple it from the KeY system with significantly extended functionality.

Different symbolic execution engines can be easily integrated into the SED to control symbolic execution and to present results of the analysis using it. By default, the SED comes with KeY as symbolic execution engine and supports also the inspection of proof attempts. Compared to KeY, a proof attempt is presented from the developer's perspective and can be understood without any knowledge about KeY and formal methods in general.

The contributions of this thesis are summarized in Table 1.1. All tools presented in this thesis can be added to an existing Eclipse installation via an update-site. The supported Eclipse versions and the concrete update-site URLs are available on the KeY website (www.key-project.org/download).

[3] www.eclipse.org
[4] www.jetbrains.com/idea
[5] netbeans.org
[6] www.visualstudio.com

Contribution		Kind
1.	An extension of symbolic execution to support specifications in the form of loop invariants and method contracts. This ensures finite execution paths in presence of loops and recursive methods.	Conceptual
2.	A concept of how to build a symbolic execution engine on top of a proof.	
3.	A technique to trace truth statuses evaluated during a proof.	
4.	Slicing of proof trees and symbolic execution trees.	
5.	A lightweight approach to integrate an interactive verification tool into an IDE. This allows one to manage proofs as part of an IDE project and to give users feedback as soon as possible.	
6.	A symbolic execution engine realized on top of the KeY system, called KeY's Symbolic Execution Engine. It realizes contributions 1 to 4.	Implementation
7.	The Symbolic Execution Debugger (SED), an extension of the Eclipse debug platform for interactive symbolic execution. It comes with an integration of KeY as symbolic execution engine (contribution 6) which allows one in addition to interactive symbolic execution the inspection of proof attempts.	
8.	KeY Resources, an implementation of the lightweight proof management concept (contribution 5) integrating KeY into Eclipse.	
9.	Several Eclipse extensions for an optimal user experience for verification with KeY. This includes, for instance, support for writing JML specifications.	
10.	An experimental evaluation which compares the tools KeY and SED with respect to an inspection of proof attempts.	Evaluation
11.	Another experimental evaluation comparing code reviews with and without the SED.	
12.	An evaluation of the optimizations suggested by the lightweight proof management concept (contribution 5) to reduce the overall proof time.	

Table 1.1.: Contributions

1.3 Overview of Publications

Table 1.2 lists the author's publications and puts them into relation to this thesis. A contribution declared as "Lead author" means that the author of this thesis did most of the theoretical investigation, implementation and writing. The author of this thesis helped in writing of [5, 70, Chapter 7] and implemented the tool MonKeY used by [120, 121]. In addition, the author of this thesis helped with tool support and verification issues of [122].

Publication	Contribution	Relation to Thesis
[67]	Lead author	Chapter 3, Section 10.1 and Section 11.1 are an updated version of the original publication
[5, Chapter 11]	Lead author	Chapter 4, Chapter 6 and Section 11.2 are an updated and extended version of the original publication
[68]	Lead author	Chapter 7, Section 10.6 and Section 11.3 are an updated and extended version of the original publication
[66]	Lead author	Presents the SED similar as Chapter 6; Section 6.2 and Section 10.5 are an updated and extended version of the original publication
[4]	Lead author of Section 7	Presents the SED in short
[5, Chapter 16]	Lead author of Section 16.6	Presents KeY Resources in short
[5, Chapter 7] and [70]	Coauthor	A detailed JML tutorial
[120] and [121]	Coauthor	MonKeY, a predecessor of KeY Resources is used
[122]	Coauthor	KeY and its Eclipse integration are used for verification

Table 1.2.: Publications

1.4 Structure of this Thesis

The background of this thesis is discussed in Chapter 2. Then, symbolic execution using specifications is presented in Chapter 3. A symbolic execution engine based on the KeY system which supports specifications is discussed in Section 4. The Symbolic Execution Debugger (SED), a platform for interactive symbolic execution, is presented in Chapter 6. KeY's integration into Eclipse consists of a novel proof management (Chapter 7) and several additional features like editing facilities for JML (Chapter 8). The evaluation of the SED and the proof management are presented in Chapter 9. Work related to this thesis is discussed in Chapter 10. Finally, the conclusion and future work is presented in Chapter 11.

2 Background

This chapter introduces the background of this thesis. First, JML, the language to specify Java programs, is introduced in Section 2.1. Second, the formal verification tool KeY which allows one to verify the correctness of Java programs annotated with JML specifications is presented in Section 2.2. Third, Java DL, the logic used by KeY, is explained in Section 2.3. Finally, the Eclipse platform is discussed in Section 2.4.

2.1 JML

The Java Modeling Language (JML) [92, 93] is a specification language used to specify the expected behavior of Java modules following the *design-by-contract* paradigm [101]. In the following, the features of JML relevant for this thesis are introduced in tutorial style. To be more precise, a subset of JML with KeY specific extensions is presented. A complete tutorial can be found in [5].

First, Section 2.1.1 introduces the comments in which JML specifications are placed. Second, Section 2.1.2 introduces invariants used to limit the possible state space of instances. Third, method contracts are introduced in Section 2.1.3. Finally, loop invariants used to guide verification tools are discussed in Section 2.1.4.

2.1.1 JML Comments

JML specifications are placed as special comments within the Java source code. Such comments have to start with the `@` character. The shape of JML comments used in this thesis are shown in Listing 2.1 (multi-line) and Listing 2.2 (single-line).

```
1 /*@ ...
2   @ ...
3   @ ...
4   @*/
```

Listing 2.1: Multi-line JML Comment

```
1 //@ ...
```

Listing 2.2: Single-line JML Comment

2.1.2 Invariants

An invariant is a property used to limit the possible state space of instances. All constructors of a class have to establish the invariant whereas all methods have to preserve it. It can only be temporarily violated within a method execution.

In JML keyword **`invariant`** followed by a JML expression defines an invariant. A JML expression is basically a normal boolean Java expression with some JML specific extensions.

Consider for instance class `Course` in Listing 2.3. The state space of instance field `credits` is limited to positive values (line 2). Additionally, **`non_null`** specifies that `name` is never **`null`** whereas **`nullable`** allows `description` to be **`null`**.

```
1 public class Course {
2   //@ invariant credits > 0;
3   private final int credits;
4
```

```
5    private final /*@ non_null @*/ String name;
6
7    private final /*@ nullable @*/ String description;
8 }
```

Listing 2.3: Invariant of Class `Course`

JML expressions with additional JML features can be seen in Listing 2.4. First, the state space of instance field `passedCoursesSize` is limited in line 2 using a normal Java expression. It specifies that `passedCoursesSize` is greater or equal to zero and that `passedCoursesSize` is less or equal than `passedCourses.length`. Second, invariants are given for instance field `passedCourses`. The invariant at line 5 states that `passedCourses` is never **`null`**. The next invariant at line 6 ensures that `passedCourses` points to an instance of exact type `Course[]` and not to one with a subtype of `Course[]`. Then, line 7 forces with help of the quantifier (**`\forall`**) that all array indices below `passedCoursesSize` are not **`null`**. Finally, line 8 forces that other array indices equal or greater than `passedCoursesSize` are **`null`**. The keyword **`non_null`** is not used, because it forces also the values at all array indices to be **`non_null`**.

```
1  public class Student {
2    //@ invariant passedCoursesSize >= 0 && passedCoursesSize <= passedCourses.length;
3    private int passedCoursesSize;
4
5    /*@ invariant passedCourses != null;
6      @ invariant \typeof(passedCourses) == \type(Course[]);
7      @ invariant (\forall int i; i >= 0 && i < passedCoursesSize; passedCourses[i] != null);
8      @ invariant (\forall int i; i >= passedCoursesSize && i < passedCourses.length;
9      @                     passedCourses[i] == null);
10     @*/
11   private /*@ nullable @*/ Course[] passedCourses = new Course[100];
12 }
```

Listing 2.4: Invariant of Class `Student`

2.1.3 Method Contracts

A method contract specifies the expected behavior of a method in terms of pre- and postconditions. Assuming that the precondition is fulfilled when the method is called, the method guarantees that the postcondition is established when it returns. In JML the keyword **`normal_behavior`** is used to specify that a method must terminate normally without a thrown exception. Keyword **`requires`** followed by a JML expression defines a precondition whereas **`ensures`** also followed by a JML expression defines a postcondition. Please observe that the precondition is evaluated in the method call state whereas the postcondition is evaluated in the method return state.

The last part of a method contract is the **`assignable`** clause. It lists all locations the method is allowed to change. Keyword **`\everything`** expresses that the method is allowed to change all locations whereas **`\nothing`** forbids any location to be changed (but allows still that new objects are created). In KeY, **`\strictly_nothing`** can be used to express that no objects will be created and that nothing is allowed to be changed.

Consider for instance the contract in Listing 2.5. Lines 4 to 10 specify that the constructor needs to be called in a state where `credits > 0` (**`requires`**) holds. The caller has then the guarantee that all instance fields are updated (**`ensures`**), that no exception is thrown (**`normal_behavior`**) and that nothing else has changed (**`assignable`**).

The method contract in lines 19 to 21 is applicable in any state (default of **`requires`** is **`true`**) and guarantees that no exception is thrown, that the returned value (**`\result`**) is the value of instance field `credits` and that nothing has been changed. Please observe that **`pure`** and **`strictly_pure`** are a short form for **`assignable \nothing`** and **`assignable \strictly_nothing`**, respectively.

```
1  public class Course {
2     // ... Fields and Invariant from Listing 2.3
3
4     /*@ normal_behavior
5       @ requires credits > 0;
6       @ ensures this.credits == credits;
7       @ ensures this.name == name;
8       @ ensures this.description == description;
9       @ assignable this.credits, this.name, this.description;
10      @*/
11    public Course(int credits,
12                  /*@ non_null @*/ String name,
13                  /*@ nullable @*/ String description) {
14       this.credits = credits;
15       this.name = name;
16       this.description = description;
17    }
18
19    /*@ normal_behavior
20      @ ensures \result == credits;
21      @*/
22    public /*@ strictly_pure @*/ int getCredits() {
23       return credits;
24    }
25 }
```

Listing 2.5: Contracts of Class `Course`

To specify exceptional behavior, the JML keyword **exceptional_behavior** is used. The exceptions allowed to be thrown are listed in the **signals_only** clause. Additionally, the established poststate is specified for each possible exception by a **signals** clause followed by a JML expression. Different specification cases are separated by **also**. Keyword **behavior** is used for a general specification case not limited to normal or exceptional termination.

Consider for instance the two specification cases of method `addCourse` in Listing 2.6. The specification case for normal termination in lines 4 to 9 is applicable, if `passedCoursesSize` is less than `passedCourses.length` and if the invariant of `course` holds (**\invariant_for**). Its postcondition guarantees that `course` is assigned to the array index specified by the value of `passedCoursesSize` at method call time and that `passedCoursesSize` is increased by one. This is achieved with help of the **\old** keyword which contains an expression evaluated in the method call state.

The exceptional specification case in lines 11 to 15 states that if the end of array `passedCourses` is reached, a `RuntimeException` will be thrown.

The method contract of `countCredits` in lines 27 to 30 uses **\sum** to compute the sum of course credits contained in `passedCourses`.

```
1  public class Student {
2     // ... Fields and Invariant from Listing 2.4
3
4     /*@ normal_behavior
5       @ requires passedCoursesSize < passedCourses.length;
6       @ requires \invariant_for(course);
7       @ ensures passedCourses[\old(passedCoursesSize)] == course;
8       @ ensures passedCoursesSize == \old(passedCoursesSize) + 1;
9       @ assignable passedCoursesSize, passedCourses[passedCoursesSize];
10      @ also
11      @ exceptional_behavior
12      @ requires passedCoursesSize >= passedCourses.length;
```

```
13      @ signals_only RuntimeException;
14      @ signals (RuntimeException e) true;
15      @ assignable \nothing;
16      @*/
17    public void addCourse(/*@ non_null @*/ Course course) {
18       if (passedCoursesSize < passedCourses.length) {
19          passedCourses[passedCoursesSize] = course;
20          passedCoursesSize++;
21       }
22       else {
23          throw new RuntimeException();
24       }
25    }
26
27    /*@ normal_behavior
28      @ ensures \result == (\sum int i; 0 <= i && i < passedCoursesSize;
29      @                           passedCourses[i].getCredits());
30      @*/
31    public /*@ strictly_pure @*/ int countCredits() {
32       int credits = 0;
33       for (int i = 0; i < passedCoursesSize; i++) {
34          credits += passedCourses[i].getCredits();
35       }
36       return credits;
37    }
38 }
```

Listing 2.6: Contracts of Class `Student`

Declaring a method as **helper** also allows one to call the method in a state in which the class invariant does not hold. Such a **helper** method also does not need to reestablish the invariant before it is returned.

2.1.4 Loop Specifications

A loop invariant is a property which needs to hold before a loop is entered and which is preserved by each loop iteration (loop guard and loop body). Thus, it also holds after the loop. In JML, a loop specification consists of the following parts:

- Keyword **loop_invariant** is used to specify a loop invariant.
- Keyword **decreasing** specifies a value which is always positive and strictly decreased in each loop iteration. It is used to prove termination of the loop. The decreasing clause is also named variant.
- Keyword **assignable** limits the locations that may be changed by the loop.

Consider for instance the loop specification of the **for**-loop in Listing 2.7. The first loop invariant clause at line 3 limits the range of the index variable `i`. The second loop invariant clause at line 4 says that `credits` is the sum of all previously visited array indices. The **decreasing** clause is given in line 5. In KeY, local variables are automatically added to the assignable clause of a loop invariant. Consequently, only fields need to be listed. Here, nothing except the local variables is allowed to be changed (line 6).

```
1 public /*@ strictly_pure @*/ int countCredits() {
2    int credits = 0;
3    /*@ loop_invariant i >= 0 && i <= passedCoursesSize;
4      @ loop_invariant credits == (\sum int j; 0 <= j && j < i; passedCourses[j].getCredits());
5      @ decreasing passedCoursesSize - i;
```

```
6       @ assignable \strictly_nothing;
7       @*/
8     for (int i = 0; i < passedCoursesSize; i++) {
9        credits += passedCourses[i].getCredits();
10    }
11    return credits;
12 }
```

Listing 2.7: Loop Specification of a `for`-Loop

2.2 KeY

The KeY system [5] is a formal verification tool which allows one to verify the correctness of Java programs specified by JML annotations. In recent years, also other applications like information flow analysis [116, 115, 26, 126] or test case generation [46, 19, 56] were realized with KeY. This results in the KeY Framework as shown in Figure 2.1.

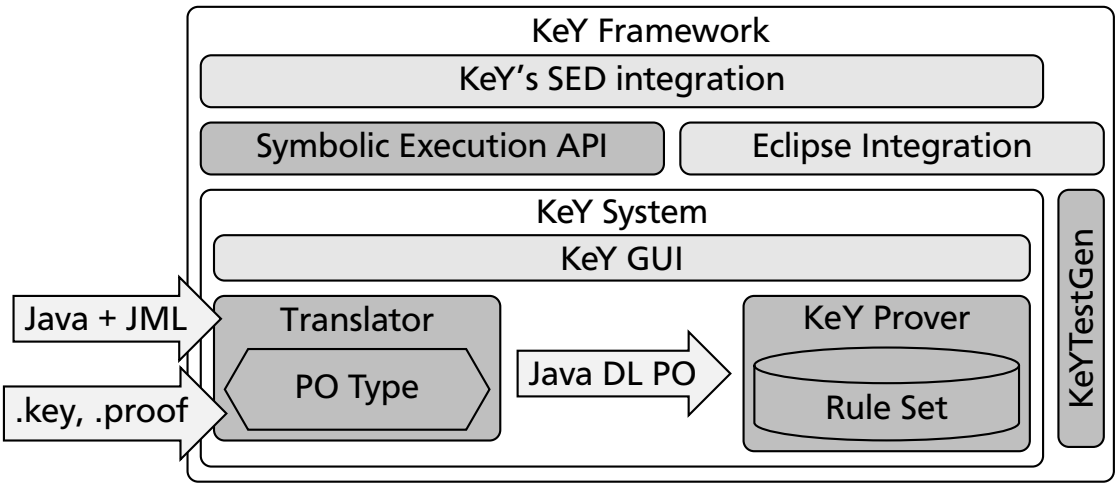


Figure 2.1.: Architecture of the KeY Tool Set (Simplified Version of [5, Chapter 1])

The core of the KeY framework is the *KeY prover* which realizes a Gentzen-style *sequent calculus* [54]. The proof obligation to verify is expressed as a *sequent* of the form

$$\phi_1, \ldots, \phi_n \Longrightarrow \psi_1, \ldots, \psi_m$$

where $\phi_1, \ldots, \phi_n$ and $\psi_1, \ldots, \psi_m$ are formulas. A sequent is valid, iff the formula

$$\bigwedge_{1=i}^{n} \phi_i \rightarrow \bigvee_{1=j}^{m} \psi_j$$

is valid. To prove validity of a sequent, a *proof tree* is constructed by automatic or interactive rule application. The root of the proof tree is the initial proof obligation to show. Rules are applied on leaves of the proof tree, the so called *goals*, until a *closing rule* evaluating the sequent to true is applied. Non-closing rules are transformation descriptions, which are applied on a sequent to construct one or multiple, to some extend simpler, sub sequents. Application of a closing rule does not produce any sub sequents and closes a goal. The proof is closed (successful), iff a closing rule is applied on each leaf of the proof tree.

Formulas $\phi_1, \ldots, \phi_n$ and $\psi_1, \ldots, \psi_m$ are typed first-order logic formulas extended by correctness *modalities* with the source code to verify. The diamond modality $\langle p \rangle \varphi$ takes as first argument the program p to execute. It states, that program p terminates in a state in which φ holds. The box modality

$[p]\varphi$ with p the program to execute states, that if program p terminates, then φ has to hold in the final state. This setup is known as *dynamic logic* [65]. In KeY, the source code within a modality is symbolically executed and state changes are represented by *updates*. The formula $\{\mathcal{U}\}\varphi$ states, that φ holds in the state which is constructed by applying the state changes specified by $\mathcal{U}$ to an initial state. The logic used by KeY to reason about Java programs is called Java DL (Java Dynamic Logic), see Section 2.3.

Consider for instance the proof obligation

$$\mathtt{x} \doteq 0 \Longrightarrow \langle \mathtt{x = x + 1;} \rangle \left(\mathtt{x} \doteq 1\right)$$

stating that assuming $\mathtt{x} \doteq 0$ holds, the Java program `x = x + 1` terminates in a state where $\mathtt{x} \doteq 1$ holds. The slightly simplified proof is shown in proof (2.1). The initial proof obligation is shown on the bottom. After applying the rule on the right, the resulting sequent is shown on top of the previous one.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\mathtt{x} \doteq 0 \Longrightarrow \mathtt{true}}\,\mathsf{closeTrue}}{\mathtt{x} \doteq 0 \Longrightarrow 1 \doteq 1}\,\mathsf{eqClose}}{\mathtt{x} \doteq 0 \Longrightarrow (0+1) \doteq 1}\,\mathsf{add\_zero\_left}}{\mathtt{x} \doteq 0 \Longrightarrow (\mathtt{x}+1) \doteq 1}\,\mathsf{applyEq}}{\mathtt{x} \doteq 0 \Longrightarrow \{\mathtt{x} := \mathtt{x}+1\}\left(\mathtt{x} \doteq 1\right)}\,\mathsf{One\ Step\ Simplification}}{\mathtt{x} \doteq 0 \Longrightarrow \{\mathtt{x} := \mathtt{x}+1\}\langle\rangle\left(\mathtt{x} \doteq 1\right)}\,\mathsf{emptyModality}}{\mathtt{x} \doteq 0 \Longrightarrow \langle \mathtt{x = x + 1;} \rangle \left(\mathtt{x} \doteq 1\right)}\,\mathsf{assignmentAdditionInt} \tag{2.1}$$

First, rule assignmentAdditionInt executes the only statement in the diamond modality. The state change is expressed by update $\{\mathtt{x := x + 1}\}$. Second, the empty modality is dropped by rule emptyModality. Third, the One Step Simplification rule applies the update to the sub formula. Fourth, rule applyEq replaces x on the right side of the sequent separator ($\Longrightarrow$) by 0 according to the formula of the left side of the sequent arrow. Fifth, the addition is performed by rule add_zero_left. Sixth, the equality is compared by rule eqClose. Finally, closing rule closeTrue is applied.

The initial proof obligation (sequent) in KeY is created by a component named *Translator* (see Figure 2.1). The input is either a Java program annotated with JML specifications for which then proof obligations of different kinds are generated, or a problem file with a specific proof obligation to verify. The proof obligation together with the available rules are then passed to the *KeY prover*.

The KeY system for interactive verification is a standalone application and its typical user interface is realized by component *KeY GUI*. The example above verified with KeY is shown in Figure 2.2. The functionality to generate test cases is realized by component *KeYTestGen*.

The remaining components are part of this thesis. First, the *Symbolic Execution API* (Chapter 4) generates a symbolic execution tree from a given proof. Interactive symbolic execution in Eclipse is offered by the SED (Chapter 6). KeY as a symbolic execution engine is made available by KeY's SED integration. Second, the *Eclipse Integration* (Chapter 8) extends the Eclipse platform for interactive verification with KeY which includes in particular an automatic proof management (Chapter 7).

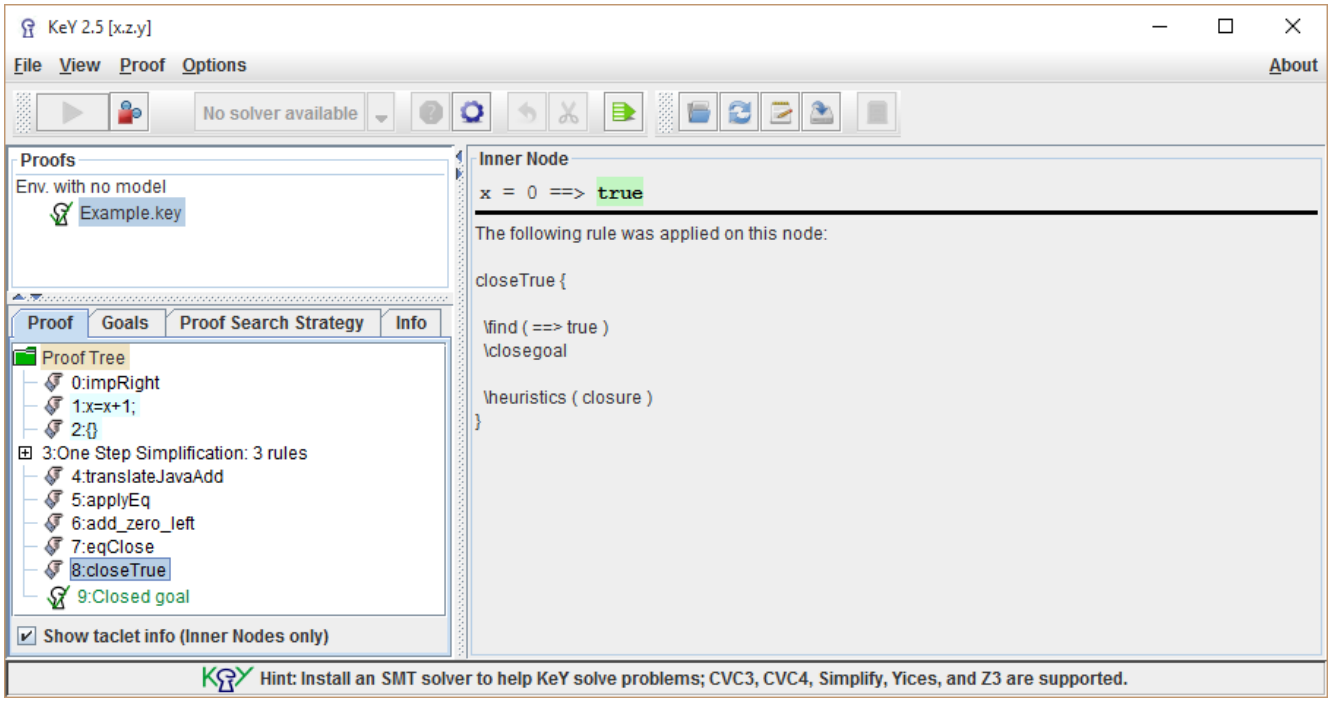


Figure 2.2.: User Interface of KeY

2.3 Java DL

This section introduces Java DL as used by KeY 2.0 [5] or newer. Please observe that some aspects like the modeling of the heap are different in older KeY [22] versions.

The following Section 2.3.1 introduces the syntax of Java DL. Then, the calculus (Section 2.3.2) and the semantics (Section 2.3.3) are discussed. Finally, labels to trace terms or formulas in a proof are introduced in Section 2.3.4.

2.3.1 Syntax

This section introduces the syntax of Java DL, an extended version of a typed first-order logic. The next two definitions specify first the used type hierarchy (Definition 2.1) and then the signature of a typed first order logic (Definition 2.2).

Definition 2.1 (Type Hierarchy [5, Chapter 2])**.** *A* type hierarchy *is a pair* $\mathcal{T} = (\text{TSym}, \sqsubseteq)$*, where*

1. TSym *is a set of type symbols;*
2. $\sqsubseteq$ *is a reflexive, transitive relation on* TSym*, called the subtype relation;*
3. *there are two designated type symbols, the* empty *type* $\bot \in \text{TSym}$ *and the* universal *type* $\top \in \text{TSym}$ *with* $\bot \sqsubseteq A \sqsubseteq \top$ *for all* $A \in \text{TSym}$.

According to [5, Chapter 2], two types A, B in $\mathcal{T}$ are called *incomparable* if neither $A \sqsubseteq B$ nor $B \sqsubseteq A$.

Definition 2.2 (Signature FOL [5, Chapter 2])**.** *A* signature $\Sigma = (\text{FSym}, \text{PSym}, \text{VSym})$ *for a given type hierarchy* $\mathcal{T}$ *is made up of*

1. *a set* FSym *of typed function symbols,*
   by $f : A_1, \ldots, A_n \to A$ *where the argument types of* $f \in \text{FSym}$ *are declared to be* $A_1, \ldots, A_n$ *in the given order and its result type to be* A*,*

2. *a set* PSym *of typed predicate symbols,*
   by $p(A_1,\ldots,A_n)$ *where the argument types of* $p \in$ PSym *are declared to be* $A_1,\ldots,A_n$ *in the given order,*
   PSym *obligatorily contains the binary dedicated symbol* $\doteq(\top,\top)$ *for equality. and the two* 0*-place predicate symbols true and false.*

3. *a set* VSym *of typed variable symbols,*
   by $v : A$ *for* $v \in$ VSym *where* v *is declared to be a variable of type* A*.*

All types A*,* A_i *in this definition must be different from* $\bot$*. A* 0*-ary function symbol* $c : \to A$ *is called a constant symbol of type* A*. A* 0*-ary predicate symbol* $p()$ *is called a propositional variable or propositional atom. Overloading is not allowed: The same symbol may not occur in* FSym $\cup$ PSym $\cup$ VSym *with different typing.*

Java first-order logic (JFOL), the basis of Java DL, is an instantiation of a typed first order logic for verification of Java programs. It requires the mandatory type hierarchy $\mathscr{T}_J$ shown in Figure 2.3 and the mandatory signature Σ_J shown in Figure 2.4.

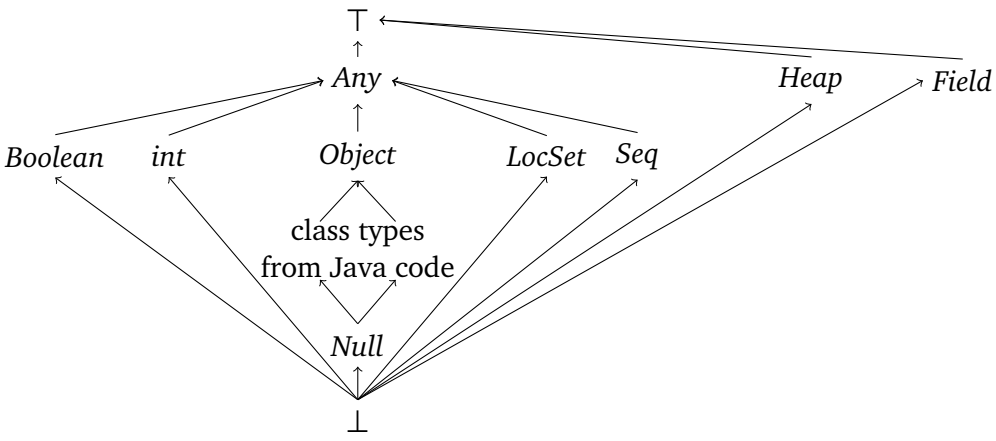


Figure 2.3.: The Mandatory Type Hierarchy $\mathscr{T}_J$ of JFOL [5, Chapter 2]

Basic Java types supported by JFOL are represented by *Boolean* and *int*. Additionally, JFOL supports the types *LocSet* and *Seq* representing location sets and sequences. Available classes are inserted between *Object* and *Null*. Consequently, the complete type hierarchy depends on the verified source code. Heaps are explicitly modeled in JFOL as a theory of arrays. This requires types *Heap* and *Field*.

Please observe that location sets and the related functions will be used later in Section 2.3.2 to express relations between heaps. For instance, to deal with the assignable (modifies) clause of method contracts or loop specifications.

The axioms and semantics related to Figure 2.4 are not important for this thesis. They are discussed in [5, Chapter 2].

int and *Boolean*	all function and predicate symbols for *int*, e.g., $+, *, <, \ldots$
	Boolean constants *TRUE*, *FALSE*
Java types	*null* : *Null*
	length : *Object* $\rightarrow$ *int*
	$cast_A$: *Object* $\rightarrow A$ for any A in $\mathcal{T}$ with $\bot \sqsubset A \sqsubseteq$ *Object*.
	$instance_A$: *Any* $\rightarrow$ *Boolean* for any type $A \sqsubseteq$ *Any*
	$exactInstance_A$: *Any* $\rightarrow$ *Boolean* for any type $A \sqsubseteq$ *Any*
Field	*created* : *Field*
	arr : *int* $\rightarrow$ *Field*
	f : *Field* for every Java field f
Heap	$select_A$: *Heap* $\times$ *Object* $\times$ *Field* $\rightarrow A$ for any type $A \sqsubseteq$ *Any*
	store : *Heap* $\times$ *Object* $\times$ *Field* $\times$ *Any* $\rightarrow$ *Heap*
	create : *Heap* $\times$ *Object* $\rightarrow$ *Heap*
	wellFormed(*Heap*)
LocSet	*elementOf*(*Object*, *Field*, *LocSet*)
	empty, *allLocs* : *LocSet*
	singleton : *Object* $\times$ *Field* $\rightarrow$ *LocSet*
	subset(*LocSet*, *LocSet*)
	disjoint(*LocSet*, *LocSet*)
	union, *intersect*, *setMinus* : *LocSet* $\times$ *LocSet* $\rightarrow$ *LocSet*
	allFields : *Object* $\rightarrow$ *LocSet*, *allObjects* : *Field* $\rightarrow$ *LocSet*
	arrayRange : *Object* $\times$ *int* $\times$ *int* $\rightarrow$ *LocSet*
	unusedLocs : *Heap* $\rightarrow$ *LocSet*
	anon : *Heap* $\times$ *LocSet* $\times$ *Heap* $\rightarrow$ *Heap*

Figure 2.4.: The Mandatory Vocabulary Σ_J of JFOL [5, Chapter 2]

The Java DL signature as extension of JFOL is introduced by Definition 2.3. The difference between *rigid* and *non-rigid* function symbols is according to Ahrendt et al. [5, Chapter 2], that the interpretation of *non-rigid* symbols can be changed by the program, whereas rigid symbols maintain their interpretation throughout program execution. Nullary non-rigid function symbols are called *program variables*.

Definition 2.3 (Signature of Java DL [5, Chapter 3])**.** *Let $\mathcal{T}$ be a Java DL type hierarchy for a Java program Prg. A Java DL* signature *w.r.t. $\mathcal{T}$ is a tuple*

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

where

- FSym_r *and* FSym_{nr} *are disjoint sets of function symbols;*
- $(\mathrm{FSym}_r, \mathrm{PSym}, \mathrm{VSym})$ *is a JFOL signature, i.e., Σ includes the vocabulary from Σ_J (see Figure 2.4);*
- *the set* $\mathrm{PVSym} \subseteq \mathrm{FSym}_{nr}$ *of all nullary non-rigid function symbols, called* program variables*, contains all local variables* a *declared in Prg, where the type of* $\mathtt{a}{:}A \in \mathrm{PVSym}$ *is given by the declared Java type T as follows:*
  - *$A = T$ if T is a reference type,*
  - *$A =$ Boolean if $T =$* `boolean`*,*
  - *$A = int$ if $T \in \{$*`byte`, `short`, `int`, `long`, `char`$\}$.
- $\mathrm{PVSym} \subseteq \mathrm{FSym}_{nr}$ *contains an infinite number of symbols of every typing.*

- $\mathrm{PVSym} \subseteq \mathrm{FSym}_{nr}$ *contains the* special *program variable* $\mathtt{heap} : \mathit{Heap} \in \mathrm{PVSym}$.

The set of all (rigid and the non-rigid) function symbols is denoted by $\mathrm{FSym} = \mathrm{FSym}_r \cup \mathrm{FSym}_{nr}$.

The following definitions introduce inductively terms (Definition 2.4), formulas (Definition 2.5), legal program fragments (Definition 2.6) and updates (Definition 2.7) of Java DL.

Definition 2.4 (Java DL Term [5, Chapter 2 and Chapter 3]). *Let Prg be a Java program,* $\mathscr{T}$ *a type hierarchy for Prg, and* Σ *a signature w.r.t.* $\mathscr{T}$. *The set* Trm_A *of Java DL terms of type A, for* $A \neq \bot$, *is inductively defined by*

1. $v \in \mathrm{Trm}_A$ *for each variable symbol* $v : A \in \mathrm{VSym}$ *of type A.*
2. $f(t_1, \ldots, t_n) \in \mathrm{Trm}_A$ *for each* $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}$ *and all terms* $t_i \in \mathrm{Trm}_{B_i}$ *with* $B_i \sqsubseteq A_i$ *for* $1 \leq i \leq n$.
3. $(\text{if } \phi \text{ then } t_1 \text{ else } t_2) \in \mathrm{Trm}_A$ *for* $\phi \in \mathrm{Fml}$ *and* $t_i \in \mathrm{Trm}_{A_i}$ *such that* $A_2 \sqsubseteq A_1 = A$ *or* $A_1 \sqsubseteq A_2 = A$.
4. $\{u\}t \in \mathrm{Trm}_A$ *for all updates* $u \in \mathrm{Upd}$ *and all terms* $t \in \mathrm{Trm}_A$.

$t \in \mathrm{Trm}_A$ *says that* t *is of (static) type A, written as* $\sigma(t) = A$. *A term is called rigid if it does not contain any occurrences of non-rigid function symbols.*

Definition 2.5 (Java DL Formula [5, Chapter 2 and Chapter 3]). *Let Prg be a Java program,* $\mathscr{T}$ *a type hierarchy for Prg, and* Σ *a signature w.r.t.* $\mathscr{T}$. *The set* Fml *of Java DL formulas is inductively defined by*

1. $p(t_1, \ldots, t_n) \in \mathrm{Fml}$
   for $p(A_1, \ldots, A_n) \in \mathrm{PSym}$, *and* $t_i \in \mathrm{Trm}_{B_i}$ *with* $B_i \sqsubseteq A_i$ *for all* $1 \leq i \leq n$.
   As a consequence of item 2 in Definition 2.2 it is known that
   $t_1 \doteq t_2 \in \mathrm{Fml}$ *for arbitrary terms* t_i *and true and false are in* Fml.
2. $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \to \psi)$, $(\phi \leftrightarrow \psi)$ *are in* Fml *for arbitrary* $\phi, \psi \in \mathrm{Fml}$.
3. $\forall v; \phi$, $\exists v; \phi$ *are in* Fml *for* $\phi \in \mathrm{Fml}$ *and* $v : A \in \mathrm{VSym}$.
4. $(\text{if } \phi \text{ then } \psi_1 \text{ else } \psi_2) \in \mathrm{Fml}$ *for* $\phi, \psi_1, \psi_2 \in \mathrm{Fml}$.
5. $\langle p \rangle \phi$, $[p]\phi \in \mathrm{Fml}$ *for all legal program fragments p.*
6. $\{u\}\phi \in \mathrm{Fml}$ *for all formulas* $\phi \in \mathrm{Fml}$ *and updates* $u \in \mathrm{Upd}$.

If need arises the dependence of these definitions on Σ *and* $\mathscr{T}$ *is made explicit by writing* $\mathrm{Trm}_{A,\Sigma}$, Fml_Σ *or* $\mathrm{Trm}_{A,\mathscr{T},\Sigma}$, $\mathrm{Fml}_{\mathscr{T},\Sigma}$. *In addition, the redundant notation* $\forall\ A\ v; \phi$, $\exists\ A\ v; \phi$ *for a variable* $v : A \in \mathrm{VSym}$ *is used when convenient.*

Formulas built by clause (1) only are called atomic formulas. *A formula is called rigid if it does not contain any occurrences of non-rigid function symbols.*

Definition 2.6 (Legal Program Fragments [5, Chapter 3]). *Let Prg be a Java program. A* legal program fragment p in the context of *Prg is a sequence of Java statements, where there are local variables* $\mathtt{a}_1, \ldots, \mathtt{a}_n \in \mathrm{PVSym}$ *of Java types* $T_1, \ldots, T_n$ *such that extending Prg with an additional class*

```
class C {
    static void m(T_1 a_1, ..., T_n a_n) { p }
}
```

yields again a legal program according to the rules of the Java language specification [60], except that

- *p may refer to fields, methods and classes that are not visible in* C, *and*

- *p may contain* method frames *in addition to normal Java statements. A method frame is a statement of the form*

  `method-frame(result->`r`, this=`t`) : { `*body*` }` ,

  where (i) r is a local variable, (ii) t is an expression free from side-effects and from method calls, and (iii) body is a legal program fragment in the context of Prg. The semantics of a method frame is that, inside body (but outside of any nested method frames that might be contained in body), the keyword `this` *evaluates to the value of t, and the meaning of a* `return` *statement is to assign the returned value to r and to then exit the method frame.*

- *p may contain a method body statement of the form methodName*$(a_1, \ldots, a_n)$*@className. Intuitively, this is a placeholder for the method body of method methodName with the most specific signature according to the types of the arguments* $a_1, \ldots, a_n$ *as implemented in class className.*

Definition 2.7 (Updates [5, Chapter 3])**.** *Let Prg be a Java program,* $\mathcal{T}$ *a type hierarchy for Prg, and* Σ *a signature for* $\mathcal{T}$*. The set* Upd *of updates is inductively defined*

- $(\mathtt{a} := t) \in \mathrm{Upd}$ *for each program variable symbol* $\mathtt{a} : A \in \mathrm{PVSym}$ *and each term* $t \in \mathrm{Trm}_{A'}$ *such that* $A' \sqsubseteq A$*.*
- $(u_1 \,||\, u_2) \in \mathrm{Upd}$ *for all updates* $u_1, u_2 \in \mathrm{Upd}$*.*
- $(\{u_1\}u_2) \in \mathrm{Upd}$ *for all updates* $u_1, u_2 \in \mathrm{Upd}$*.*

2.3.2 Calculus

This section introduces the Java DL sequent calculus to the extent that is necessary for this thesis. The calculus and related rules are discussed in detail in [5].

First Order Rules

In a sequent calculus, rules are applied to *sequents* of the form

$$\phi_1, \ldots, \phi_n \Longrightarrow \psi_1, \ldots, \psi_m$$

where $\phi_1, \ldots, \phi_n$ and $\psi_1, \ldots, \psi_m$ are formulas. Formulas $\phi_1, \ldots, \phi_n$ on the left side of the sequent separator $\Longrightarrow$ are named *antecedents* whereas formulas $\psi_1, \ldots, \psi_m$ are called *succedents*. In Java DL, formulas in the antecedent and succedent are unordered sets. A sequent is valid according to Ahrendt et al. [5, Chapter 2], iff the formula

$$\bigwedge_{1=i}^{n} \phi_i \rightarrow \bigvee_{1=j}^{m} \psi_j$$

is valid.

Rules applied to a sequent are of the form

$$\mathsf{ruleName}\ \frac{P_1, \ldots, P_n}{C}$$

where $P_1, \ldots, P_n$ are the *premisses* of the form $\Gamma \Longrightarrow \Delta$ and C is the *conclusion* of the form $\Gamma \Longrightarrow \Delta$. The schematic variables Γ, Δ are added to premiss and conclusion to represent the set of formulas not touched by the rule. Let ϕ and ψ be schematic variables: Γ, ϕ and ψ, Δ mean $\Gamma \cup \{\phi\}$ and $\{\psi\} \cup \Delta$,

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

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

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

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

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

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

Figure 2.5.: Classical Propositional Logic Rules [5, Chapter 2 and Chapter 3]

respectively. In Java DL, rules are applied from bottom to top by replacing the schematic variables in premiss and conclusion by the entities according to the sequent on which the rule is applied. The classical propositional logic rules are shown for instance in Figure 2.5. Missing first order rules are presented by Ahrendt et al. [5, Chapter 2].

Rules with an empty premiss do not require further rule applications and are called *closing rules*, see Definition 2.8.

Definition 2.8 (Closing Rules [5, Chapter 2])**.** *The rules* close, closeFalse, *and* closeTrue *from Figure 2.5 are called* closing rules *since their premisses are empty.*

Some rules have more than one premiss resulting in a *proof tree* according to Definition 2.9.

Definition 2.9 (Proof Tree [5, Chapter 2])**.** *A* proof tree *is a finite tree, shown with the root at the bottom, such that*

1. *each node is labeled with a sequent or the symbol* $*$*,*
2. *if an inner node* n *is annotated with* $\Gamma \Longrightarrow \Delta$ *then there is an instance of a rule whose conclusion is* $\Gamma \Longrightarrow \Delta$ *and the child node, or children nodes of* n *are labeled with the premiss or premisses of the rule instance.*

A branch in a proof tree is called closed *if its leaf is labeled by* $*$*. A proof tree is called* closed *if all its branches are closed, or equivalently if all its leaves are labeled with* $*$*.*

A sequent $\Gamma \Longrightarrow \Delta$ *can be derived if there is a closed proof tree whose root is labeled by* $\Gamma \Longrightarrow \Delta$*.*

The proof tree of proof (2.2) for instance verifies $p \wedge q \rightarrow q \wedge p$ with help of the classical propositional rules from Figure 2.5.

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{p, q \Longrightarrow q}\ \text{close} \quad \dfrac{*}{p, q \Longrightarrow p}\ \text{close}}{p, q \Longrightarrow q \wedge p}\ \text{andRight}}{p \wedge q \Longrightarrow q \wedge p}\ \text{andLeft}}{\Longrightarrow p \wedge q \rightarrow q \wedge p}\ \text{impRight} \tag{2.2}$$

Rewrite Rules

In addition to sequent rules which are applied to formulas, *rewrite rules* allow one to replace a term or formula anywhere in the sequent. Ahrendt et al. [5, Chapter 2] introduce a rewrite rule $s \rightsquigarrow t$ as shorthand for a sequent rule $\frac{\Gamma' \Longrightarrow \Delta'}{\Gamma \Longrightarrow \Delta}$ where $\Gamma' \Longrightarrow \Delta'$ arises from $\Gamma \Longrightarrow \Delta$ by replacing one or more occurrences of the term s by t. The rewrite rules add_zero_left and eqClose applied in proof (2.1) are shown in Figure 2.6.

$$\begin{array}{rl} \mathsf{add\_zero\_left} & 0 + i \rightsquigarrow i \\ \mathsf{eqClose} & s \doteq s \rightsquigarrow \mathrm{true} \end{array}$$

Figure 2.6.: Examples of Rewrite Rules

In KeY, nearly all rules are specified in the *taclet* [5, Chapter 4] language. Consider for instance rewrite rule applyEq in proof (2.1) which replaces x by 0 because $\mathtt{x} \doteq 0$ is part of the antecedent. The rule definition as taclet is shown in Figure 2.7. First, schematic variables are declared by `\schemaVar`. Second, the rule is only applicable if the `\assumes` clause is fulfilled. Third, the position at which the rule is applied is specified by the `\find` clause. Fourth, with `\sameUpdateLevel` it is ensured that the rule is only applicable if `\find` and `\replacewith` are in the same execution state (updates are discussed later in detail). Fifth, `\replacewith` specifies the replacement term. Finally, `\heuristics` are hints for an automatic proof search strategy about when the rule should be applied and does not have any logical meaning.

Taclet

```
applyEq {
  \schemaVar \term G s;
  \schemaVar \term H t;

  \assumes (s = t ==>  )
  \find ( s )
  \sameUpdateLevel
  \replacewith ( t )
  \heuristics ( apply_select_eq, apply_equations )
}
```

Taclet

Figure 2.7.: Taclet of Rewrite Rule applyEq

Symbolic Execution Rules

The Java DL calculus reduces according to Ahrendt et al. [5, Chapter 3] the question of a formula's validity to the question of the validity of several simpler formulas. The rules dealing with programs in modalities are designed to follow the symbolic execution paradigm. A symbolic execution rule always executes the first *active* statement in a modality, see Definition 2.10.

Definition 2.10 (First Active Statement [5, Chapter 3])**.** *Rules performing symbolic execution operate on the first* active *statement p in a modality* $\langle \pi p \omega \rangle$ *or* $[\pi p \omega]$*. The non-active prefix* π *consists of an arbitrary sequence of opening braces "*`{`*", labels, beginnings "*`try{`*" of* `try-catch-finally` *blocks, and beginnings "*`method-frame(...){`*" of method invocation blocks. The prefix is needed to keep track of the blocks that the*

(first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, *and* `continue` *can be handled appropriately.*

The postfix ω denotes the rest of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on (in particular, ω contains closing braces corresponding to the opening braces in π).

Consider for instance the following Java block of a modality taken from [5, Chapter 3]. The non-active prefix π, the first active statement p and the postfix ω are highlighted.

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

After all statements in a modality are executed, the empty modality can be removed by the rules in Figure 2.8.

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

Figure 2.8.: Empty Modality Rules [5, Chapter 3]

In the following, different aspects of symbolic execution in Java DL relevant for this thesis are introduced. Rules are only presented for the diamond modality. For the box modality, the rules are mostly the same except for the handling of abrupt termination (e.g. exceptions).

The rule treating an assignment of a local variable is shown in Figure 2.9. The rule removes the active assignment from the modality and ensures that execution is continued in a state constructed by the added update. Intuitively, updates can be seen as explicit substitutions for which several simplification and application rules are provided. For details see [5, Chapter 3]. A simple syntactic substitution is not possible because multiple variables might point to the same object in Java. This is called *aliasing*.

In the context of this thesis, all rules dealing with update simplification and application are automatically applied as part of the *One Step Simplification*. This is a macro in KeY which applies several rewrite rules on the same term for efficiency reasons. Rules for update simplification and application are part of the one step simplification.

$$\text{assignment}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{variable := value\}\langle \pi\ \ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\,\langle \pi\ \ variable\ =\ value;\ \ \omega\rangle\phi, \Delta}$$

Figure 2.9.: Assignment Rule [5, Chapter 3]

Consider for example proof (2.3) verifying $\langle \mathtt{x\ =\ 1;}\rangle\left(\mathtt{x} \doteq 1\right)$. First, the assignment rule is applied to execute the assignment. Second, the empty modality is removed. Third, rewriting rules applying the update and evaluating the equality are applied at once by the *One Step Simplification*. Finally, the proof is closed by closeTrue.

$$\dfrac{\dfrac{\dfrac{\overset{*}{\Longrightarrow \text{true}}\ \text{closeTrue}}{\Longrightarrow \{\mathtt{x} := 1\}\left(\mathtt{x} \doteq 1\right)}\ \textit{One Step Simplification}}{\Longrightarrow \{\mathtt{x} := 1\}\,\langle\rangle\left(\mathtt{x} \doteq 1\right)}\ \text{emptyModality}}{\Longrightarrow \langle \mathtt{x\ =\ 1;}\rangle\left(\mathtt{x} \doteq 1\right)}\ \text{assignment} \tag{2.3}$$

A single assignment rule, like the one in Figure 2.9, is not enough to cover all kinds of assignments in Java. Besides local variables, static or instance fields might be assigned as well. Additionally, the

declaration of a variable or a field might be combined with an assignment to define the initial value. Also the left or right side of the assignment might be a complex expression containing method calls or computing values. Such expressions may have side effects, meaning that they change other locations during execution before or after the assignment is performed.

To cover all the details of Java, a process called *unfolding* is performed by Java DL rules. Complex Java expressions are step-wise decomposed by rules into atomic statements. Fresh program variables are introduced to store intermediate results. In case of assignments, a number of assignment rules exist to perform unfolding and to cover the different kinds of written locations.

Consider for instance Listing 2.8 which declares instance fields of an item of a `LinkedList`. The proof verifying $\langle$`li.next.value = 8;`$\rangle$ true where `li` is an instance of `ListItem` is shown in proof (2.4).

```
1 public class ListItem {
2   private ListItem next;
3   private int value;
4 }
```

Listing 2.8: Fields of a List Item as Used by a Linked List

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\{\texttt{l} := \texttt{li.next}\}\texttt{l} \doteq \texttt{null},\ \Longrightarrow\ \texttt{li} \doteq \texttt{null},\ \text{true}}\ \text{closeTrue}}{\{\texttt{l} := \texttt{li.next}\}\texttt{l} \doteq \texttt{null},\ \Longrightarrow\ \texttt{li} \doteq \texttt{null},\ \{\texttt{l} := \texttt{li.next}\}\{\texttt{heap} := \texttt{heap}[\texttt{l.value} := 8]\}\ \text{true}}\ \textit{One Step Simplification}}{\{\texttt{l} := \texttt{li.next}\}\texttt{l} \doteq \texttt{null},\ \Longrightarrow\ \texttt{li} \doteq \texttt{null},\ \{\texttt{l} := \texttt{li.next}\}\{\texttt{heap} := \texttt{heap}[\texttt{l.value} := 8]\}\langle\rangle\ \text{true}}\ \text{emptyModality}}{\Longrightarrow\ \texttt{li} \doteq \texttt{null},\ \{\texttt{l} := \texttt{li.next}\}\langle\texttt{l.value = 8;}\rangle\ \text{true}}\ \text{assignment\_write\_attribute}}{\Longrightarrow \left\langle \begin{array}{l}\texttt{l = li.next;}\\ \texttt{l.value = 8;}\end{array}\right\rangle \text{true}}\ \text{assignment\_read\_attribute}}{\Longrightarrow \left\langle \begin{array}{l}\texttt{ListItem l;}\\ \texttt{l = li.next;}\\ \texttt{l.value = 8;}\end{array}\right\rangle \text{true}}\ \text{variableDeclaration}}{\Longrightarrow \left\langle \begin{array}{l}\texttt{ListItem l = li.next;}\\ \texttt{l.value = 8;}\end{array}\right\rangle \text{true}}\ \text{variableDeclarationAssign}}{\Longrightarrow \langle\texttt{li.next.value = 8;}\rangle\ \text{true}}\ \text{eval\_order\_access1}$$

NPE Branch

NPE Branch

(2.4)

Without giving the rules, how symbolic execution works in the Java DL calculus can be seen in proof (2.4). First, to assign a value to `li.next.value` the object referenced by `li.next` needs to be known. Rule eval_order_access1 declares for this reason a fresh local program variable and assigns `li.next` to it. Second, a statement declaring a variable and assigning a value to it needs to be unfolded. Here, unfolding is performed by rule variableDeclarationAssign. Third, the variable `l` is created by rule variableDeclaration. Fourth, the instance field `li.next` is read and the result is assigned to `l` by rule assignment_read_attribute. Accessing an instance field in Java can cause a `NullPointerException` at runtime. The shown branch continues execution in case that $\neg$(`li` $\doteq$ `null`) holds which is added in

negated form to the succedent. The additional NPE branch continues execution in case that `li` $\doteq$ `null` holds instead by instantiating and throwing a `NullPointerException`. Fifth, 8 is assigned to the instance field `l.value` by rule assignment_write_attribute. As before, accessing the field can cause a `NullPointerException` and the proof splits for this reason. Finally, the empty modality is removed, the formula is simplified and this proof tree branch is closed.

Some of the Java DL rules dealing with an **`if`**-statement are shown in Figure 2.10. If the condition is a complex expression, then it needs to be unfolded, e.g. by rule ifElseUnfold. Once this is done, the case distinction is made by rule ifElseSplit. Other conditional statements of Java are rewritten into **`if`**-statements during symbolic execution.

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

$$\text{ifElseSplit}\ \frac{\begin{array}{c}\Gamma, \mathcal{U}\left(se \doteq \text{TRUE}\right) \Longrightarrow \mathcal{U}\langle \pi\ p\ \omega\rangle\phi, \Delta \\ \Gamma, \mathcal{U}\left(se \doteq \text{FALSE}\right) \Longrightarrow \mathcal{U}\langle \pi\ q\ \omega\rangle\phi, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \texttt{if (}se\texttt{)}\ p\ \texttt{else}\ q\ \omega\rangle\phi, \Delta}$$

Figure 2.10.: Some Conditional Rules [5, Chapter 3]

Loops can be unrolled during symbolic execution by rule loopUnwind shown in Figure 2.11 as long as the loop contains no forward jumps in form of **`break`** or **`continue`** statements. The extended version of the rule supporting jumps can be found in [5, Chapter 3].

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

Figure 2.11.: Simple Loop Unwinding Rule [5, Chapter 3]

Treating a method call by inlining the method implementation consists according to [5, Chapter 3] of the following steps in Java DL:

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.

Please observe that method inlining is not modular and requires that all possible method implementations are available (closed world assumption).

The first step identifies all possible method implementations based on the statically available information according to the Java language specification [60]. The next two steps unfold the target reference and the arguments of the method call. The fourth step checks for a possible `NullPointerException` and simulates the *dynamic dispatch* by inserting an **`if`**-cascade over all fitting method implementations. During symbolic execution of the **`if`**-cascade, the proof will split and each branch will inline a different method implementation in the fifth step. Then the inlined code is executed and finally the method returns in the sixth step.

Without giving the rules, the example proof (2.5) verifying $\langle \texttt{m = obj.magic();} \rangle \, \text{true}$ shows how method inlining works in Java DL. The implementation of method `magic` is shown in Listing 2.9.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Longrightarrow \texttt{obj} \doteq \texttt{null},\ \text{true}}\,*}{\Longrightarrow \texttt{obj} \doteq \texttt{null},\ \langle \texttt{x = 42;} \rangle \text{true}}\,\text{closeTrue}}{\Longrightarrow \texttt{obj} \doteq \texttt{null},\ \{\texttt{heapBefore\_magic := heap}\}\{\texttt{x := 42}\} \langle \texttt{m = x;} \rangle \text{true}}\,\textit{One Step Simplification}}{\Longrightarrow \texttt{obj} \doteq \texttt{null},\ \{\texttt{heapBefore\_magic := heap}\}\{\texttt{x := 42}\}\{\texttt{m := x}\} \langle \rangle \text{true}}\,\text{emptyModality}}{\Longrightarrow \texttt{obj} \doteq \texttt{null},\ \{\texttt{heapBefore\_magic := heap}\}\{\texttt{x := 42}\}\{\texttt{m := x}\} \text{true}}\,\text{assignment}}{\Longrightarrow \texttt{obj} \doteq \texttt{null},\ \{\texttt{heapBefore\_magic := heap}\}\{\texttt{x := 42}\} \left\langle \begin{array}{l} \texttt{method-frame(this=obj) : \{\}} \\ \texttt{m = x;} \end{array} \right\rangle \text{true}}\,\text{methodCallEmpty}}{\Longrightarrow \texttt{obj} \doteq \texttt{null},\ \{\texttt{heapBefore\_magic := heap}\} \left\langle \begin{array}{l} \texttt{method-frame(this=obj) : \{x = 42;\}} \\ \texttt{m = x;} \end{array} \right\rangle \text{true}}\,\text{assignment}}{\texttt{obj} \doteq \texttt{null},\ \{\texttt{heapBefore\_magic := heap}\} \left\langle \begin{array}{l} \texttt{method-frame(result->x, this=obj) : \{return 42;\}} \\ \texttt{m = x;} \end{array} \right\rangle \text{true} \Longrightarrow}\,\text{methodCallReturn}}{\Longrightarrow \texttt{obj} \doteq \texttt{null},\ \langle \texttt{x = obj.magic()@Example; m = x;} \rangle \text{true}}\,\text{methodBodyExpand}}{\Longrightarrow \texttt{obj} \doteq \texttt{null},\ \langle \texttt{int x; x = obj.magic()@Example; m = x;} \rangle \text{true}}\,\text{variableDeclaration}}{\Longrightarrow \langle \texttt{m = obj.magic();} \rangle \text{true}}\,\text{methodCallWithAssignment}$$

$\vdots$ NPE Branch

(2.5)

```
1 public class Example {
2    public int magic() {
3       return 42;
4    }
5 }
```

Listing 2.9: Simple Method Symbolically Executed in Proof (2.5)

First, rule methodCallWithAssignment simulates the dynamic dispatch. Here, the method is not overridden and only one implementation is available. Consequently, an **`if`**-cascade is not needed. In case that $\neg(\texttt{obj} \doteq \texttt{null})$ holds, symbolic execution is continued by introducing the local variable `x` which receives the return value after inline execution of `obj.magic()@Example`. The alternative branch continuing execution by throwing a `NullPointerException` is indicated by the NPE branch. Second, variable `x` is declared by rule variableDeclaration. Third, the inlining is performed by rule methodBodyExpand. The rule also creates a new method frame specifying the **`this`** object and where the return value should later be stored. Fourth, the inlined return statement is then executed by methodCallReturn. Fifth, after the inlined source code is executed, the returned value is assigned to the result variable by rule assignment.

Sixth, the now empty method frame is removed by rule methodCallEmpty. Seventh, symbolic execution is now continued at the method call point. Here, by removing an empty block followed by an assignment. Finally, the modality is empty and the branch is closed after performing the One Step Simplification.

Memory Model

In Java DL, the method call stack is modeled by method frames (Definition 2.6). But local variables are not part of each method frame as in Java, instead, a global name space containing all local variables is used. Consequently, if a local variable is declared which is already part of the name space, it is renamed during declaration by adding an ongoing number as suffix. This is done by rule variableDeclaration.

A heap is modeled in Java DL as type *Heap* following the patterns of the theory of arrays. A field of type A is queried by the $select_A$ function and a field value is set by the *store* function. Additionally, the *create* function allows one to create new objects and the *anon* function to assign fresh symbolic values nothing is known about to fields. Consider for instance $select_A(store(h, o_1, f_1, x), o_2, f_2)$ which queries the value of field f_2 with type A on the object o_2 in the heap specified by $store(h, o_1, f_1, x)$. This heap is derived from heap h by setting the value of field f_1 on object o_1 to x. The result of the $select_A$ function is x, if $o \doteq o_2 \wedge f \doteq f_2$ holds and $select_A(h, o2, f2)$ otherwise.

The heap modified by execution of the active statement in a modality in Java DL is stored in program variable `heap`. Update {`heap` := `heap[l.value := 8]`} of proof (2.4) shows for instance a heap modification where 8 is assigned to the field `l.value` of the current heap `heap`. Please observe that this is a pretty syntax for {`heap` := *store*(`heap`, `l`, `ListItem::$value`, 8)}.

As a heap is a type, different program variables can contain different heaps. This can be seen in proof (2.5) where the update {`heapBefore_magic` := `heap`} stores a copy of `heap` in `heapBefore_magic`.

In the context of this thesis, the *anon* function is only used to assign fresh symbolic values to fields. Consider for instance $anon(store(h_1, o, f, x), singleton(o, f), h_N)$ which derives a new heap from $store(h_1, o, f, x)$ in which the single location $singleton(o, f)$ is evaluated in h_N whereas all other locations are evaluated in h_1. Anonymization is achieved, because h_N is in the context of this thesis always a fresh heap nothing is known about.

Symbolic Execution Rules for Specifications

To avoid a closed world assumption and to support modular reasoning, method contracts (Definition 2.11) can be used instead of method inlining. A method contract specifies by the precondition when it is applicable and by the postcondition the behavior of the method.

Definition 2.11 (Method Contract [5, Chapter 3])**.** *A method contract for a method or constructor op declared in a class or interface C of the Java program is a quadruple*

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

where:

- $Pre \in$ Fml *is the precondition that may contain the following program variables:*
  - *self for the receiver object (the object on which a caller invokes the method); 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.*
- $Post \in$ Fml *is the postcondition of the form*

$$(exc \doteq \texttt{null} \rightarrow \phi) \wedge (exc \mathrel{!}\doteq \texttt{null} \rightarrow \psi)$$

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 formulas ϕ and ψ may contain the following program variables:

- *self for the receiver object; again the receiver object variable is not allowed for static methods;*
- *$p_1, \ldots, p_n$ for the parameters;*
- *result for the returned value;*

- *Mod is a modifier set for the method op. It is also known as assignable or modifies clause.*
- *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 simplified rule to apply a method contract is shown in Figure 2.12. The rule is applicable, if a contract $(Pre, Post, Mod, \text{TOTAL})$ for the method $mname(T_1, \ldots, T_n)$ declared in class C is given.

Use Operation Contract

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\{self := se_{target} \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n\} Pre, \Delta \\ \Gamma \Longrightarrow \mathcal{U}\, se_{target} \mathrel{!\doteq} \texttt{null}, \Delta \\ \Gamma, \mathcal{U}\{\mathcal{V}(Mod)\} exc \doteq \texttt{null} \Longrightarrow \begin{array}{l}\mathcal{U}\{\mathcal{V}(Mod) \,||\, self := se_{target} \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n \,||\, lhs := result\} \\ (Post \rightarrow \langle \pi\ \omega \rangle \phi)\end{array}, \Delta \\ \Gamma, \mathcal{U}\{\mathcal{V}(Mod)\} exc \mathrel{!\doteq} \texttt{null} \Longrightarrow \begin{array}{l}\mathcal{U}\{\mathcal{V}(Mod) \,||\, self := se_{target} \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n\} \\ (Post \rightarrow \langle \pi\ \texttt{throw}\ exc;\ \omega \rangle \phi)\end{array}, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \ lhs{=}se_{target}.mname(se_1, \ldots, se_n)@C;\ \ \omega \rangle \phi, \Delta}$$

Figure 2.12.: Simplified Use Operation Contract Rule [5, Chapter 3]

The first premiss verifies that the precondition is fulfilled in the current state. The second premiss ensures that no `NullPointerException` is thrown in case the target reference se_{target} is an object other than **`this`**.

Third and fourth premiss continue execution after the method call in case of normal and exceptional termination. $\mathcal{V}(Mod)$ is an anonymising update w.r.t. the modifier set *Mod* of the method contract. In Java DL, anonymization of heap locations is achieved by the *anon* function as explained above. The parameters are the current heap to anonymize, the set of locations to anonymize and a different heap providing fresh symbolic values. Rule Use Operation Contract introduces for the third parameter always a new heap never used before.

Instead of unrolling a loop, a loop specification according to Definition 2.12 can be applied.

Definition 2.12 (Loop Specification)**.** *A loop specification for a loop l is a tuple*

$$(Inv, Var, Mod)$$

where:

- *$Inv \in \text{Fml}$ is the invariant.*
- *$Var \in \text{Trm}_{int}$ is the variant (decreasing term) of type int.*
- *Mod is a modifier set for the loop l. It is also known as assignable or modifies clause.*

Loop Invariant

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\,Inv, \Delta \\ \Gamma, \mathcal{U}\{\mathcal{V}(Mod)\}Inv \Longrightarrow \mathcal{U}\{\mathcal{V}(Mod) \,||\, \mathtt{v} := Var\} \\ \left(\begin{array}{l} [immf(\pi)\{\texttt{boolean b = } e\texttt{;}\}\big(\mathtt{b} \doteq \mathrm{TRUE}\big) \\ \quad \rightarrow \left\langle \begin{array}{l} immf(\pi)\{\texttt{Throwable twnExc = null;} \\ \texttt{try \{if (}e\texttt{) } p\texttt{\}} \\ \texttt{catch(Throwable t) \{twnExc = t;\}\}} \end{array} \right\rangle \\ \left(\begin{array}{l} \big(\texttt{twnExc} \doteq \texttt{null} \rightarrow (Inv \wedge Var < \mathtt{v} \wedge Var > 0 \wedge Mod_T)\big) \\ \wedge \big(\texttt{twnExc !}\doteq \texttt{null} \rightarrow \langle \pi \ \texttt{throw twnExc;} \ \omega\rangle\phi\big) \end{array} \right) \end{array} \right), \Delta \\ \Gamma, \mathcal{U}\{\mathcal{V}(Mod)\}Inv \Longrightarrow \mathcal{U}\{\mathcal{V}(Mod)\}[immf(\pi)\{\texttt{boolean b = } e\texttt{;}\}\big(\mathtt{b} \doteq \mathrm{FALSE} \rightarrow \langle \pi \ \omega\rangle\phi\big), \Delta \end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi \ \texttt{while (}e\texttt{) } p \ \omega\rangle\phi, \Delta}$$

Figure 2.13.: Simplified Loop Invariant Rule

The loop invariant rule for loops without **break**, **continue** and **return** statements is shown in Figure 2.13.

The rule Loop Invariant is applicable, if a loop specification (Definition 2.12) is given for the loop to execute. The first premiss ensures that the loop invariant *Inv* holds in the current state. The second premiss ensures, that if the loop guard e holds in the current state, after execution of loop guard e and loop body p in an arbitrary loop iteration, the loop invariant still holds. Additionally, the variant *Var* needs to be decreased and must be greater then 0. Mod_T is a predicate, which ensures that only locations according to *Mod* are changed. In case the loop guard or loop body throws an exception, execution is continued to verify ϕ by throwing the caught exception. The third premiss continues execution after an arbitrary number of loop iterations in case that loop guard e does not hold. $\mathcal{V}(Mod)$ is an anonymising update w.r.t. the modifier set *Mod* of the loop specification. Local variables are anonymized by assigning a fresh symbolic value to them. $immf(\pi)$ is the inner most method frame of π.

Soundness and Completeness

Soundness (Proposition 2.1), the property of a calculus that everything which is derivable is valid, is the most important property of the Java DL calculus. Ahrendt et al. [5, Chapter 3] argue that soundness is achieved if all Java DL core rules are sound. The taken measures to validate the soundness of the core rules are presented as well. Additional user defined rules can be validated within the calculus itself.

Proposition 2.1 (Soundness [5, Chapter 3])**.** *If a sequent $\Gamma \Longrightarrow \Delta$ is derivable in the Java DL calculus (Def. 2.9), then it is valid, i.e., the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is logically valid (Def. 2.17).*

Completeness, the property of a calculus that everything which is valid is derivable, is impossible for the Java DL calculus according to Ahrendt et al. [5, Chapter 3]. One mentioned reason is that Java DL includes first-order arithmetic, which is already inherently incomplete as established by Gödel's Incompleteness Theorem. Apart from that, a relative completeness (Proposition 2.2) is achieved. It is proven for the object-oriented dynamic logic ODL, which captures the essence of Java DL, by Platzer [106].

Proposition 2.2 (Relative Completeness [5, Chapter 3])**.** *If a sequent $\Gamma \Longrightarrow \Delta$ is valid, i.e., the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is logically valid (Def. 2.17), then there is a finite set Γ_{FOL} of logically valid first-order formulas such that the sequent*

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

is derivable in the Java DL calculus.

2.3.3 Semantics

In Java DL, updates represent different program states. Therefore, formulas are evaluated in a *Kripke structure* according to Definition 2.15. Beforehand, a domain (Definition 2.13) and a first-order structure (Definition 2.14) are introduced.

Definition 2.13 (Domain [5, Chapter 2]). *A* universe *or* domain *for a given type hierarchy $\mathscr{T}$ and signature Σ consists of*

1. *a set D,*
2. *a typing function $\delta : D \to \mathrm{TSym} \setminus \{\bot\}$ such that for every $A \in \mathrm{TSym}$ the set $D^A = \{d \in D \mid \delta(d) \sqsubseteq A\}$ is not empty.*

The set $D^A = \{d \in D \mid \delta(d) \sqsubseteq A\}$ is called the type universe or type domain for A. Definition 2.13 implies that for different types $A, B \in \mathrm{TSym} \setminus \{\bot\}$ there is an element $o \in D^A \cap D^B$ only if there exists $C \in \mathrm{TSym}$, $C \neq \bot$ with $C \sqsubseteq A$ and $C \sqsubseteq B$.

Definition 2.14 (First-Order Structure [5, Chapter 2]). *A first-order* structure *$\mathscr{M}$ for a given type hierarchy $\mathscr{T}$ and signature Σ consists of*

- *a domain (D, δ),*
- *an interpretation I*

such that

1. *$I(f)$ is a function from $D^{A_1} \times \cdots \times D^{A_n}$ into D^A for $f : A_1, \ldots, A_n \to A$ in* FSym*,*
2. *$I(p)$ is a subset of $D^{A_1} \times \cdots \times D^{A_n}$ for $p(A_1, \ldots, A_n)$ in* PSym*,*
3. *$I(\doteq) = \{(d, d) \mid d \in D\}$.*

Definition 2.15 (Java DL Kripke Structure [5, Chapter 3]). *Let Prg be a Java program, $\mathscr{T}$ a type hierarchy for Prg and Σ a signature w.r.t. $\mathscr{T}$. A* Java DL Kripke structure *for Σ is a tuple*

$$\mathscr{K} = (\mathscr{S}, \varrho)$$

consisting of

- *an infinite set $\mathscr{S}$ of first-order structures over Σ (Def. 2.14), called* states*, such that:*
  - *Any two states $s_1, s_2 \in \mathscr{S}$ coincide in their domain and in the interpretation of predicate and rigid function symbols.*
  - *$\mathscr{S}$ is closed under the above property, i.e., any FOL structure coinciding with the states in $\mathscr{S}$ in the domain and the interpretation of the non-rigid function symbols is also in $\mathscr{S}$.*
- *a function ϱ that associates with every legal program fragment p a* transition relation *$\varrho(p) \subseteq \mathscr{S}^2$ such that $(s_1, s_2) \in \varrho(p)$ iff p, when started in s_1, terminates normally in s_2 (i.e., not by throwing an exception). (Java programs are considered to be deterministic, so for all legal program fragments p and all $s_1 \in \mathscr{S}$, there is at most one s_2 such that $(s_1, s_2) \in \varrho(p)$.)*

For Java DL, the transition relation ϱ is informally specified by the Java language specification [60].

A Java DL term $t \in \mathrm{Trm}_A$ is evaluated by an evaluation function $val_{\mathscr{K},s,\beta}(t)$ where *Prg* is a Java program, $\mathscr{T}$ a type hierarchy for *Prg*, Σ a signature w.r.t. $\mathscr{T}$, $\mathscr{K} = (\mathscr{S}, \varrho)$ a Kripke structure for Σ, $s \in \mathscr{S}$ a state, and $\beta : \mathrm{VSym} \to D$ a variable assignment (Definition 2.16). The definition of $val_{\mathscr{K},s,\beta}(t)$ is not relevant for this thesis and can be found in [5, Chapter 2 and Chapter 3].

Definition 2.16 (Variable Assignment [5, Chapter 2]). *Let $\mathcal{M}$ be a first-order structure with universe D.*
A variable assignment *is a function $\beta : \text{VSym} \to D$ such that $\beta(v) \in D^A$ for $v : A \in \text{VSym}$.*

A Java DL formula $\phi \in \text{Fml}$ is considered to be true w.r.t. $\mathcal{K}$, s and β, denoted by $(\mathcal{K}, s, \beta) \models \phi$. The definition of $\models$ can be found in [5, Chapter 2 and Chapter 3]. Satisfiability and validity of a formula are specified by Definition 2.17.

Definition 2.17 (Satisfiability and Validity of Java DL Formulas [5, Chapter 3]). *Let Prg be a Java program, $\mathcal{T}$ a type hierarchy for Prg, Σ a signature w.r.t. $\mathcal{T}$, and $\phi \in \text{Fml}$ a formula.*
ϕ is satisfiable *if there is a Kripke structure $\mathcal{K}$, a state $s \in \mathcal{S}$ and a variable assignment β such that $(\mathcal{K}, s, \beta) \models \phi$.*
ϕ is logically valid*, denoted by $\models \phi$, if $(\mathcal{K}, s, \beta) \models \phi$ for all Kripke structures $\mathcal{K}$, all states $s \in \mathcal{S}$ and all variable assignments β.*

2.3.4 Labeled Terms and Labeled Formulas

Labels are attached to terms or formulas to trace them in a proof tree (Definition 2.9). This requires to extend the Java DL signature by label symbols (Definition 2.18), to define labels (Definition 2.19) and to extend the syntax of terms (Definition 2.20) and formulas (Definition 2.21).

Definition 2.18 (Label Symbols). *The Java DL signature (Definition 2.3) is extended by the set of label symbols* LSym *consisting of label symbols $l : n$ where $n \in \mathbb{N}_0$ is the arity of l.*

Definition 2.19 (Label). *The set of labels* Lbl *consists of $l(p_1, \ldots, p_n)$ where $l : n \in \text{LSym}$. In general, parameters p_i, $i \in \{1, \ldots, n\}$ are untyped. However, specific classes of labels might restrict the parameters to certain types which do not necessarily coincide with types in $\mathcal{T}$. For labels with arity 0, l is written instead of $l()$.*

For a given $l \in \text{LSym}$, the set of all labels with label symbol l is denoted by Lbl_l.

Definition 2.20 (Labeled Term). *The set Trm_A of Java DL terms of type A, for $A \neq \bot$, is inductively defined by Clauses 1–4 as shown in the definition of Java DL terms (Definition 2.4) and extended by the new clause:*

5. *t«l» for all terms $t \in \text{Trm}_A$ and labels $l \in \text{Lbl}$.*

Definition 2.21 (Labeled Formula). *The set* Fml *of Java DL formulas is inductively defined by Clauses 1–6 as shown in the definition of Java DL formulas (Definition 2.5) and extended by the new clause:*

7. *ϕ«l» for all formulas $\phi \in \text{Fml}$ and labels $l \in \text{Lbl}$.*

As labels do not influence the semantics of terms and formulas (Definition 2.22), the Java DL calculus remains sound (Proposition 2.1) and relatively complete (Proposition 2.2) in presence of labeled terms and formulas.

Definition 2.22 (Semantics of Labeled Terms and Formulas). *Labels do not influence the semantics of terms and formulas:*

- *Given a term t«l» $\in \text{Trm}_A$ labeled with label $l \in \text{Lbl}$, it is evaluated by $val_{\mathcal{K},s,\beta}(t\text{«}l\text{»}) = val_{\mathcal{K},s,\beta}(t)$.*
- *Given a formula ϕ«l» $\in \text{Fml}$ labeled with label $l \in \text{Lbl}$, $(\mathcal{K}, s, \beta) \models \phi$«$l$» is considered to be true iff $(\mathcal{K}, s, \beta) \models \phi$ is considered to be true.*

For each label exists a *propagation function* (Definition 2.23), which defines how labels are propagated by calculus rule applications.

Definition 2.23 (Propagation Function). *For each $l \in \text{LSym}$ exists a label specific* propagation function p_l *which defines how labels of kind l are propagated by rule applications.*

Please observe that a propagation function only changes labels and not the structure of formulas and terms.

In the context of this thesis, propagation functions are given by meta rule schemata which define how a rule propagates labels. Consider for instance label *DiamondModality* with arity 0. It is attached to diamond modalities $\langle\rangle\phi$ and should stay with the modality when a rule performing symbolic execution is applied.

For instance, calculus rules performing symbolic execution are applied to a sequent of the form $\Gamma \Longrightarrow \mathcal{U}\langle\alpha\rangle\phi, \Delta$ where $\mathcal{U}$ is an update and α is the Java program modified by the rule application. The resulting sequent is usually of the form $\Gamma' \Longrightarrow \mathcal{U}'\langle\alpha'\rangle\phi, \Delta'$ where the update $\mathcal{U}$ is rewritten into $\mathcal{U}'$ and the Java program α is rewritten into α'. In addition, formulas might be added to Γ and Δ resulting in Γ' and Δ'. The meta rule schema

$$\Gamma \Longrightarrow \mathcal{U}\,(\langle\alpha\rangle\phi)\text{«DiamondModality»}, \Delta \quad \rightsquigarrow \quad \Gamma' \Longrightarrow \mathcal{U}'\,\big(\langle\alpha'\rangle\phi\big)\text{«DiamondModality»}, \Delta'$$

says, that if a calculus rule is applied to a labeled modality of the conclusion (left side of $\rightsquigarrow$) which results into a premiss (right side of $\rightsquigarrow$) of the given form, then the label will be added to the resulting modality in the premiss. No label propagation takes place in case that conclusion or premiss are not of the given form. For instance, if no modality is rewritten or if the rewritten modality does not have the *DiamondModality* label. In addition, no rule propagation takes place for rewrite rules. As they are of a different form, additional meta rule schemata are needed to maintain the *DiamondModality* label by rewrite rules.

The proof (2.6) shows, how the *DiamondModality* label is propagated according to the discussed meta rule schema.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overset{*}{\Longrightarrow} \text{true}}{}\ \text{closeTrue}}{\Longrightarrow \{\mathtt{x} := 1\}\big(\mathtt{x} \doteq 1\big)}\ \textit{One Step Simplification}}{\Longrightarrow \{\mathtt{x} := 1\}\big(\langle\rangle\big(\mathtt{x} \doteq 1\big)\big)\text{«DiamondModality»}}\ \text{emptyModality}}{\Longrightarrow \big(\langle\mathtt{x} = 1;\rangle\big(\mathtt{x} \doteq 1\big)\big)\text{«DiamondModality»}}\ \text{assignment} \tag{2.6}$$

As labels do not influence the semantics of formulas, it is desirable that the same sequent modulo labels is constructed whether labels are present or not. In Java DL, antecedent and succedent of a sequent are sets of formulas. Assume $\phi, \phi\text{«}l_1\text{»}, \phi\text{«}l_2\text{»}$ with $\phi \in \text{Fml}$ and $l_1 \neq l_2 \in \text{Lbl}$. A set constructed by adding these three formulas would result in set $\{\phi, \phi\text{«}l_1\text{»}, \phi\text{«}l_2\text{»}\}$ with cardinality 3. But adding the unlabeled versions of $\phi, \phi\text{«}l_1\text{»}, \phi\text{«}l_2\text{»}$ to a set results in the singleton set $\{\phi\}$. To ensure that antecedent and succedent do not contain formulas which only differ in labels, labels are not considered in the check whether a formula is already part of antecedent or succedent. In case that a formula is already present, it is replaced by a new formula in which all labels of the same kind are merged with help of the *label merge function* of Definition 2.24.

Definition 2.24 (Label Merge Function). *For each $l \in \text{LSym}$ exists a label specific* label merge function $c_l : \text{Lbl}, \text{Lbl} \to \text{Lbl}$ *which defines how the two labels given as argument are merged to the resulting label.*

Assume label *ID* with arity 1 where the argument is a set of integers. The label merge function $c_{ID} : \text{Lbl}_{ID}, \text{Lbl}_{ID} \to \text{Lbl}_{ID}$ is defined by $c_{ID}\,(ID(a), ID(b)) := ID(a \cup b)$ where a and b are sets of integers. An application of this label merge function can be seen in the right branch of proof (2.7) where formulas $p\text{«}ID(\{1\})\text{»}$ and $p\text{«}ID(\{3\})\text{»}$ are merged to $p\text{«}ID(\{1,\ 3\})\text{»}$.

$$\dfrac{\Longrightarrow p\text{«}ID(\{1\})\text{»}, q\text{«}ID(\{2\})\text{»} \quad \Longrightarrow p\text{«}ID(\{1,\ 3\})\text{»}}{\Longrightarrow p\text{«}ID(\{1\})\text{»}, q\text{«}ID(\{2\})\text{»} \wedge p\text{«}ID(\{3\})\text{»}}\ \text{andRight} \tag{2.7}$$

2.4 Eclipse

Eclipse is an integrated development environment (IDE) released in 2001 as an open source project. In 2011, Eclipse had an estimated market share of 65% in the Java IDE space and became with CDT the de facto standard developer IDE in the embedded and real-time operating system market [51]. Nowadays, Eclipse is packaged[1] for instance for Java, C/C++ or PHP development. In addition, a large number of extensions are available on the Eclipse market place[2].

In the following, Section 2.4.1 introduces the relevant parts of the Eclipse architecture needed for this thesis. Section 2.4.2 presents then the debug model which is extended by the Symbolic Execution Debugger (Section 6) for symbolic execution.

2.4.1 Architecture of the Eclipse Platform

Eclipse is a platform which is designed from the ground up to be extendable and to be used to build different applications. This is achieved by so called *plug-ins*. Each plug-in realizes one or multiple functionalities by using or extending other plug-ins. In order to be extendable, a plug-in can define so called *extension points*. An extension point consists of a unique name and a definition of the expected data, e.g. implementations of a given type. Once an extension point is defined, plug-ins can contribute to it by offering data of the expected form.

A simplified picture of the architecture of Eclipse is shown in Figure 2.14. Each component represents a set of plug-ins realizing one or multiple functionalities. The *Platform Runtime* is the only mandatory component which is responsible to discover plug-ins and to start them when needed.

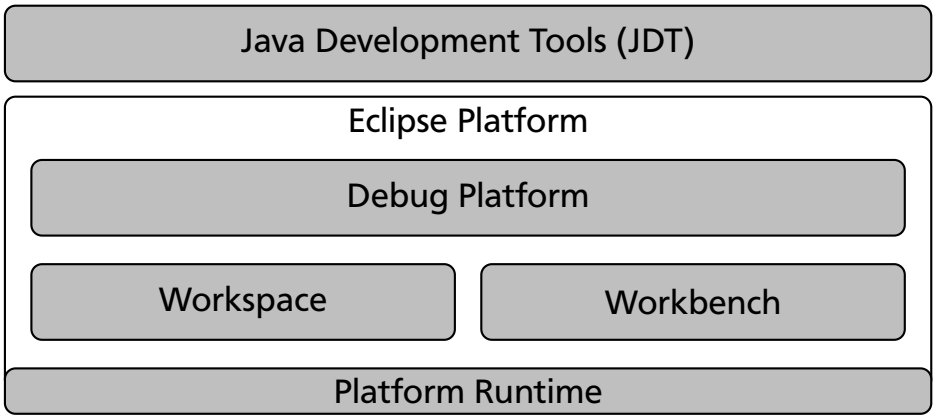


Figure 2.14.: Architecture of the Eclipse Platform [52, Simplified and Extended Version of Eclipse SDK]

Functionalities typically available in IDEs are offered by the *Eclipse Platform*. Relevant for this thesis are the *Workbench*, the *Workspace* and the *Debug Platform*. The workbench provides the typical user interface of Eclipse with perspectives, views and editors whereas the workspace allows one to manage resources in form of projects, files and folders. Language independent facilities to launch and to debug programs is offered by the debug platform.

Several extensions of the Eclipse platform are available. Most important for this thesis are the Java Development Tools (JDT)[3] which offer everything related to Java development. It includes for instance an editor for Java source code and a Java debugger.

[1] www.eclipse.org/downloads
[2] marketplace.eclipse.org
[3] www.eclipse.org/jdt

2.4.2 The Debug Model

The *Debug Platform*[4] offers language independent facilities to launch and to debug programs. This is achieved by a common debug model (shown in Figure 2.15) and a user interface which presents instances of the debug model to the user. In addition, the user interface allows one to manage breakpoints and to control program execution.

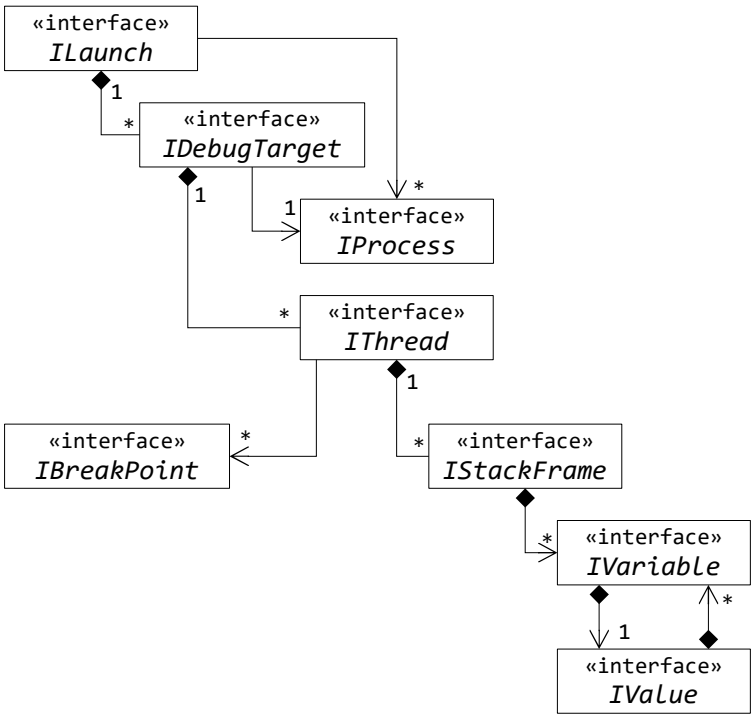


Figure 2.15.: The Debug Model [136, Simplified and Updated Version]

A launched program is represented by an `ILaunch` instance which offers access to the related processes (`IProcess` instances) and the debuggable execution contexts (`IDebugTarget` instances). Currently running threads are represented by `IThread` instances. If a thread is suspended, it allows one to inspect the stack trace modeled by `IStackFrame` instances. The part of the memory visible by a stack frame is modeled by hierarchical pairs of `IVariable` and `IValue` instances. In addition, a thread lists the breakpoints (`IBreakPoint` instances) causing its suspended state.

[4] www.eclipse.org/eclipse/debug

3 Symbolic Execution

Symbolic execution [28, 25, 84, 85] is a method to explore systematically *all* possible execution paths of a program for *all* possible input data. This property makes it into a powerful program analysis technique that is useful in a wide variety of application scenarios. Recently, symbolic execution enjoyed renewed interest and efficient implementations of symbolic execution engines for industrial programming languages are available (e.g. KeY [22] for Java, KLEE [30] for C or Pex [40] for .NET). From its very inception, symbolic execution has been employed in two fundamentally different scenarios: (i) state exploration for the purpose of, for example, test case generation or debugging [85] and (ii) formal verification of programs against a functional property [28]. For the latter, program annotations in the form of loop specifications and method contracts, are necessary. Recent verification approaches use contract-based specification languages specific to the target language for this purpose, such as the Java Modeling Language (JML) [93] or Spec# [15].

A main drawback of the second scenario is that meaningful contracts are often unavailable. But in fact also the first scenario cannot do without annotations in practice. Loops with symbolic bounds and method calls quickly render symbolic execution infeasible or allow one to explore only a fraction of all possible execution paths. As a remedy, compositional symbolic execution was proposed by Godefroid [57] where approximate contracts, so-called method summaries, are constructed on the fly.

This chapter presents a concept to explore *all* possible symbolic execution paths which unifies both strands of symbolic execution. This is achieved by integrating method contracts and loop specifications into a symbolic execution engine (Chapter 4) and rendering them visually in a suitable manner (Chapter 6).

The following Section 3.1 introduces symbolic execution before the use of specifications in form of loop specifications (Section 3.2) and method contracts (Section 3.3) is explained.

3.1 Symbolic Execution in General

Symbolic execution means to execute a program with symbolic values in lieu of concrete values. Method `sum` shown in Listing 3.1 for example takes an array `a` as argument and computes the sum of all of `a`'s elements. If the passed reference `a` is **`null`**, an exception is thrown.

```
1 public static int sum(int[] a) throws Exception {
2    if (a == null) {
3       throw new Exception("Array␣is␣null.");
4    }
5    else {
6       int sum = 0;
7       for (int i = 0; i < a.length; i++) {
8          sum += a[i];
9       }
10      return sum;
11   }
12 }
```

Listing 3.1: Sum of Array Elements

For a Java method to be executed it must be called explicitly. The expression `sum(new int[]{42})` for instance invokes method `sum` with a freshly created array of length one with content `42` as argument. This results in a single execution path. First, the guard of the **`if`**-statement is evaluated in line 2. It evaluates

to **false**, so execution continues with lines 6 to 9, where the sum is computed. Finally, execution finishes with the return statement in line 10.

Please observe that concrete execution (i) requires fixture code to set up a specific state and to call the method of interest with specific arguments, and (ii) results (assuming a deterministic programming language) in a single execution path through the program.

Executing the same method symbolically, instead with a concrete argument, the symbolic value $a0$ is given as argument. It can stand for any array object or **null**. The symbolic interpreter starts execution at line 2 and evaluates the guard of the conditional statement. As there is no information about the value of $a0$, one cannot rule out any of the two branches of the conditional statement. Consequently, the symbolic interpreter has to split execution into two continuations, (i) for the case where the condition $a0 = \textbf{null}$ is true, and (ii) for its complement. These conditions are called *branch conditions*.

Symbolic execution on branch (i) continues with the **throw**-statement, whereas on branch (ii) it is concerned with the sum computation. Executing the next statement on branch (ii) declares the local variable `sum` and initializes it with value `0`. The symbolic state maintained by the interpreter looks then as follows:

a:	$a0$	$\{a0 \neq \textbf{null}\}$
sum:	`0`	

The left column lists all relevant (i.e., until now accessed) locations such as local variables, fields and array elements (here: `a` and `sum`), whereas the second column shows their symbolic value. Concrete values are considered as special cases of symbolic ones. The third column lists possible constraints on symbolic values. These represent knowledge about a symbolic value where the knowledge was obtained either a priori from preconditions in specifications (see Section 3.2 and Section 3.3) or the knowledge was accumulated during symbolic execution from the branch conditions (here, $a0 \neq \textbf{null}$).

Now symbolic execution enters the loop: First the initializer of the counter `i` is executed. Then the loop guard `i < a.length` is tested, i.e., the interpreter attempts to determine whether `0 <` $a0$`.length` holds. Clearly, the current symbolic state represents concrete states, where the condition is true, but also concrete states, where it is false. Hence, two new execution branches are obtained with branch conditions (ii.1) $a0$`.length = 0`[1] and (ii.2) $a0$`.length > 0`. Each execution path is determined by the conjunction of all branch conditions that occur on it. For a given path, this conjunction is called *path condition*.

In branch (ii.1) the loop exits and the **return** statement is executed. In branch (ii.2) the loop is entered and its body is executed. After execution of the loop body, the loop condition is evaluated again, causing further branching. The branch conditions are: (ii.2.1) $a0$`.length = 1`, (ii.2.2) $a0$`.length > 1`, etc.

The symbolic execution tree up to here is shown in Figure 3.1. Each node consists of a top and an optional bottom compartment where the top compartment contains the statement to be executed *next* in the symbolic state shown in the bottom compartment. It lists the locations modified by its parent with name, symbolic value and optional constraints in braces. The complete symbolic state of a node consists of all variables defined in parent nodes with their most recent value and constraints. Branch conditions appear as annotations along the edges of branching nodes.

In contrast to concrete execution, symbolic execution does not require fixture code, but can start execution at any code position. Newly encountered locations are initialized with a fresh symbolic value about which nothing is assumed. Symbolic execution generates a *symbolic execution tree* that represents *all* possible concrete execution paths (up to a given depth). Each symbolic execution path through the symbolic execution tree may represent infinitely many concrete executions and is determined by its path condition. Symbolic execution may not terminate in presence of loops, e.g., when iterating over data structures with unbounded symbolic length. In standard symbolic execution method calls are handled

[1] The condition `!(i <` $a0$`.length)` is simplified to $a0$`.length = 0` because the concrete value of `i` is known and the length of arrays is nonnegative in Java.

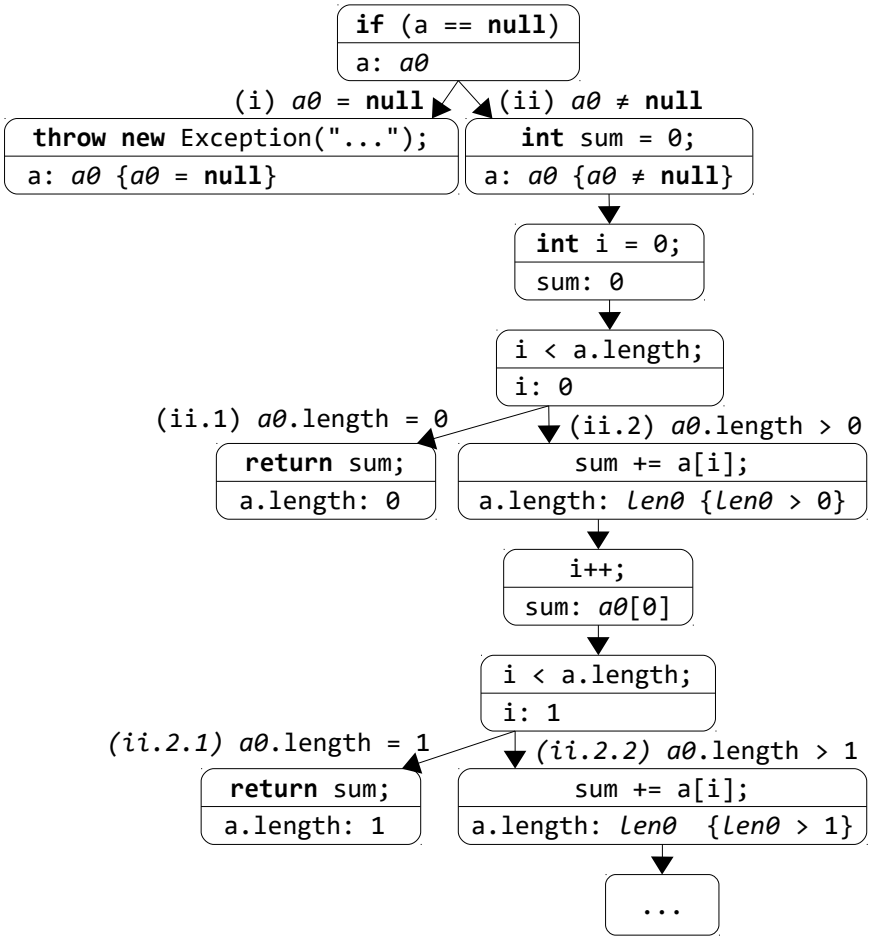


Figure 3.1.: Initial Section of Infinite Symbolic Execution Tree of `sum` from Listing 3.1

by inlining the method body of the callee. If multiple implementations for a method exist, symbolic execution creates one branch for each of them. Recursive methods exhibit the same problem as loops and can result in infinite symbolic execution trees.

The symbolic execution tree in Figure 3.1 does not contain all possible branches. For example, the one checking the array for being **`null`** during the access in line 8 is omitted, because it can be shown to be infeasible (its path condition is contradictory). Only feasible paths need to be explored.

3.2 Symbolic Execution using Loop Specifications

Symbolic execution of loops leads to infinite symbolic execution trees as discussed in Section 3.1. The solution to use loop invariants is also used in program verification to keep proof representations (proof trees) finite. To explain this better, how loops are handled in program verification is briefly discussed. For ease of presentation only programs that contain local variables but no fields or arrays are considered. For the same reason, termination is not considered. A full presentation is given by Beckert et al. [22].

Finite representation of all possible loop executions can be achieved either by induction or by loop invariants. The latter approach is chosen because it is technically somewhat simpler: assume proving that if a program is started in a state satisfying property P then in its final state property Q holds. In Hoare-style notation this is usually written as "$\{P\}$ `while (b) { body } rest;` $\{Q\}$".

A loop invariant I is a property that is satisfied on entering a loop and is preserved throughout the execution of the loop, i.e., it has to hold at the beginning and the end of each loop iteration. The loop invariant rule in Hoare notation is:

$$\text{loopInv}\ \frac{\begin{array}{ll} \vdash P \rightarrow I & (\textit{init}) \\ \{I \wedge \mathtt{b}\}\ \mathtt{body}\ \{I\} & (\textit{preserves}) \\ \{I \wedge \neg \mathtt{b}\}\ \mathtt{rest}\ \{Q\} & (\textit{use}) \end{array}}{\{P\}\ \textbf{while}\ \texttt{(b) \{ body \} rest;}\ \{Q\}}$$

Here, $\vdash \phi$ is a first-order problem to be discharged by first-order reasoning. The rule is applied bottom to top and splits a proof into three subgoals, where (*use*) marks the exit from the loop. The established loop invariant I is then used to prove the program `rest` following the loop.

Please observe that precondition P in the conclusion cannot be used to prove the (*preserves*) and (*use*) subgoals, because the execution of `body` might change the value of locations in P. This means that those parts of P that are not affected by `body` must be encoded into the loop invariant, which is clearly inefficient. For this reason, the set of locations that are modifiable in the loop body is tracked with a so-called *assignable* (or *modifies*) clause *mod*. In this case, P can be used to prove (*preserves*) and (*use*) provided the program is executed in a state where all knowledge about locations occurring in *mod* is wiped out. This can be achieved by a kind of skolemization. Details and the generalization to fields and arrays are given by Beckert et al. [22] and by Weiß [133]. See also Section 2.3.2 which explains symbolic execution using specifications in Java DL.

For the remainder of this chapter, termination is not considered. This simplifies the notion of a loop specification (Definition 2.12) to the pair $L = (I, mod)$ with I being a first-order formula and *mod* a set of locations. This is the partial correctness version of Definition 2.12.

Listing 3.2 shows a loop specification for the loop from Listing 3.1 in JML (see Section 2.1):

```
1 /*@ loop_invariant i >= 0 && i <= a.length &&
2   @                sum == (\sum int j; 0<=j && j<i; a[j]);
3   @ assignable sum, i; // In KeY: \strictly_nothing should be used, because local variables
4                        // which might be modified by the loop are added automatically.
5   @*/
6 for (int i = 0; i < a.length; i++) { ... }
```

Listing 3.2: Loop Specification of the `for`-Loop in Listing 3.1

The loop invariant limits the range of `i` and stipulates that `sum` is equal to the sum of all array elements visited so far. In addition, only local variables `i` and `sum` are allowed to be changed by the loop.

Loop specifications are used to render symbolic execution trees finite in presence of loops. Assume a symbolic execution path whose leaf n_l refers to a code position with a loop statement l as the next statement to be executed. This can be seen in Figure 3.2. Let $L = (I, mod)$ be the loop specification for loop l. Then, instead of unwinding the loop, two new symbolic execution branches with root nodes n_{it} and n_{exit} are created. The subtree rooted at n_{it} represents the symbolic execution of an arbitrary loop iteration and n_{it} refers to the first statement of the loop body to be executed next, whereas n_{exit} represents the execution of the program after exiting the loop.

The corresponding branch conditions are $I \wedge \mathtt{b}$ and $I \wedge \neg \mathtt{b}$ where I is the loop invariant as given by the loop specification L and `b` is the loop guard. The assignable clause of the loop specification is used to compute the symbolic states for the nodes n_{it} and n_{exit}. These symbolic states coincide with the state of n_l on all locations which are not contained in *mod* whereas the symbolic value of all locations contained in *mod* are replaced by fresh symbolic values. Please observe that there might already exist constraints on the fresh symbolic values. These constraints stem from the loop invariant and the branch condition.

The resulting symbolic execution tree for the array sum computation using the loop specification from Listing 3.2 is shown in Figure 3.3. The values of the local variables `i` and `sum` in the assignable clause have been replaced by fresh symbolic values *i0*, *sum0* and *i1*, *sum1*. The constraints shown in the bottom compartment of the nodes directly after the loop invariant node are obtained using the branch condition.

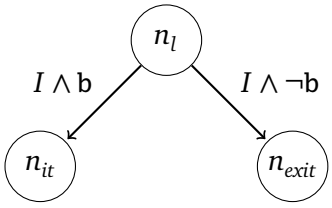


Figure 3.2.: Symbolic Execution Tree Construction with Loop Specifications

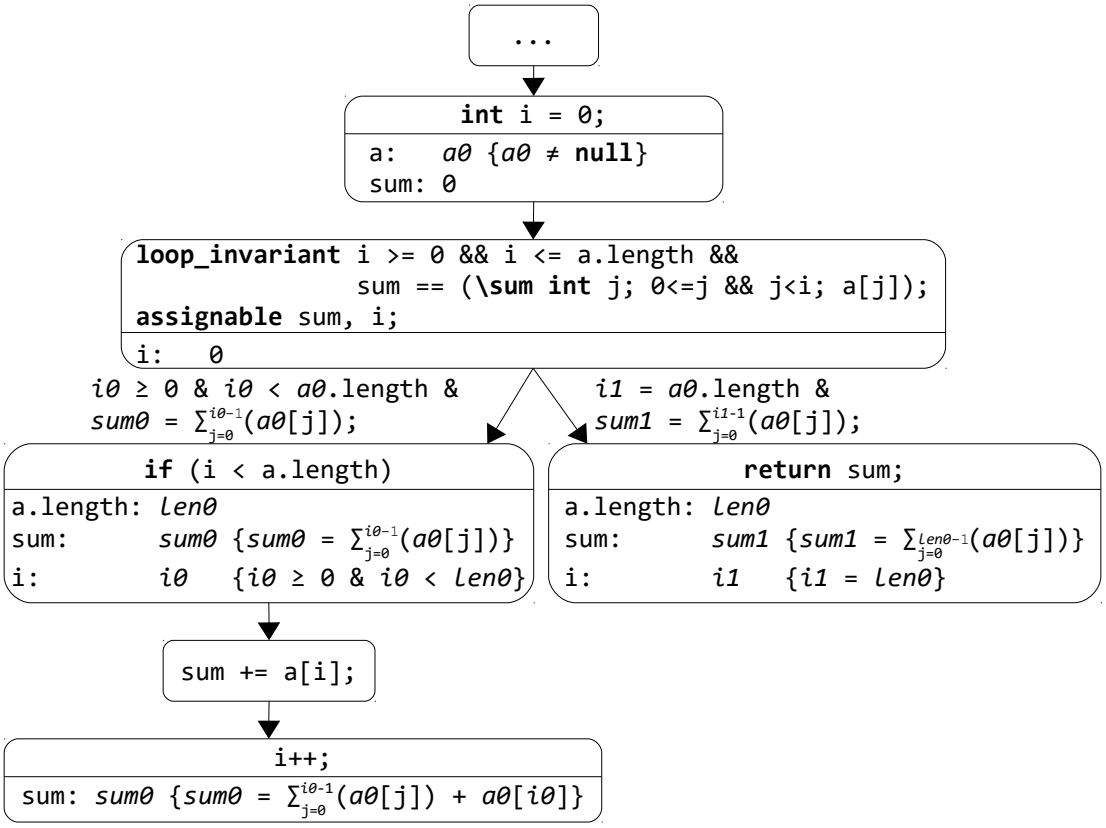


Figure 3.3.: Finite Symbolic Execution Tree of `sum` Using Loop Specification in Listing 3.2

Please observe that the loop guard has to be evaluated (indicated by the **`if`**-statement) in order to compute side effects.

Where do loop specifications come from? They can either be inferred automatically or simply provided by the user. Particularly in the latter case, the provided loop specification might be wrong or insufficient. One can always supply the trivial loop invariant **`true`**. A symbolic execution tree can be constructed regardless of whether the loop invariant and assignable clause are provable, but if not, then the corresponding node is marked accordingly. Whether it is acceptable to consider a possibly incorrect symbolic execution tree or not is a decision to be made by the application using the tree.

3.3 Symbolic Execution using Method Contracts

To apply a method contract, as known from program specification and verification, is an alternative to method inlining in symbolic execution. This has three major advantages: (i) symbolic execution becomes compositional and more robust against implementation changes, (ii) it becomes possible to create a finite symbolic execution tree for recursively defined methods with unbounded recursion depth, and (iii) the size of the symbolic execution tree stays manageable compared to inlining. The necessity

to have method contracts available is no principal limitation, because it is possible to use schematically generated, abstract contracts [64].

The notion of a method contract (Definition 2.11) is used in the sense of the *design by contract* specification paradigm [101]. Without considering termination, the termination marker of a method contract is not needed. This means a method contract $C_m := (R, E, mod)$ of a method m consists of a precondition formula R, a postcondition formula E, and an assignable clause mod containing the locations modifiable by m.

An implementation of a method m *satisfies* its contract if it guarantees that whenever m is called in a state satisfying R then E holds in the final state and it changes at most those locations contained in mod. There can be more than one method contract for a method.

JML permits to specify method contracts as structured Java comments. Listing 3.3 shows the contract for `sum`. It consists of two specification cases (separated by **`also`**). The normal behavior case is applicable if `a` is not **`null`** and ensures that the returned value is the sum over all array elements of `a`. In addition, nothing is allowed to be changed. The exceptional specification case is applicable if `a` is **`null`** and forces that an exception is thrown without changing anything else. Both specification cases can be encoded in terms of a single method contract. The details depend on the formalism used and are out of scope of this thesis.

```
1  /*@ normal_behavior
2    @ requires a != null;
3    @ ensures \result == (\sum int i; 0<=i && i<a.length; a[i]);
4    @ assignable \nothing;
5    @ also
6    @ exceptional_behavior
7    @ requires a == null;
8    @ signals (Exception e) true;
9    @ assignable \nothing;
10   @*/
11 public static int sum(/*@ nullable @*/ int[] a) throws Exception {...}
```

Listing 3.3: Specification of Method `sum`

A method contract can be used instead of method inlining to represent a method invocation within a symbolic execution tree. The idea is borrowed once again from program verification. The method contract application proof rule looks as follows (simplified for presentation purposes, e.g., ignoring null pointer exceptions and assignable clause):

$$\mathsf{mcApp}_{C_m} \frac{\vdash P \to R' \qquad \{E'\}\texttt{rest};\{Q\}}{\{P\}o.m(\bar{v});\texttt{rest};\{Q\}}$$

Here, the method contract is $C_m := (R, E, \_)$. R' and E' are constructed from R and E by replacing the **`this`** reference by o and the method parameters by the actual parameters $\bar{v}$. The first subgoal establishes that before the invocation of `m` on object o the precondition holds. The second subgoal assumes that the postcondition holds and continues execution of the remaining program `rest` following the method invocation.

In symbolic execution tree construction, method contracts can be used instead of inlining as follows: let n_{before} denote a node with an invocation of method m as the statement to be executed next. Similar as in loop specification, a node n_{after} is generated, which represents the symbolic state directly after the invoked method returns and before the next statement is executed. The edge between n_{before} and n_{after} is labeled with the precondition R of method m. The assignable clause and postcondition are used to update the symbolic state in n_{after} similar as in the use case of a loop specification. All locations

contained in the assignable clause are given a fresh symbolic value. Constraints on their values stem from the postcondition of the method contract.

A check whether the implementation of a method satisfies its contract is not done. This is a task that has to be performed independently using program verification or other means. In this sense a symbolic execution tree is only correct relative to the correctness of the used contracts. However, a check whether the precondition of a method is satisfied at invocation time is performed. If this check fails the invocation node is flagged. Another alternative would be to *generate* correct partial specifications by compositional symbolic execution and to insert them.

The node representing the applied contract may have several children, if there are several exclusive specification cases. In this case, the branch condition of each child refers to the precondition of the corresponding specification case. This can be seen in Figure 3.4 where the applied method contract consists of m specification cases.

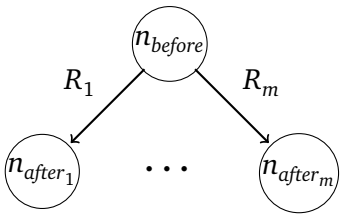


Figure 3.4.: Symbolic Execution Tree Construction with Method Contracts

Method contracts provide also a solution with respect to branching induced by dynamic dispatch. Method inlining branches over all possible method implementations as the exact dynamic type of the call cannot be determined in general. However, if Liskov's principle [99] is satisfied (JML enforces it by specification inheritance), branching can be avoided by using the method contract of the callee's static type.

```
1 public static int average(int[] a) throws Exception {
2    try {
3       int sum = sum(a);
4       return sum / a.length;
5    }
6    catch (Exception e) {
7       throw new Exception("Can't␣compute␣average.");
8    }
9 }
```

Listing 3.4: Average of Array Elements

Method `average` shown in Listing 3.4 extends the previous example. It computes the average of all array values by invoking `sum`. The initial segment of the resulting symbolic execution tree in which the method specification from Listing 3.3 is applied is shown in Figure 3.5.

The first statement in the **`try`**-clause declares local variable `sum` and calls method `sum`, which is handled in the next node by using its contract. Since nothing is known about parameter `a`, execution splits into two branches. The left branch continues symbolic execution when the method terminates normally, whereas the right branch continues execution in case that an uncaught exception has been thrown during method execution. Symbolic execution is stopped just before the return statement in the left branch and before the throw statement in the right branch.

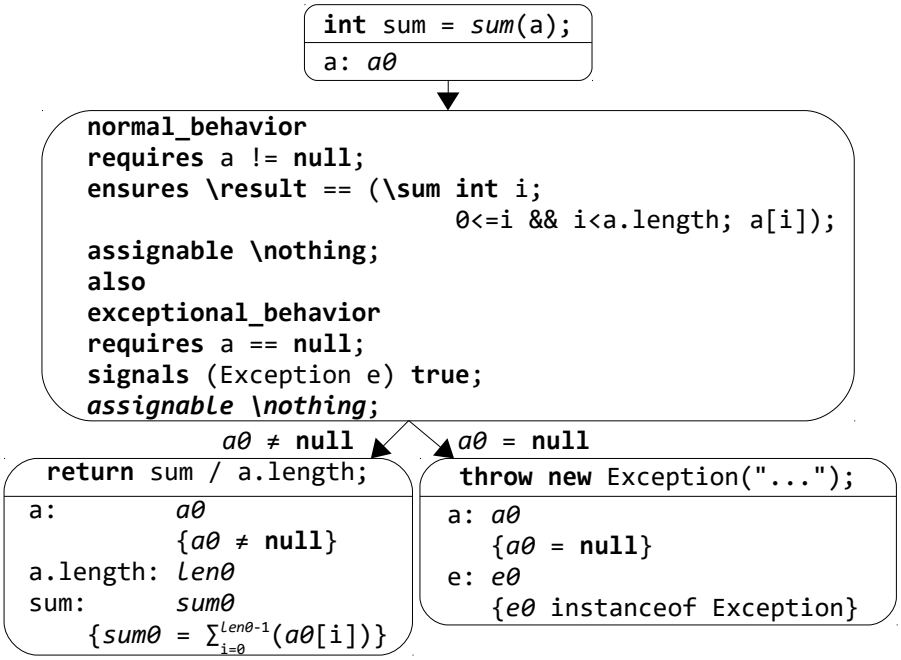


Figure 3.5.: Partial Symbolic Execution Tree of Method `average` Using Contract of `sum`

4 A Symbolic Execution Engine based on Proofs

Symbolic execution (Chapter 3) has nowadays a broad field of application areas (see Section 1.1) including for instance test case generation, formal verification, information flow analysis or model checking. Most tools come with their own symbolic execution engine, optimized for their specific needs. Only a few symbolic execution engines such as JPF–SE [10], a symbolic execution extension of Java PathFinder, are available as an API for general usage.

How execution paths are explored can be classified as static or dynamic symbolic execution. Static symbolic execution (e.g. [85]) interprets the code to explore feasible execution paths. In dynamic symbolic execution (e.g. [90]), the program is concretely executed and symbolic values are maintained in parallel. Whenever an alternative execution path is possible, execution is set back or started again with different input values to follow the alternative path as well.

Dynamic symbolic execution is usually preferred when robustness is required. Calls for which the source code is not available can be dealt with by just executing them. The price to be paid is that not all execution paths might be explored. Static symbolic execution on the other hand is more flexible. Execution can start at an arbitrary position and instead of executing a piece of code a specification can be applied instead.

The Java DL calculus (Section 2.3.2) contains rules to statically execute Java programs. This chapter presents a concept of how this calculus can be used to realize a general purpose symbolic execution engine. It is implemented on top of the KeY prover and called KeY's Symbolic Execution Engine.

In the following, Section 4.1 discusses how to generate a symbolic execution tree from a proof designed for symbolic execution before arbitrary verifications proofs are considered in Section 4.2. Then, Section 4.3 explains how branch and path conditions are computed. The computation of additional information like the call stack (Section 4.4), method return values (Section 4.5), the current state (Section 4.6) or a full memory layout (Section 4.7) are explained next. How undesired branching in proofs can be avoided is discussed in Section 4.8. Then, execution needs to be controlled which is explained in Section 4.9. Section 4.10 generalizes the concepts to support other languages and discusses requirements to built a similar symbolic execution engine based on a different verification tool. How the API offered by KeY's Symbolic Execution Engine can be used is shown in Section 4.11. Finally, Section 4.12 lists projects using KeY's Symbolic Execution Engine.

4.1 Symbolic Execution Tree Generation

In a Java DL proof verifying a Java program, application of rules performing symbolic execution is interleaved with application of rules performing logical reasoning. In the following, the rule applications constituting a proof tree are analyzed to generate a symbolic execution tree.

Each proof verifying a piece of Java code starts with an initial proof obligation of the form:

$$\Longrightarrow pre \rightarrow \mathcal{U} \left\langle \begin{array}{l} \texttt{try } \{codeOfInterest\} \\ \texttt{catch (Exception e) \{exc = e\}} \end{array} \right\rangle post$$

The meaning is as follows: assuming precondition *pre* holds, then executing the code of the diamond modality in symbolic state $\mathcal{U}$ terminates, and afterwards postcondition *post* holds. The catch-block around *codeOfInterest* is used to assign the caught exception to variable `exc` which can be used by the post condition to separate normal from exceptional termination. The code of interest is usually the initial method call but can be also a block of statements.

As discussed in Section 2.3.2, rules applied to a modality rewrite the first active statement to continue symbolic execution. Symbolic execution is performed at the level of atomic statements, such that complex Java statements and expressions have to be decomposed before they can be executed. For example,

the method call `even(2 + 3)` requires a simple argument expression, so that the sum must be computed before the method call can be performed. As a consequence, many intermediate steps might be required to execute a single statement of the source code. An empty modality can be removed, and the next step is then to show that the postcondition is fulfilled in the current proof context.

All symbolic execution rules have in common that, if necessary, they will split the proof tree to cover all conceivable execution paths. This means that the rules themselves do not prune infeasible paths. It is the task of the automatic proof strategy (or the user) to check the infeasibility of new proof premisses before execution is continued.

In order to realize a symbolic execution engine which analyzes a Java DL proof to generate a symbolic execution tree, the following tasks needs to be done:

1. Define a 'normal form' for proof trees that makes them suitable for generation of a symbolic execution tree.
2. Design a proof strategy that ensures proof trees to be of the expected shape.
3. Separate feasible and infeasible execution paths.
4. Identify intermediate proof steps stemming from decomposition of complex statements into atomic ones. Such intermediate steps are not represented in the symbolic execution tree.
5. Realize support for using specifications (Chapter 3) as an alternative to unwind loops and to inline method bodies.

It is important to postpone any splits of the proof tree not caused by the program's control flow (e.g. by an attempt to show the postcondition). Otherwise, multiple proof branches representing the same symbolic execution path might be created. While this does not affect the validity of a proof, it would cause redundant branches in a symbolic execution tree.

Also at most one modality formula per sequent is allowed, otherwise it is not clear what the target of symbolic execution is. Later, this condition has to be relaxed to support the use of specifications.

The standard proof strategy used by KeY for verification of Java programs almost ensures proof trees of the required shape. The only needed modifications are that (i) symbolic execution rules which introduce multiple modalities are forbidden for the moment, and that (ii) all rules not pertaining to symbolic execution and which cause splitting are only applied after finishing symbolic execution. Even with these restrictions the proof strategy is often powerful enough to close infeasible execution paths immediately.

A new feature of the strategy is to optionally check for aliased objects. If active, a case distinction $o_1 \doteq o_2$ (ignoring symmetry) is applied for each term o_1 and o_2 with a reference type, meaning that it represents an object in Java. This happens directly on the initial proof obligation and each time when a new name is introduced (in particular when a location is accessed the first time).

Whenever the strategy stops, symbolic execution tree generation takes place. During this it is required to separate proof branches representing a feasible execution path from infeasible ones. This information is not available in the proof itself, because it is not needed for proving. Complicating is also the fact that information not needed for verification might be eliminated, however, it might later be needed for symbolic execution tree generation. This can be easily solved with the following trick. The uninterpreted predicate *SET* is added to the postcondition of the initial proof obligation:

$$\Longrightarrow pre \rightarrow \mathcal{U}\left\langle \begin{array}{l} \texttt{try \{}codeOfInterest\texttt{\}} \\ \texttt{catch (Exception e) \{exc = e\}} \end{array} \right\rangle post \wedge SET(\texttt{exc})$$

The effect is that infeasible paths will be closed as before and feasible paths remain open since no rules exist for the predicate *SET*. Variables of interest are listed as parameters, so KeY is not able to remove them for efficiency if no longer needed.

To separate statements that occur in the source code from statements that are introduced by decomposition, metadata in the form of suitable tags is used.[1] Each statement occurring in the source code contains position information about its source file as well as the line and column where it was parsed. Statements introduced during a proof by decomposition have no such tags.

The mechanisms described above are sufficient to generate a symbolic execution tree (SET) by iterating over a given proof tree. Each node in a proof tree on an open path (infeasible paths are closed) is classified according to the criteria in Table 4.1 and added to the previously created symbolic execution tree node on the current proof tree path. Java API methods can optionally be excluded. In this case only method calls to non-API methods are added and statement nodes are only included if they are contained in non-API methods.

SET node type	Criterion in Java DL proof tree
Start	The root of the proof tree.
Method Call	The active statement is a method body statement.
Branch Statement	The active statement is a branch statement and the position information is defined.
Loop Statement	The active statement is a loop statement, the position information is defined, and it is the first loop iteration.
Loop Condition	The active statement is a loop statement and the position information is defined.
Statement	The active statement is not a branch, loop, or loop condition statement and the position information is defined.
Branch Condition	Parent of proof tree node has at least two open children and at least one child symbolic execution tree node exist (otherwise split is not related to symbolic execution).
Normal Termination	Rule *emptyModality* is applied and `exc` variable has value **`null`**.
Exceptional Termination	Rule *emptyModality* is applied and `exc` variable has not value **`null`**.
Method Return	A method return is performed by removing the current method frame and the related method call is part of the symbolic execution tree.

Table 4.1.: Classification of Proof Nodes for Symbolic Execution Tree Nodes (Excluding Specifications)

To decide whether the postcondition holds at a normal termination or exceptional termination node, the proof tree goals below it need to be considered. These nodes are marked as *not verified*, if at least one open goal exists in which *SET* is not a top level formula. Such open goals exist, because the postcondition of the modality is a conjunction containing the uninterpreted predicate.

To detect the use of specifications in the form of method contracts and loop specifications it is sufficient to check whether one of the rules *Use Operation Contract* (see Figure 2.12) or *Loop Invariant* (see Figure 2.13) is applied. The problem is that specifications may contain method calls, as long as these are side effect-free (so-called query methods). During the Java DL proof, these give rise to additional modalities in a sequent. Hence, it is required to separate such 'side executions' from the target of symbolic execution. This is again done with the help of meta information. Formula label *SE* is added to the modality of the proof obligation as follows:

$$\Longrightarrow pre \rightarrow \mathscr{U}\left(\left\langle \begin{array}{l} \texttt{try } \{codeOfInterest\} \\ \texttt{catch (Exception e) \{exc = e\}} \end{array} \right\rangle (post \wedge SET(\texttt{exc}))\right) \text{«SE»}$$

When symbolic execution encounters a modality with an *SE* label, it will be inherited to all child modalities. The proof strategy is modified to ensure that modalities without an *SE* label are executed first,

[1] The metadata is provided by the used parser and has nothing to do with term labels (Section 2.3.4).

because their results are required for the ongoing symbolic execution. Finally, during symbolic execution tree generation only rules applied to a modality with an *SE* label are considered.

A complication is that symbolic execution of modalities without an *SE* label may split the proof tree and the gained knowledge is used later when symbolically executing the target code. Such splits either have to be reflected in the symbolic execution tree or can be avoided as discussed in Section 4.8.

When a method contract is applied, two branches continue symbolic execution, one each for normal and an exceptional method return. Two additional branches check whether the precondition is fulfilled in the current state and whether the caller object is **`null`**. The latter two are proven without symbolic execution and their proof branches will be closed if successful. In case the proof branches remain open indicating that the precondition is not established or that the caller object might be **`null`**, the created method contract node is marked to indicate the *not verified* status, see Section 3.3. In the SED, marked nodes are crossed out (Section 6.6).

The situation is similar for the application of a loop specification. One proof branch checks whether the loop invariant is initially (at the start of the loop) fulfilled. A marker on the loop invariant node indicates its verified status. A second branch continues symbolic execution after the loop and a third branch is used to show that the loop invariant is preserved by loop guard and loop body. The latter is complex, because in case that an exception is thrown or that the loop terminates abnormally via a **`return`**, **`break`** or **`continue`**, the loop invariant does not need to hold. The loop invariant rule of Java DL (Figure 2.13) solves this issue by first executing loop guard and loop body in a separate modality. If this modality terminates normally, then the proof that the loop invariant holds is initiated. Otherwise, symbolic execution is continued in the original modality, without assuming that the invariant holds. As above, the problem of multiple modalities is solved by formula labels. A (proof global) counter is added to each *SE* label (Definition 4.1) to ensure that symbolic execution is continued according to the Java semantics. The label of the original proof obligation is *SE*(0) and it is incremented whenever a loop body needs to be executed. The proof strategy is modified to ensure that symbolic execution is continued in the modality with the highest counter first.

Definition 4.1 (Symbolic Execution Label)**.** *The* symbolic execution label *SE*(*id*) *is a label according to Definition 2.19 with* $id \in \mathbb{N}_0$.

The fact that a loop body is completely executed is indicated by a *loop body termination* node. A marker on this node is set if the loop invariant is not preserved. However, in case of an abrupt loop exit, execution is continued outside the loop and no loop body termination node is created. To separate both cases of termination, the loop invariant rule (Figure 2.13) introduces an implication. This implication is labeled with *LNT* (see Definition 4.2) as part of the symbolic execution label propagation (see Definition 4.4) and propagated according to Definition 4.3. A merge function is not needed, because the labeled implication has no parameters. A loop body termination node is created, if the left side of such a labeled implication evaluates to true[2].

Definition 4.2 (Loop Body Normal Termination Label)**.** *The* loop body normal termination label *LNT is a label according to Definition 2.19.*

Definition 4.3 (Loop Body Normal Termination Label Propagation Function)**.** *The propagation of the loop body normal termination label (Definition 4.2) is defined by the following meta rule schema:*

For a rewrite rule $\phi \to \psi \rightsquigarrow \phi' \to \psi'$ *with* $\phi, \phi', \psi, \psi' \in \mathrm{Fml}$*, the labels are propagated as follows:*

$$(\phi \to \psi)\,\text{«LNT»} \rightsquigarrow (\phi' \to \psi')\,\text{«LNT»}$$

The symbolic execution label propagation function is given by Definition 4.4. A label merge function is not needed, because rules duplicating a modality are forbidden.

[2] The full Loop Invariant is more complicated and rule impRight might be applied in order to evaluate the labeled implication. However, the strategy does this only after all scenarios with an abrupt termination are rejected. For this reason, a loop body termination node is also created, if rule impRight is applied on it.

Definition 4.4 (Symbolic Execution Label Propagation Function)**.** *The propagation of the symbolic execution label (Definition 4.1) is defined by the following meta rule schemata (box modality analogously):*

1. *For a rewrite rule* $\mathcal{U}\langle\alpha\rangle\phi \rightsquigarrow \mathcal{U}'\langle\alpha'\rangle\phi$ *with* $\mathcal{U}, \mathcal{U}' \in \text{Upd}$, $\phi \in \text{Fml}$, α, α' *are legal program fragments and* $id \in \mathbb{N}_0$*, the labels are propagated as follows:*

$$\mathcal{U}\left(\langle\alpha\rangle\phi\right)\text{«SE(id)»} \rightsquigarrow \mathcal{U}'\left(\langle\alpha'\rangle\phi\right)\text{«SE(id)»}$$

2. *For a rewrite rule* $\mathcal{U}\langle\alpha\rangle\phi \rightsquigarrow (\Gamma' \Longrightarrow \mathcal{U}'\langle\alpha'\rangle\phi, \Delta')$ *or a calculus rule* $\Gamma \Longrightarrow \mathcal{U}\langle\alpha\rangle\phi, \Delta \rightsquigarrow \Gamma' \Longrightarrow \mathcal{U}'\langle\alpha'\rangle\phi, \Delta'$ *with* $\phi \in \text{Fml}$, $\mathcal{U}, \mathcal{U}' \in \text{Upd}$, α, α' *are legal program fragments and* $id \in \mathbb{N}_0$*, the labels are propagated as follows:*

$$\Gamma \Longrightarrow \mathcal{U}\left(\langle\alpha\rangle\phi\right)\text{«SE(id)»}, \Delta \quad \rightsquigarrow \quad \Gamma' \Longrightarrow \mathcal{U}'\left(\langle\alpha'\rangle\phi\right)\text{«SE(id)»}, \Delta'$$

3. *For a rewrite rule* $\mathcal{U}\langle\alpha\rangle\phi \rightsquigarrow (\Gamma', \mathcal{U}'\langle\alpha'\rangle\phi \Longrightarrow \Delta')$ *or a calculus rule* $\Gamma, \mathcal{U}\langle\alpha\rangle\phi \Longrightarrow \Delta \rightsquigarrow \Gamma', \mathcal{U}'\langle\alpha'\rangle\phi \Longrightarrow \Delta'$ *with* $\phi \in \text{Fml}$, $\mathcal{U}, \mathcal{U}' \in \text{Upd}$, α, α' *are legal program fragments and* $id \in \mathbb{N}_0$*, the labels are propagated as follows:*

$$\Gamma, \mathcal{U}\left(\langle\alpha\rangle\phi\right)\text{«SE(id)»} \Longrightarrow \Delta \quad \rightsquigarrow \quad \Gamma', \mathcal{U}'\left(\langle\alpha'\rangle\phi\right)\text{«SE(id)»} \Longrightarrow \Delta'$$

4. *For an application of the simplified* Loop Invariant *rule:*

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta \\ \Gamma, \mathcal{U}\{\mathcal{V}(\mathit{Mod})\}\mathit{Inv} \Longrightarrow \mathcal{U}\{\mathcal{V}(\mathit{Mod})\,||\,\mathtt{v} := \mathit{Var}\} \\ \quad \left(\begin{array}{l}[\mathit{immf}(\pi)\{\mathtt{boolean\ b\ =\ e;}\}]\left(\mathtt{b} \doteq \mathit{TRUE}\right) \\ \quad \rightarrow \left(\begin{array}{l}\left\langle\begin{array}{l}\mathit{immf}(\pi)\{\mathtt{Throwable\ tExc\ =\ null;} \\ \mathtt{try\ \{if\ (}e\mathtt{)}\ p\mathtt{\}} \\ \mathtt{catch(Throwable\ t)\ \{tExc\ =\ t;\}\}}\end{array}\right\rangle \\ \left(\begin{array}{l}\left(\mathtt{tExc} \doteq \mathtt{null} \rightarrow (\mathit{Inv} \wedge \mathit{Var} < \mathtt{v} \wedge \mathit{Var} > 0 \wedge \mathit{Mod}_T)\right)\text{«LNT»} \\ \wedge\left(\mathtt{tExc}\ !\doteq \mathtt{null} \rightarrow (\langle\pi\ \mathtt{throw\ tExc;}\ \omega\rangle\phi)\text{«SE(id)»}\right)\end{array}\right)\end{array}\right)\text{«SE(id + 1)»}\end{array}\right), \Delta \\ \Gamma, \mathcal{U}\{\mathcal{V}(\mathit{Mod})\}\mathit{Inv} \Longrightarrow \begin{array}{c}\mathcal{U}\{\mathcal{V}(\mathit{Mod})\}[\mathit{immf}(\pi)\{\mathtt{boolean\ b\ =\ e;}\}] \\ \left(\mathtt{b} \doteq \mathit{FALSE} \rightarrow (\langle\pi\ \ \omega\rangle\phi)\text{«SE(id)»}\right)\end{array}, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}\left(\langle\pi\ \mathtt{while\ (}e\mathtt{)}\ p\ \ \omega\rangle\phi\right)\text{«SE(id)»}, \Delta}$$

There is one special case not covered yet. The proof branches that check (i) whether a loop invariant holds initially, (ii) whether a precondition holds, and (iii) whether the caller object is **null**, each can be proven without symbolic execution, as they contain no modality. This does not hold, however, when a loop specification or a method contract is applied on the branch that shows the invariant to be preserved by loop condition and loop body. The reason is that in this case the modality which continues symbolic execution in case of an abrupt loop exit is still present and the proof strategy is free to continue symbolic execution on it. As this execution is not of interest, all *SE* and *LNT* labels have to be removed from proof branches that only check the conditions listed above.

Listing 4.1 shows class `Example`, the running example of this chapter, where method `magic` is specified using JML (Section 2.1). The method contract requires that the method is called in a state where method parameters `a` and `b` are not **null** and ensures that the return value is 42 as well as that no exception is thrown. The missing assignable clause means that everything is allowed to be changed.

The related symbolic execution tree as visualized by the SED (see Section 6.1) is shown in Figure 4.1. It was generated from the simplified proof (4.2) continued in proof (4.1) by iterating over the proof nodes from 1 to 16. Each node is checked according to Table 4.1. The needed position information is

```
1  public class Example {
2     private int value;
3
4     /*@ normal_behavior
5       @ ensures \result == 42;
6       @*/
7     public static int magic(/*@ non_null @*/ Example a, /*@ non_null @*/ Example b) {
8        a.value = 42;
9        b.value = 2;
10       return a.value;
11    }
12 }
```

Listing 4.1: Simple Method to Demonstrate Symbolic Execution Tree Generation

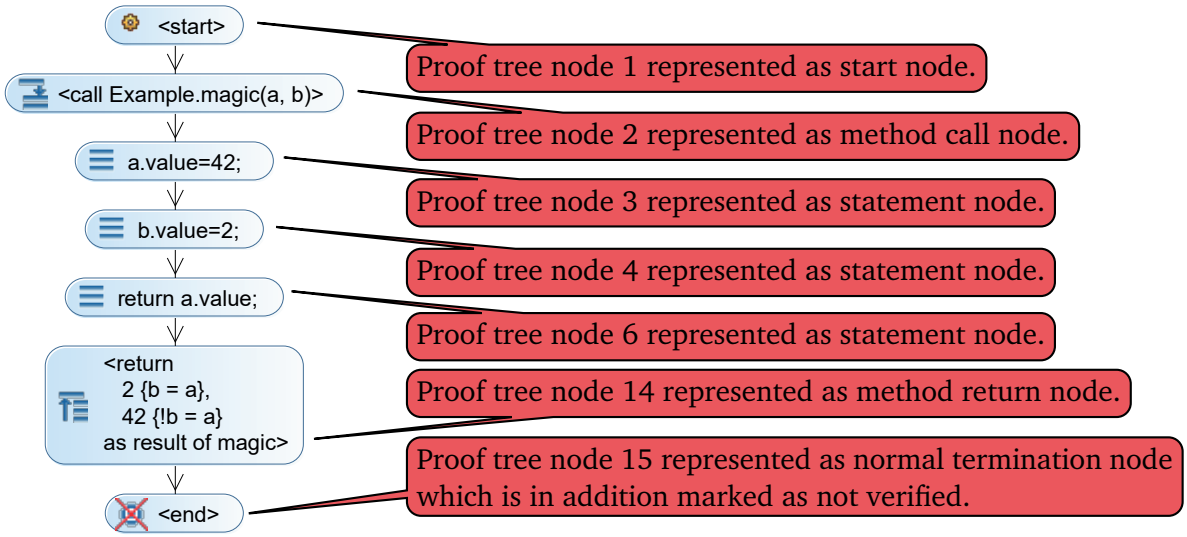


Figure 4.1.: Symbolic Execution Tree of Method `magic` as Visualized by the SED

Statement	Source	Line	Column
`result=Example.magic(a,b)@Example;`			
`a.value = 42;`	Listing 4.1	8	6
`b.value = 2;`	Listing 4.1	9	6
`return a.value;`	Listing 4.1	10	6
`int x = a.value;`			
`int x;`			
`x = a.value;`			
`return x;`			

Table 4.2.: Position Information of Statements in Proof (4.2) and Proof (4.1)

shown in Table 4.2. If a criteria matches, then a symbolic execution tree node is created and added to the previously created symbolic execution tree node on the current proof tree path. In Figure 4.1, the callouts refer to the proof tree node and mention the kind of the created symbolic execution node.


Further rule applications try to show the postcondition without success
16: ⟹ a ≐ null, b ≐ null, { heap := heap[a.value := 42][b.value := 2] || result := (if a ≐ b then 2 else 42) } (result ≐ 42 ∧ SET(a,b))
emptyModality
15: ⟹ a ≐ null, b ≐ null, { heap := heap[a.value := 42][b.value := 2] || result := (if a ≐ b then 2 else 42) } (⟨⟩(result ≐ 42 ∧ SET(a,b))) «SE(0)»
methodCallEmpty
14: ⟹ a ≐ null, b ≐ null, { heap := heap[a.value := 42][b.value := 2] || result := (if a ≐ b then 2 else 42) } (⟨method-frame():{}⟩(result ≐ 42 ∧ SET(a,b))) «SE(0)»
13: ⟹ a ≐ null, b ≐ null, { heap := heap[a.value := 42][b.value := 2] || x := (if a ≐ b then 2 else 42) } {result := x} (⟨method-frame():{}⟩(result ≐ 42 ∧ SET(a,b))) «SE(0)»
assignment
12: ⟹ a ≐ null, b ≐ null, { heap := heap[a.value := 42][b.value := 2] || x := (if a ≐ b then 2 else 42) } (⟨ method-frame():{ result = x;} ⟩(result ≐ 42 ∧ SET(a,b))) «SE(0)»
methodCallReturn
11: ⟹ a ≐ null, b ≐ null, { heap := heap[a.value := 42][b.value := 2] || x := (if a ≐ b then 2 else 42) } (⟨ method-frame(result->result):{ return x;} ⟩(result ≐ 42 ∧ SET(a,b))) «SE(0)»
10: ⟹ a ≐ null, b ≐ null, {heap := heap[a.value := 42][b.value := 2]} (a ≐ null), {heap := heap[a.value := 42][b.value := 2]} {x := a.value} (⟨ method-frame(result->result):{ return x;} ⟩(result ≐ 42 ∧ SET(a,b))) «SE(0)»
NPE Branch
assignment_read_attribute
9: ⟹ a ≐ null, b ≐ null, {heap := heap[a.value := 42][b.value := 2]} (⟨ method-frame(result->result):{ x = a.value; return x;} ⟩(result ≐ 42 ∧ SET(a,b))) «SE(0)»

(4.1)

$$\dfrac{\text{Continued in proof (4.1)}}{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}\mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null},\\ \{\mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42][\mathtt{b.value} := 2]\}\\ 8:\ \Longrightarrow \left(\begin{array}{l}\langle \mathtt{method\text{-}frame(result->result)\,:\{}\\ \mathtt{int\ x;\ x\ =\ a.value;\ return\ x;\}}\rangle\\ (\mathtt{result} \doteq 42 \land \mathit{SET}(\mathtt{a},\mathtt{b}))\end{array}\right)\end{array}}\,\text{«SE(0)»}}{\begin{array}{l}\mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null},\\ \{\mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42][\mathtt{b.value} := 2]\}\\ 7:\ \Longrightarrow \left(\begin{array}{l}\langle \mathtt{method\text{-}frame(result->result)\,:\{}\\ \mathtt{int\ x\ =\ a.value;\ return\ x;\}}\rangle\\ (\mathtt{result} \doteq 42 \land \mathit{SET}(\mathtt{a},\mathtt{b}))\end{array}\right)\end{array}}\,\text{«SE(0)»}}\,\text{variableDeclarationAssign}}{\begin{array}{l}\mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null},\\ \{\mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42][\mathtt{b.value} := 2]\}\\ 6:\ \Longrightarrow \left(\begin{array}{l}\langle \mathtt{method\text{-}frame(result->result)\,:\{}\\ \mathtt{return\ a.value;\}}\rangle\\ (\mathtt{result} \doteq 42 \land \mathit{SET}(\mathtt{a},\mathtt{b}))\end{array}\right)\end{array}}\,\text{«SE(0)»}}\,\text{returnUnfold}}{\begin{array}{l}\mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null},\\ \{\mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42]\}\\ \quad(\mathtt{b} \doteq \mathtt{null}),\\ \{\mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42]\}\\ 5:\ \Longrightarrow \quad \{\mathtt{heap} := \mathtt{heap}[\mathtt{b.value} := 2]\}\\ \qquad \left(\begin{array}{l}\langle \mathtt{method\text{-}frame(result->result)\,:\{}\\ \mathtt{return\ a.value;\}}\rangle\\ (\mathtt{result} \doteq 42 \land \mathit{SET}(\mathtt{a},\mathtt{b}))\end{array}\right)\end{array}}\,\text{«SE(0)»}}\,\text{assignment\_write\_attribute}}{\begin{array}{l}\mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null},\\ \{\mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42]\}\\ 4:\ \Longrightarrow \left(\begin{array}{l}\langle \mathtt{method\text{-}frame(result->result)\,:\{}\\ \mathtt{b.value\ =\ 2;}\\ \mathtt{return\ a.value;\}}\rangle\\ (\mathtt{result} \doteq 42 \land \mathit{SET}(\mathtt{a},\mathtt{b}))\end{array}\right)\end{array}}\,\text{«SE(0)»}}\,\text{assignment\_write\_attribute}}{\begin{array}{l}\mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null},\\ 3:\ \Longrightarrow \left(\begin{array}{l}\langle \mathtt{method\text{-}frame(result->result)\,:\{}\\ \mathtt{a.value\ =\ 42;\ b.value\ =\ 2;}\\ \mathtt{return\ a.value;\}}\rangle\\ (\mathtt{result} \doteq 42 \land \mathit{SET}(\mathtt{a},\mathtt{b}))\end{array}\right)\end{array}}\,\text{«SE(0)»}}\,\text{methodBodyExpand}}{\begin{array}{l}\mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null},\\ 2:\ \Longrightarrow \left(\begin{array}{l}\langle \mathtt{result=Example.magic(a,b)@Example;}\rangle\\ (\mathtt{result} \doteq 42 \land \mathit{SET}(\mathtt{a},\mathtt{b}))\end{array}\right)\end{array}}\,\text{«SE(0)»}}}{\begin{array}{l}(\neg(\mathtt{a} \doteq \mathtt{null}) \land \neg(\mathtt{b} \doteq \mathtt{null}))\\ 1:\ \Longrightarrow \rightarrow \left(\begin{array}{l}\langle \mathtt{result=Example.magic(a,b)@Example;}\rangle\\ (\mathtt{result} \doteq 42 \land \mathit{SET}(\mathtt{a},\mathtt{b}))\end{array}\right)\end{array}}\,\text{«SE(0)»}$$

(4.2)

4.2 Symbolic Execution Tree Generation from Verification Proofs

Section 4.1 explains how a symbolic execution tree can be generated from a proof fulfilling the defined 'normal form'. But is it also possible to generate a symbolic execution tree from an arbitrary verification proof? The main difference is that interactive rule application makes it impossible to assume a normal form. Also the uninterpreted predicate *SET* to separate feasible from infeasible paths is not available.

As a consequence, symbolic execution tree generation is performed as introduced in Section 4.1 on all branches as closed branches cannot be ignored anymore, because they might represent a feasible execution path on which the post condition is successfully proven. Formula labels can be safely used as they do not influence the applicability of rules.

The good news is that usually only a few rules are applied interactively and most rules are applied by the verification strategy (without the modifications discussed in Section 4.1). Interaction is usually only required after symbolic execution terminates to help KeY proving the postcondition. As the verification strategy also tries to close infeasible paths as soon as possible, no symbolic execution nodes are usually found on infeasible execution paths. In order to generate the symbolic execution tree, all proof nodes are now considered. In addition, the criterion for branch condition nodes is adjusted to consider closed paths as well. The new criterion is that the parent of the actual proof node has at least two children and at least one child symbolic execution node exists. Also the order in which multiple modalities are treated by the verification strategy usually follows the Java semantics. Finally, a normal termination or exceptional termination node is marked as not verified if and only if the proof tree below is not closed.

To conclude, yes it is possible to generate a symbolic execution tree from an arbitrary verification proof. The difference is, that often more branch condition nodes are contained. Despite that there is no guarantee that infeasible paths are correctly pruned and that the order of statements is according to the Java semantics, this allows one for instance to inspect proof attempts performed by KeY Resources (Section 7.4) using the SED (Chapter 6) from a developer's perspective.

4.3 Branch and Path Conditions

Applicability of a proof rule in Java DL generally depends only on the sequent it is applied to, not on other nodes in the proof tree. Consequently, Java DL does not maintain branch and path conditions during proof construction, because the full knowledge gained by a split is encoded in the child nodes. A branch condition can be seen as the logical difference between the current node and its parent and the path condition is simply the conjunction over all parent branch conditions or, in other words, the logical difference between the current node and the root node.

In Java DL, each rule written as taclet adds the branch condition as a new formula to the resulting sequent (`\add` clause). In addition, an existing formula might be replaced by a new one (`\replacewith` clause). Consequently, branch conditions are defined by:

$$\left(\bigwedge \textit{added antecedent formula}\right) \wedge \neg \left(\bigvee \textit{added succedent formula}\right)$$

Consider for instance proof tree node 5 of proof (4.1). Rule assignment_write_attribute is applied on parent node 4 which adds a formula to the succedent. The branch condition is thus $(\text{true}) \wedge \neg\left(\{\texttt{heap} := \texttt{heap}[\texttt{a.value} := 42]\}\left(\texttt{b} \doteq \texttt{null}\right)\right)$ which can be simplified to $\neg\left(\texttt{b} \doteq \texttt{null}\right)$. Unsurprisingly, to change the value of an instance field requires that the modified instance is not **`null`**.

Method contract and loop invariant rules are so complex that they cannot be expressed schematically in KeY with the help of taclets, but are computed. After applying a method contract the branch conditions contain the knowledge that the caller object is not **`null`** and that the conjunction of all preconditions (both for normal and exceptional termination) hold. The branch condition on the proof branch ensuring that a loop invariant is preserved (*Body Preserves Invariant*) is the conjunction of the loop invariant and the loop guard. The branch condition on the branch that continues symbolic execution after the loop (*Use Case*) is the conjunction of the loop invariant and the negated loop guard.

4.4 Symbolic Call Stack

In Java DL, the method call stack is encoded with help of method frames directly in the Java program of a modality. For each inlined method, a new method frame is added that contains the code of the method body to execute.

During symbolic execution tree construction the symbolic call stack has to be maintained. Whenever a method call node is detected, it is pushed onto the call stack. All other nodes remove entries from the maintained call stack until its size is equal to the number of method frames in their modality.

The branch of the loop invariant rule that checks whether the loop body preserves the loop invariant contains multiple modalities with different call stacks. The modality that executes only the loop guard and the loop body contains only the current method frame. All parent method frames are removed. This requires to maintain a separate call stack for each counter used in *SE* labels. Whenever a modality with a new counter is introduced, its call stack is initialized with the top entry from the call stack of the modality where the loop invariant was applied.

The call stack of the running example is shown in Figure 4.2. Each callout shows the call stack of a symbolic execution tree node as an ordered set.

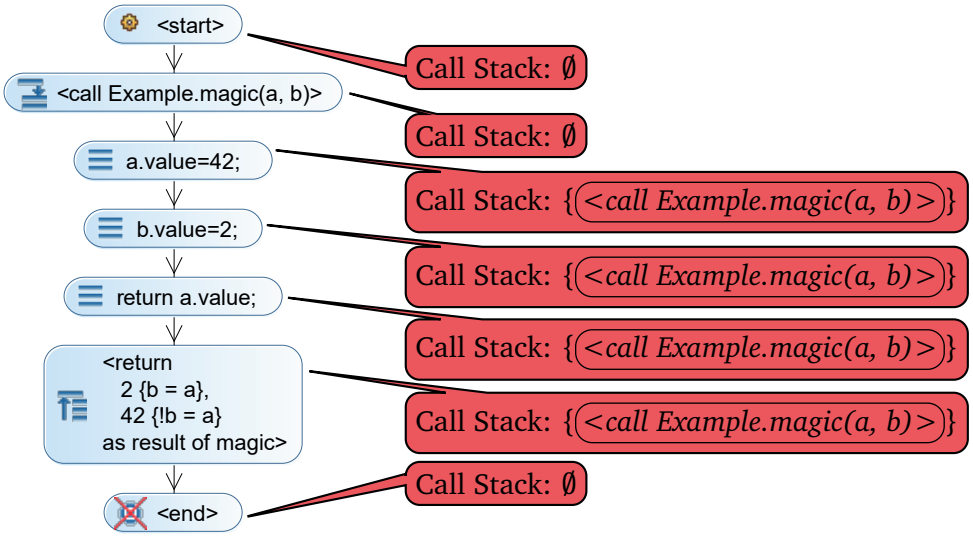


Figure 4.2.: Call Stack of each Symbolic Execution Tree Node of Method `magic`

4.5 Method Return Values

Method return nodes in a symbolic execution tree returning from non-void methods allow one to access return values. As usual in symbolic execution, return values are symbolic values. In case that the current state does not define a single return value, e.g. in case of aliasing, multiple return values are possible.

In Java DL, several rules are involved when returning from a method (see Section 2.3.2). Assuming that the argument of the **`return`** statement has been decomposed into a simple expression, rule methodCallReturn executes the return. For this, the rule adds an assignment statement that assigns the returned value to the result variable given in the current method frame. As the result variable is then no longer needed, it is removed from the method frame. A subsequent rule executes that assignment and yet another rule completes the method return by removing the by now empty method frame.

According to Table 4.1 a method return node is the proof tree node that removes the current method frame, say `cmf`. At this point, however, the name of the result variable is no longer available. This requires to go back to the parent proof tree node r, where rule methodCallReturn which assigns the returned value to the result variable of `cmf` was applied.

A side proof, not affecting the original proof, can be performed to compute returned values and the conditions under which they are valid. The proof obligation is:

$$\Gamma \Longrightarrow \mathcal{U} \left\langle \begin{array}{l} \texttt{cmf(result->resVar, ...):} \\ \texttt{return resExp;} \end{array} \right\rangle \mathit{ResultPredicate}(\texttt{resVar}), \Delta$$

The symbolic state $\mathcal{U}$ is that of return node r. Only the return statement is executed in the current method frame `cmf`. The postcondition is the uninterpreted predicate *ResultPredicate* that collects the computed result. Γ and Δ are all first-order top-level formulas of the sequent of r representing the context knowledge. After applying the standard verification strategy, each open branch represents a return value valid under its path condition (Section 4.3).

Please observe that in the context of this thesis, formulas containing a modality or a query are excluded from the context knowledge. Otherwise, a side proof would reason about the original proof obligation as well.

In the running example, the method return is performed by proof tree node 14 (see proof (4.1)). Proof tree node 11 is parent r, at which the rule methodCallReturn is applied. The performed side proof is shown in proof (4.3). The uninterpreted predicate *ResultPredicate* contains on each open goal a return value valid under the path condition. Therefore, 2 is returned in case that $\mathtt{a} \doteq \mathtt{b}$ holds and 42 otherwise.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{a} \doteq \mathtt{b} \Longrightarrow \mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null}, \mathit{ResultPredicate}(2) \quad \Longrightarrow \mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null}, \mathtt{a} \doteq \mathtt{b}, \mathit{ResultPredicate}(42)}\ \text{ifthenelse\_split}}{\Longrightarrow \mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null}, \mathit{ResultPredicate}(\text{if } \mathtt{a} \doteq \mathtt{b} \text{ then } 2 \text{ else } 42)}\ \textit{One Step Simplification}}{\Longrightarrow \mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null}, \left\{ \begin{array}{l} \mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42][\mathtt{b.value} := 2] \\ \quad \| \, \mathtt{x} := (\text{if } \mathtt{a} \doteq \mathtt{b} \text{ then } 2 \text{ else } 42) \end{array} \right\} \{\mathtt{result} := \mathtt{x}\} \mathit{ResultPredicate}(\mathtt{result})}\ \text{emptyModality}}{\Longrightarrow \mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null}, \left\{ \begin{array}{l} \mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42][\mathtt{b.value} := 2] \\ \quad \| \, \mathtt{x} := (\text{if } \mathtt{a} \doteq \mathtt{b} \text{ then } 2 \text{ else } 42) \end{array} \right\} \{\mathtt{result} := \mathtt{x}\} \langle\rangle \mathit{ResultPredicate}(\mathtt{result})}\ \text{methodCallEmpty}}{\Longrightarrow \mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null}, \left\{ \begin{array}{l} \mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42][\mathtt{b.value} := 2] \\ \quad \| \, \mathtt{x} := (\text{if } \mathtt{a} \doteq \mathtt{b} \text{ then } 2 \text{ else } 42) \end{array} \right\} \{\mathtt{result} := \mathtt{x}\} \langle \mathtt{method\text{-}frame()\,:\{\}} \rangle \mathit{ResultPredicate}(\mathtt{result})}\ \text{assignment}}{\Longrightarrow \mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null}, \left\{ \begin{array}{l} \mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42][\mathtt{b.value} := 2] \\ \quad \| \, \mathtt{x} := (\text{if } \mathtt{a} \doteq \mathtt{b} \text{ then } 2 \text{ else } 42) \end{array} \right\} \left\langle \begin{array}{l} \mathtt{method\text{-}frame()\,:\{} \\ \mathtt{result\ =\ x;\}} \end{array} \right\rangle \mathit{ResultPredicate}(\mathtt{result})}\ \text{methodCallReturn}}{\Longrightarrow \mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null}, \left\{ \begin{array}{l} \mathtt{heap} := \mathtt{heap}[\mathtt{a.value} := 42][\mathtt{b.value} := 2] \\ \quad \| \, \mathtt{x} := (\text{if } \mathtt{a} \doteq \mathtt{b} \text{ then } 2 \text{ else } 42) \end{array} \right\} \left\langle \begin{array}{l} \mathtt{method\text{-}frame(result->result)\,:\{} \\ \mathtt{return\ x;\}} \end{array} \right\rangle \mathit{ResultPredicate}(\mathtt{result})} \tag{4.3}$$

4.6 Current State

KeY's Symbolic Execution Engine allows one to compute for each symbolic execution tree node the visible variables and the related symbolic values on demand, see Section 4.11. Visible variables are the current **this** reference, method parameters, and variables changed by assignments. This includes local variables and static fields, but highlights also two differences as compared to Java:

- Java DL does not maintain local variables on the call stack. If a name is already in use it is renamed instead. As a consequence, the current state contains also local variables from all previous methods in the call stack.
- For efficiency, variables might be removed from symbolic states as soon as they are no longer needed. This means that a previously modified local variable may get removed if it is not used in the remaining method body. This can be seen for instance on proof tree node 14 (see proof (4.1)) where the value of `x` is removed because it is no longer needed.

Each variable can have multiple symbolic values caused by, for instance, aliasing, or because nothing is known about it yet. The values for a variable `loc`, together with the conditions under which they are valid, are computed in a side proof, similar as in Section 4.5. The proof obligation is

$$\Gamma \Longrightarrow \mathcal{U}\, \mathit{ResultPredicate}(\texttt{loc}), \Delta$$

where $\mathcal{U}$ is the update specifying the current state, *ResultPredicate* is an uninterpreted predicate and Γ and Δ are all first-order top-level formulas of the original sequent representing the context knowledge.

If a value is an object, then it is possible to query its fields in the same way. This brings the problem that it is possible to query fields about which no information is contained in the current sequent. Imagine, for instance, class `LinkedList` with instance variable `next` of type `LinkedList` and a sequent which says that `obj.next` is not **`null`**. When `obj.next` is now queried, its value will be a symbolic object. Since the value is not **`null`**, `obj.next.next` can be queried. But this time, the sequent says nothing about `obj.next.next`, consequently it could be **`null`** or not. In case it is not **`null`**, the query `obj.next.next.next` can be asked, etc. To avoid states with unbounded depth, the symbolic execution engine returns simply `<unknown value>` in case a field is not mentioned in the queried sequent.

Assume the value of the location `a.value` at proof tree node 14 (see proof (4.1)) should be computed. The simplified performed side proof is shown in proof (4.4). The uninterpreted predicate *ResultPredicate* contains on each open goal a value of `a.value` valid under the path condition. Therefore, the value is 2 in case that $\texttt{a} \doteq \texttt{b}$ holds and 42 otherwise.

$$\dfrac{\dfrac{\texttt{a} \doteq \texttt{b} \Longrightarrow \texttt{a} \doteq \texttt{null}, \texttt{b} \doteq \texttt{null},\ \mathit{ResultPredicate}(2) \qquad \Longrightarrow \texttt{a} \doteq \texttt{null}, \texttt{b} \doteq \texttt{null},\ \texttt{a} \doteq \texttt{b}\ \mathit{ResultPredicate}(42)}{\Longrightarrow \texttt{a} \doteq \texttt{null}, \texttt{b} \doteq \texttt{null},\ \mathit{ResultPredicate}\big(\text{if } \texttt{a} \doteq \texttt{b} \text{ then } 2 \text{ else } 42\big)}\ \text{ifthenelse\_split} \quad \vdots}{\Longrightarrow \left\{ \begin{array}{l} \texttt{heap} := \texttt{heap}[\texttt{a.value} := 42][\texttt{b.value} := 2] \\ \quad \Vert\, \texttt{result} := \big(\text{if } \texttt{a} \doteq \texttt{b} \text{ then } 2 \text{ else } 42\big) \end{array} \right\} \texttt{a} \doteq \texttt{null}, \texttt{b} \doteq \texttt{null},\ \mathit{ResultPredicate}(\texttt{a.value})} \tag{4.4}$$

Defining the current state by visible variables offers a view related to the source code. Alternatively, the current state could be defined as all locations and objects contained in the update (ignoring visibility). This offers a view related to verification with Java DL.

4.7 Memory Layouts

Aliased references do not necessarily result in different execution paths. One single symbolic execution path can represent many concrete execution paths with differently aliased references, corresponding to different data structures in memory. The symbolic execution engine allows one to compute for each node in the symbolic execution tree all possible aliasing structures and the resulting data structures in memory. Each equivalence class of variables referring to the same object, together with the resulting memory structure, is named a *memory layout*.

Memory layouts can be computed for the current state as well as for the initial state where the current computation started. The first step in doing this is to compute all possible equivalence classes of the current state. Based on this, it is then possible to compute the specific values resulting in the memory structure.

To compute the equivalence classes, the used objects occurring in the current sequent must be known. These are all terms with a reference type, meaning that they represent an object in Java, except those objects created during symbolic execution, and the variable `exc` in the initial proof obligation (Section 4.1). In Java DL, the *create* function (Section 2.3.2) is used to create new objects on a heap, so they can be easily filtered out. The constant **`null`** is also added to the used objects to check whether an object can be **`null`**.

After the used objects are identified, a side proof checks which of them can be aliases. The initial proof obligation is simply the current context knowledge $\Gamma \Longrightarrow \Delta$.

For each possible combination of two used objects o_1 and o_2 (ignoring symmetry), first a case distinction on $\mathcal{U}_{root}(o_1 \doteq o_2)$ is applied to all open goals of the side proof, then the verification strategy is started. The update $\mathcal{U}_{root}$ of the proof tree root is considered because it backs up the initial state and thus provides additional equality constraints.

This will close all branches representing impossible equivalence classes. The branch conditions (see Section 4.3) from the case distinctions on each open branch of the side proof represent the equivalence classes of a memory layout m.

Assume the equivalence classes at proof tree node 14 (see proof (4.1)) should be computed. The objects occurring in proof tree node 14 are `a`, `b` and `null`. The simplified performed side proof is shown in proof (4.5). Here, the update $\mathcal{U}_{root}$ is empty according to proof tree node 1. The open branches represent possible memory layouts determined by their path conditions. Therefore, the equivalence classes are $\{\neg(\mathtt{a} \doteq \mathtt{null}) \land \neg(\mathtt{b} \doteq \mathtt{null}) \land \mathtt{a} \doteq \mathtt{b}\}$ and $\{\neg(\mathtt{a} \doteq \mathtt{null}) \land \neg(\mathtt{b} \doteq \mathtt{null}) \land \neg(\mathtt{a} \doteq \mathtt{b})\}$.

$$\dfrac{\dfrac{\dfrac{}{\mathtt{b} \doteq \mathtt{null}\ \text{branch}}{}^{*} \qquad \dfrac{\mathtt{a} \doteq \mathtt{b} \Longrightarrow\ a \doteq \mathtt{null}, b \doteq \mathtt{null} \qquad \Longrightarrow\ a \doteq \mathtt{null}, b \doteq \mathtt{null}, \mathtt{a} \doteq \mathtt{b}}{\Longrightarrow\ a \doteq \mathtt{null}, b \doteq \mathtt{null}}\ \mathtt{cut} : \mathtt{a} \doteq \mathtt{b}}{\Longrightarrow\ a \doteq \mathtt{null}, b \doteq \mathtt{null}}\ \mathtt{cut} : \mathtt{b} \doteq \mathtt{null} \qquad \dfrac{\dfrac{}{\mathtt{a} \doteq \mathtt{null}\ \text{branch}}{}^{*} \qquad \Longrightarrow\ a \doteq \mathtt{null}, b \doteq \mathtt{null}}{\Longrightarrow\ \mathtt{a} \doteq \mathtt{null}, \mathtt{b} \doteq \mathtt{null}}\ \mathtt{cut} : \mathtt{a} \doteq \mathtt{null}} \tag{4.5}$$

The symbolic values of locations $\mathtt{loc}_1, \ldots, \mathtt{loc}_n$ can be queried for a memory layout m similar as in Section 4.6, but with the slightly modified proof obligation

$$\Gamma, cbc \Longrightarrow \mathcal{U}\mathit{ResultPredicate}(\mathtt{loc}_1, \ldots, \mathtt{loc}_n), \Delta$$

where *cbc* is the conjunction of the branch conditions from case distinctions on the path specifying m. As the case distinctions were exhaustive on all used objects, only a single value can be computed from this query. The side proof can be based either on the current node or on the root of the proof to inspect how the memory was before symbolic execution started.

The implementation does not query field by field to compute the full data structures of the memory. Instead, all variables used in the sequent are queried at once, which is achieved by adding them as parameters $\texttt{var}_1, \ldots, \texttt{var}_n$ to predicate *ResultPredicate*.

Assume the values of locations `a.value` and `b.value` at proof tree node 14 (see proof (4.1)) should be computed in the memory layout $\{\neg(\texttt{a} \doteq \texttt{null}) \land \neg(\texttt{b} \doteq \texttt{null}) \land \texttt{a} \doteq \texttt{b}\}$. The simplified performed side proof is shown in proof (4.6). The uninterpreted predicate *ResultPredicate* contains on the open goal the requested values which is 2 in both cases.

$$\dfrac{\texttt{a} \doteq \texttt{b} \Longrightarrow \begin{array}{l}\texttt{a} \doteq \texttt{null}, \texttt{b} \doteq \texttt{null},\\ \mathit{ResultPredicate}(2,2)\end{array}}{\neg(\texttt{a} \doteq \texttt{null}) \land \neg(\texttt{b} \doteq \texttt{null}) \land \texttt{a} \doteq \texttt{b} \Longrightarrow \begin{array}{l}\texttt{a} \doteq \texttt{null}, \texttt{b} \doteq \texttt{null},\\ \left\{\begin{array}{l}\texttt{heap} := \texttt{heap}[\texttt{a.value} := 42][\texttt{b.value} := 2]\\ \quad \mid\mid \texttt{result} := (\text{if } \texttt{a} \doteq \texttt{b} \text{ then } 2 \text{ else } 42)\end{array}\right\}\\ \mathit{ResultPredicate}(\texttt{a.value}, \texttt{b.value})\end{array}} \quad \vdots \tag{4.6}$$

4.8 Hiding the Execution of Query Methods

As pointed out in Section 4.1, the presence of query methods in specifications or in loop guards may spawn modalities that have nothing to do with the target code. These are used to compute a single value, such as a method return value or a boolean flag. Even though their execution is hidden in the symbolic execution tree, possible splits in the proof tree caused by them are visible, because the knowledge gained from them is used during subsequent symbolic execution. Such splits complicate symbolic execution trees and should be avoided.

These modalities have in common that they are top level formulas in a sequent that computes a single value *const* in the current symbolic state $\mathcal{U}$:

$$\mathcal{U}\langle \texttt{tmp = ...}\rangle \mathit{const} \doteq \texttt{tmp}$$

This computation can be optionally 'outsourced' from the main proof via a rule that executes the modality in a side proof. The initial proof obligation of the side proof is:

$$\Gamma \Longrightarrow \mathcal{U}\langle \texttt{tmp = ...}\rangle \mathit{ResultPredicate}(\texttt{tmp}), \Delta$$

It executes the modality in state $\mathcal{U}$ with an uninterpreted predicate called *ResultPredicate* as postcondition. That predicate is parametrized with variable `tmp`, which will be replaced during the proof by its computed value. Γ and Δ are all first-order top-level formulas of the original sequent, representing the context knowledge.

The standard verification strategy is used in the side proof. If it stops with open goals, where no rule is applicable, the results can be used in the original sequent.[3] Each open branch in the side proof contains a result *res* as parameter of the predicate *ResultPredicate*(*res*) that is valid relative to a path condition *pc* (Section 4.3). For each such open branch a new top-level formula is added to the sequent from which the side proof was outsourced. If the modality with the query method was originally in the antecedent, then $pc \rightarrow \mathit{const} \doteq \mathit{res}$ is added to the antecedent, otherwise, $pc \land \mathit{const} \doteq \mathit{res}$ is added to the succedent. The last step is to remove the original modality.

[3] The side proof is never closed, because the predicate in the postcondition is not provable. If the proof terminates, because the maximal number of rule applications has been reached, then the side proof is abandoned.

4.9 Controlled Execution

A proof strategy in KeY does not only decide which rule is applied next, but also selects the branch on which the next rule is applied and it decides when to stop rule application. The strategy used for verification applies rules on one branch until it is closed or no more rules are applicable. It continues then with another branch in the same way until the whole proof is closed or a user provided maximal number of rule applications is reached.

This behavior is not suitable for symbolic execution because a single path may never terminate. Instead, the symbolic execution strategy applies rules on a branch until the next rule application would generate a new symbolic execution tree node. Before that rule is applied, the strategy continues on another branch. When the next rule on all branches would cause a new symbolic execution tree node, the cycle starts over on the first branch. This ensures that one symbolic execution step at a time is performed on all branches. A preset number m of maximally executed symbolic execution tree nodes per branch is used as a stop condition in case that a symbolic execution tree has an unbounded depth.

If m is set to one, this corresponds to a *step into* instruction in a traditional debugger. A *step over* can be realized by stopping when a node with the same or lower stack trace size than the current one is encountered. A *step return* is even more strict and requires that the stack trace size is indeed lower.

More advanced stop conditions are available to realize breakpoints similar to interactive debuggers[4]:

- *Line Breakpoint*: Hit if the position information of the active statement equals the specified one.
- *Method Breakpoint*: Hit if a method call or return is performed according to Table 4.1 and the method is the specified one. It is also hit if the method contract of the specified method is applied.
- *Exception Breakpoint*: Hit if the active statement throws an exception.
- *Field Watchpoint*: Hit if the active statement writes or reads the specified field.
- *State Watchpoint* (called *KeY Watchpoint* in the SED): Hit if the position information of the active statement is defined and the specified condition c evaluates to true or is satisfiable.

The condition c of the state watchpoint is evaluated in a side proof. The proof obligation to check for true is $\Gamma \Longrightarrow \mathcal{U}\, c, \Delta$ and for satisfiable $\Gamma \Longrightarrow \mathcal{U}\, (\neg c), \Delta$. Γ and Δ are all first-order top-level formulas of the original sequent, representing the context knowledge. Condition c is true if the proof is closed and satisfiable if the proof remains open.

4.10 Generalization to Support other Languages or Systems

In Java DL the program to execute is contained in modalities and state changes are encoded by updates. Only the content within a modality and the rules which perform symbolic execution vary between different languages. This allows one to generalize the concept presented in this chapter about how to realize a symbolic execution engine on top of a Java DL proof. To support other languages only the classification of symbolic execution tree nodes in Table 4.1 needs to be adjusted.

A symbolic execution engine based on other verification tools can be realized similarly, as long as the following requirements are fulfilled:

1. Proof states performing symbolic execution are accessible (proof tree nodes in Java DL). This allows one to construct the symbolic execution tree.
2. The difference between proof states is accessible (a rule application in Java DL). This allows one to compute branch and path conditions.
3. The symbolic state of the program is accessible (updates in Java DL). This allows one to query the current state.

[4] Breakpoints in the SED and the KeYIDE were implemented as part of the Bachelor's thesis by Drebing [44].

4.11 Usage of KeY's Symbolic Execution Engine

KeY's Symbolic Execution Engine is realized and available[5] as a pure Java API. The first step to perform symbolic execution is to instantiate a proof. This is identical to verification with KeY. The only difference is that an additional `SymbolicExecutionTreeBuilder` instance needs to be created which will later generate the symbolic execution tree as discussed in this chapter. The steps are in detail:

1. Set taclet options (e.g. the default one for symbolic execution)
2. Load source code using the *Symbolic Execution Java Profile* (this contains the additional rules for symbolic execution, see Section 4.8)
3. Instantiate a proof (proof obligations from verification or new ones to execute a piece of Java code can be used)
4. Create the `SymbolicExecutionTreeBuilder` instance

After a proof is created, the symbolic execution strategy can be executed on given or all goals. Optionally, one or more stop conditions as discussed in Section 4.9 can be used. After the strategy stops, it is required to call `analyse` on the used `SymbolicExecutionTreeBuilder` instance to update the symbolic execution tree. The steps to continue symbolic execution are in detail:

1. Set symbolic execution strategy and the strategy settings to use on the proof (this includes to enable or disable the hiding of query methods, see Section 4.8)
2. Set stop condition to use on the proof (see Section 4.9)
3. Execute strategy on given or all goals of the proof
4. Call `analyse` on the used `SymbolicExecutionTreeBuilder` instance

The symbolic execution tree is represented by instances of `IExecutionNode`, as shown in Figure 4.3. The instances are maintained by the `SymbolicExecutionTreeBuilder` and updated each time when the `analyse` method is called. This realizes the generation of symbolic execution trees according to Section 4.1 and Section 4.2. Different sub types of `IExecutionNode` are used to represent the different kind of nodes as classified by Table 4.1. Only some information like the call stack (Section 4.4) are computed immediately during symbolic execution tree generation. Most of the information, especially that which requires a side proof, is computed lazily when requested the first time. Examples of lazily computed information are the path condition (Section 4.3), the current state (Section 4.6), the method return values (Section 4.5) or the memory layouts (Section 4.7).

The current state is computed according to Section 4.6 and is represented by `IExecutionVariable` instances which provide one or multiple `IExecutionValue` instances. An `IExecutionValue` offers for instance the condition under which the value is valid and if available additional child `IExecutionVariable` instances representing instance fields.

Memory layouts are computed according to Section 4.7 and are represented by `ISymbolicLayout` instances. It provides access to (i) the `ISymbolicEquivalenceClass` instances where each lists aliased locations, (ii) to the state (`ISymbolicState` instance) with local variables and (iii) to the objects on the heap (`ISymbolicObject` instances). Local variables of the state and objects field might be associations pointing to an object (`ISymbolicAssociation` instance) or a fixed value (`ISymbolicValue` instance).

An `IExecutionMethodReturn` allows one to compute possible method return values (Section 4.5) which are represented by `IExecutionMethodReturnValue` instances. Each instance offers the symbolic return value and the condition under which it is returned.

[5] KeY's Symbolic Execution Engine is part of the nightly build, see www.key-project.org/download/#nightly

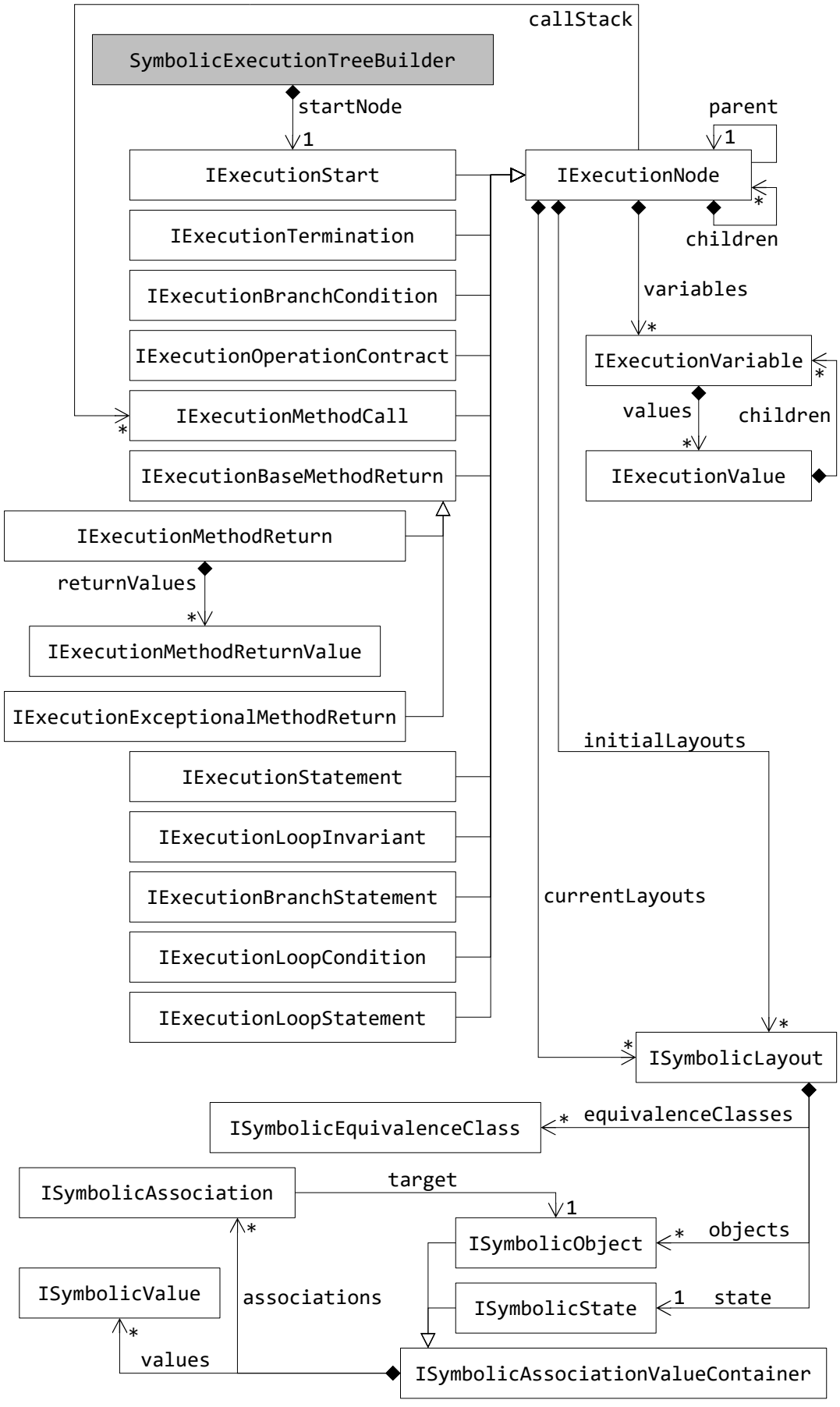


Figure 4.3.: Selected Types of KeY's Symbolic Execution Engine

4.12 Projects based KeY's Symbolic Execution Engine

In the following, projects using KeY's Symbolic Execution Engine are listed:

- Do et al. [43] use the symbolic execution engine in KEG[6] as backend to generate exploits for information flow leaks. After the code is symbolically executed, models of violated information flow policies are searched and represented as executable test cases. The usage of specifications guarantees finite symbolic execution trees and thus that the full program behavior is considered during exploit generation. Weak specifications which allow behavior which is not possible during program execution can be filtered out by executing the generated test case.
- Chimento et al. [32] combine static and runtime verification in the StaRVOOrS[7] project. They generate optimized assertions checked at runtime for branches KeY could not verify automatically. The implementation uses the symbolic execution engine instead of KeY directly, because it offers direct access to not verified termination states and to path conditions used to optimize the runtime assertions.
- The KeY system comes with a test case generation facility which is integrated into its user interface. A fully automatic test case generation tool could be built by using symbolic execution first to explore execution paths and then by using the existing functionality to generate test cases for a given proof tree node. A similar approach is realized by Svanefalk [119] who completely reimplemented the test case generation based on KeY's Symbolic Execution Engine.

[6] www.se.tu-darmstadt.de/research/projects/albia/download/exploit-generation-tool
[7] www.cse.chalmers.se/~chimento/starvoors

5 Proof Tree Analyses

How a symbolic execution tree can be generated from a proof is discussed in Chapter 4. The lessons learned are reused in this chapter to realize additional analyses of proofs. Labels are used to track a term or formula during rule applications like the modality of interest. Labeling formulas allows one to trace their use in a proof as discussed in Section 5.1. Once the nodes performing symbolic execution are identified, a slice can be computed containing only statements relevant to a given slicing criterion. How this works is explained in Section 5.2.

5.1 Truth Status Tracing

There are many situations when the user needs to understand a given proof attempt and has to decide how to continue. Such a situation occurs whenever a goal cannot be closed automatically, when a proof cannot be replayed after a change or if a proof attempt was performed by somebody else.

Understanding the current proof state is tedious and time consuming. The open goals are the result of many rule applications. The sequent of an open goal consists of additional constraints gained by previous rule applications and of the remaining parts of the initial proof obligation that have still to be shown to hold. Already evaluated parts of the proof obligation which do not help to close the proof branch were removed and are no longer part of the sequent. Also the shape of the remaining proof obligation formulas might have changed completely by previous rule applications. To find out from where a formula stems, the user has to inspect previous rule applications. Depending on the size of the verified code and the complexity of the specifications, a proof consists easily of many thousands of rule applications.

Each rule in Java DL is designed to have a clear semantics. Nearly all rules are written in the taclet [5, Chapter 4] language, a domain specific language to express rules. In KeY, an applied taclet is shown together with the sequent at which it is applied to the user. A few calculus rules require elaborate transformations of the program and Java model lookups that are too complicated to be expressed using the taclet language. Nevertheless, to help the verifiers understanding, the names of these rules have been chosen to be self-explanatory. Even if each rule has (with some practice) a clear semantics, this does not hold for the consecutive rule application performed by the proof search strategy. In the worst case, a user may need to prune the proof tree back to the last well understood state.

Beckert et al. [23] explain the difficulty of understanding proof attempts with the gap between the user's model of the proof and the system's current proof state. The user's model of the proof is the plan of the user about how to close the proof. The simplest user's model might be that KeY closes the proof automatically. But whenever interaction is required, the user needs to understand the current proof state. By inspecting the current goal and the applied rules the user tries to map the current state to her model of the proof. Beckert et al. [23] suggest that interactive theorem provers should (i) keep the gap small, (ii) bridge the gap and (iii) support interaction to increase usability.

Tracking the evaluated truth statuses of formulas of interest in the proof helps to bridge the gap between the user's model of the proof and the system's current proof state. A formula of interest might be the postcondition to verify, the check that the precondition of an applied method contract is fulfilled, or the checks that a loop invariant holds initially and is preserved. These formulas of interest have in common that they occur in the JML specifications (modulo some KeY specific representations). To know which parts of a specification are proven, do not hold, or were not evaluated is very helpful to understand the current proof situation (see Section 9.1). The truth status tracing is part of KeY's SED integration (see Section 6.7) and can be used in particular to inspect verification proofs performed by KeY or KeY Resources (see Section 7.4).

The following Section 5.1.1 discusses how formulas can be traced during rule application and Section 5.1.2 explains the truth status tracing in detail. How the SED presents the results to the user when KeY is used as symbolic execution engine is explained in Section 6.7.

5.1.1 Tracing Formulas

The prerequisite to trace the truth status of a formula is to trace the formula itself. Consider for instance Figure 5.1 which shows a rule application of andRight. Tracing formulas p and q from the premisses down to the conclusion should reveal that q in the left premiss is derived from q and $q \wedge p$ of the conclusion, as well as, that p in the right premiss stems from both p occurrences and $q \wedge p$ of the conclusion.

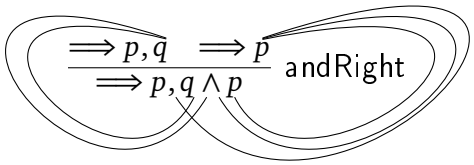


Figure 5.1.: Tracing of Formulas

Tracing of formulas is realized with the help of the *formula tracing label* according to Definition 5.1.

Definition 5.1 (Formula Tracing Label)**.** *A* formula tracing label *F(id, derivedIDs) is a label according to Definition 2.19 with*

- *label symbol* $F : 2 \in \text{LSym}$*,*
- *an identifier* $id \in \text{IDSym}$*,*
- *and a set* $derivedIDs \in 2^{\text{IDSym}}$ *not containing id.*

In the following, multiple IDs in derivedIDs are separated by a semicolon. In addition, F(id, $\emptyset$*) is written as F(id).*

Each formula of interest $\phi \in \text{Fml}$ is labeled with a formula tracing label *F(id)*, where *id* is a proof global unique identifier representing ϕ.

How labels are used in the example of Figure 5.1 is shown in proof (5.1).

$$\frac{\Longrightarrow p\text{«}F(1.0)\text{»}, q\text{«}F(2.0,\ 4.0)\text{»} \quad \Longrightarrow p\text{«}F(3.0,\ 4.0;1.0)\text{»}}{\Longrightarrow p\text{«}F(1.0)\text{»}, (q\text{«}F(2.0)\text{»} \wedge p\text{«}F(3.0)\text{»})\text{«}F(4.0)\text{»}}\ \text{andRight} \tag{5.1}$$

As the example shows, each formula has a consecutive number as ID. Labels of unchanged formulas keep their IDs, but a list of IDs from which the formula was derived from might be added. In the right branch *p«F(3.0, 4.0;1.0)»* represents the previous occurrence of *p«F(1.0)»* and *p«F(3.0)»*. It does not matter which ID is preserved, instead of *p«F(3.0, 4.0;1.0)»* also *p«F(1.0, 4.0;3.0)»* could be used.

The format of IDs is not important for traceability, but influences readability. In the context of this thesis, each ID $\in$ IDSym is of the form *majorID.minorID*. The *majorID* identifies a formula of interest

in the sequent where it occurs the first time. The *minorID* is needed to achieve uniqueness without any additional meaning. Whenever a new ID is required, it will have the same *majorID* as the formula it stems from, but with a different *minorID*. An example for rules whose application introduces new IDs is shown in proof (5.2). The applied rewrite rule replaces `self.<inv>` with the actual invariant $(\texttt{self.x} \geq 0) \land (\texttt{self.x} \leq 10)$ which requires to assign new IDs to newly introduced formulas.

$$\dfrac{\Longrightarrow ((\texttt{self.x} \geq 0)\,\text{«F(1.1)»} \land (\texttt{self.x} \leq 10)\,\text{«F(1.2)»})\,\text{«F(1.0)»}}{\Longrightarrow \texttt{self.<inv>}\,\text{«F(1.0)»}}\ \text{Class\_invariant\_axiom\_for\_...} \tag{5.2}$$

The up to now introduced tracing concept with formula tracing labels can be also used to trace terms. The application using the labels specifies when new IDs needs to be assigned and how the derived list is computed. The propagation function used by the truth status tracing (Section 5.1.2) to maintain labels during rule application is specified by Definition 5.2. The additional label merge function is defined by Definition 5.3.

Definition 5.2 (Formula Tracing Label Propagation Function)**.** *The propagation of the formula tracing label (Definition 5.1) is defined by the following meta rule schemata:*

1. *For a rewrite rule $\phi \rightsquigarrow \psi$ with $\phi, \psi \in \text{Fml}$, the labels are propagated as follows:*

$$\phi\text{«F(id, d)»} \rightsquigarrow \psi'\text{«F(id, d)»}$$

   where formula $\psi' \in \text{Fml}$ is constructed by recursively labeling all sub formulas ψ_C of ψ by ψ_C«F(freshID)». The formula ψ_C is only labeled with a fresh ID (freshID), if it does not have a formula tracing label.

2. *For a rewrite rule $\mathcal{U}\phi \rightsquigarrow (\Gamma' \Longrightarrow \Delta')$ and calculus rules $\mathcal{U}\phi \Longrightarrow$ or $\Longrightarrow \mathcal{U}\phi$ applied on $\Gamma \Longrightarrow \Delta$ with resulting premisses of the form $\Gamma' \Longrightarrow \Delta'$ and $\phi \in \text{Fml}$, $\mathcal{U} \in \text{Upd}$, the labels are propagated as follows:*

$$\mathcal{U}\phi\text{«F(id, d)»} \rightsquigarrow \forall_{\psi \in (\Gamma' \cap \Gamma)} \left(\mathcal{U}\psi'\text{«F(n(}\psi\text{), \{id\})»}\right) \Longrightarrow \forall_{\psi \in (\Delta' \cap \Delta)} \left(\mathcal{U}\psi'\text{«F(n(}\psi\text{), \{id\})»}\right)$$

   where formula $\psi' \in \text{Fml}$ is constructed by recursively labeling all sub formulas ψ_C of ψ by ψ_C«F(freshID)». The formula ψ_C is only labeled with a fresh ID (freshID), if it does not have a formula tracing label.

The function $n(\psi) : \text{IDSym}$ with $\psi \in \text{Fml}$ returns the ID of the formula tracing label of ψ if present, otherwise a fresh ID. A fresh ID is an ID not yet used in the proof.

Assuming a sound calculus, the given meta rule schemata define implicitly the propagation function (Definition 2.23).

Definition 5.3 (Formula Tracing Label Merge Function)**.** *The label merge function $c_F : \text{Lbl}_F, \text{Lbl}_F \to \text{Lbl}_F$ according to Definition 2.24 is defined by:*

$$c_F\,(\text{«F(}id_1\text{, }d_1\text{)»}, \text{«F(}id_2\text{, }d_2\text{)»}) := \text{«F(}id_1\text{, }d_1 \cup \{id_2\} \cup d_2\text{)»}$$

Definition 5.2 contains three design decisions discussed in the following. First, clause 1 conserves the previous ID instead of assigning a new one. This is not required, but helps a manual tracing of formulas by humans.

Second, only formula tracing labels of formulas involved in the current rule application are updated. Consequently, all other formula tracing labels remain untouched until a future rule application updates them. This can be seen for instance in proof (5.3) where label *F(4.0, 5.0)* introduced by impRight is not removed after application of rule andLeft.

Third, clause 2 removes the old derived list. The new derived list will list only formulas involved in the current rule application. This can be seen in proof (5.3) where ID *5.0* is not part of the derived list of $p\langle\!\langle F(1.0,\ 3.0)\rangle\!\rangle$ and $q\langle\!\langle F(2.0,\ 3.0)\rangle\!\rangle$ after application of andLeft.

$$\dfrac{\dfrac{p\langle\!\langle F(1.0,\ 3.0)\rangle\!\rangle, q\langle\!\langle F(2.0,\ 3.0)\rangle\!\rangle \Longrightarrow p\langle\!\langle F(4.0,\ 5.0)\rangle\!\rangle}{(p\langle\!\langle F(1.0)\rangle\!\rangle \wedge q\langle\!\langle F(2.0)\rangle\!\rangle)\langle\!\langle F(3.0,\ 5.0)\rangle\!\rangle \Longrightarrow p\langle\!\langle F(4.0,\ 5.0)\rangle\!\rangle}\ \text{andLeft}}{\Longrightarrow ((p\langle\!\langle F(1.0)\rangle\!\rangle \wedge q\langle\!\langle F(2.0)\rangle\!\rangle)\langle\!\langle F(3.0)\rangle\!\rangle \rightarrow p\langle\!\langle F(4.0)\rangle\!\rangle)\langle\!\langle F(5.0)\rangle\!\rangle}\ \text{impRight} \tag{5.3}$$

An alternative to the tracing with labels is to store references to all formulas involved in a rule application. This includes references to the instantiations in the conclusion, but also references to resulting formulas in all premisses. The drawback is the additional memory overhead. A proof in Java DL consists easily of hundreds of thousands of rule applications which already reaches the limit of modern desktop computers. As each rule is different, pointers would have to be organized in key-value-pairs like maps. The pointers and the additional objects for maps and map entries would thus increase memory consumption significantly.

Another alternative to tracing with labels would be to compute the effect of a rule application again just by re-applying the rule. If global proof counters are involved in the rule application, for instance to introduce new names, the result of the original and re-applied rule application will be different. Furthermore, the search of a formula of a given shape or at a given position in a child node is problematic as antecedent and succedent are unordered sets. In addition, KeY performs some built-in simplifications, if for instance the truth value of a formula is fixed (e.g. false $\wedge q$ is simplified to false), then it is directly used instead of the original formula. An implementation taking all these facts into account would be really complicated and vulnerable to changes.

5.1.2 Tracing Truth Statuses

The goal of the truth status analysis is to trace the truth status of a given formula on a path in the proof tree according to Definition 5.4.

Definition 5.4 (Truth Status Analysis)**.** *The* truth status analysis *returns for a given formula m the truth status evaluated on a given path p (down to a leaf of the proof tree). The truth status is a value in the three valued Kleene logic [87] with t (f) meaning that formula m is evaluated to true (false) on path p and u meaning that formula m is not (yet) completely evaluated on path p.*

In Java DL, rules rewriting a sequent are applied until a closing rule is applied which evaluates a sequent to true. Consider the example proof of $(p \wedge q) \rightarrow (q \wedge p)$ in proof (5.4). The left branch rewrites first q into true and applies then a closing rule. The right branch behaves similarly and rewrites p into false before closing the branch. Other formulas like p in the left branch or q in the right branch do not contribute to the proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{p,\ q \Longrightarrow \text{true}}\ \text{closeTrue}}{p,\ q \Longrightarrow q}\ \text{replace\_known\_left} \qquad \dfrac{\dfrac{*}{\text{false},\ q \Longrightarrow p}\ \text{closeFalse}}{p,\ q \Longrightarrow p}\ \text{replace\_known\_right}}{p,\ q \Longrightarrow q \wedge p}\ \text{andRight}}{p \wedge q \Longrightarrow q \wedge p}\ \text{andLeft}}{\Longrightarrow (p \wedge q) \rightarrow (q \wedge p)}\ \text{impRight} \tag{5.4}$$

With help of formula tracing labels (see Section 5.1.1) the truth status of a formula can be traced using Algorithm 5.1. The analysis can be performed at any time and consists of the following two steps:

1. Rule applications on the proof tree path are analyzed. The result is a mapping from IDs to their truth status if the formula is rewritten into true or false and otherwise an expression describing how to compute the truth status with help of other labeled formulas. This is done by function `analyzeRuleApplications` shown in Algorithm 5.2.

2. Truth statuses of IDs are queried in the mapping created by the first step. If the truth status is directly available, it is returned. Otherwise, the expression is used to compute it on the fly. This is done by function `lookup` shown in Algorithm 5.3.

Input : Proof tree leaf l, ancestor node n and ID of formula label id to look up.
Output: The truth status of formula labeled with ID id.

```
1 m ← analyzeRuleApplications(l, n);                    // Algorithm 5.2
2 return lookup(m, id);                                 // Algorithm 5.3
```

Algorithm 5.1: Algorithm to Trace the Truth Status of a Formula with a Given ID

KeY's SED integration traces truth statuses from the node at which formulas of interest are introduced down to all leaf nodes (see Section 6.7). Compared to Algorithm 5.1 which computes the truth status of a single labeled formula on one path, KeY's SED integration computes the truth statuses of all formula tracing labels on all paths at once for efficiency. This avoids redundant computations caused by the common prefix of different paths.

Algorithm 5.2 analyzes each rule application on the path from n to l. The result is a mapping of IDs to the evaluated truth status or alternatively to an expression describing how to compute the truth status with help of the other entries in the map. The iteration along the proof path from n to l is done by the while loop at line 2. The function `childOnPathToLeaf` returns the child of the proof tree node given as first argument which is on the path to the proof tree node given as second argument.

The task of the for each loop at line 4 is to compute for each formula involved in the current rule application an expression describing how the truth status can later be computed. Function `listInvolvedFormulaTermLabels` returns a set of all formula tracing labels involved in the current rule application. In Java DL, these are (i) all subformulas rooted at the rule application position, (ii) all enclosing formulas of the application formula, and (iii) the instantiations of the assumes clauses. For each found formula tracing label the expression is computed by `computeExp`. Let *at* (*st*) be all formulas in the antecedent (succedent) of child in which the ID of the formula tracing label is part of the derived list. The expression is defined as $\left(\bigwedge at\right) \wedge \left(\neg\left(\bigvee st\right)\right)$ if rule at n is applied in antecedent and $\left(\bigwedge at\right) \rightarrow \left(\bigvee st\right)$ otherwise. If *at* and *st* are empty, `computeExp` returns **null** instead of an expression. If an expression is computed, an entry mapping the ID of the formula tracing label to the expression is added to the result at line 7.[1] Please observe that later only the formula tracing label of each formula in *at* and *st* will be used. The formula structure is only maintained for efficiency reasons.

The **`if`**-statements at line 11 and line 16 check if the application formula is replaced by true or false. The check is realized by functions `isReplacedByTrue` and `isReplacedByFalse` which check if the applied rule replaces a formula by true and false, respectively. In Java DL, this can be seen in the `\replacewith` clause of the applied taclet. If the check is successful, the truth status is stored for all formula tracing labels IDs at the application position. The formula at the application position is returned by function `getApplicationTerm`.

The **`if`**-statement at line 21 checks if a closing rule is applied. If this is the case, for all formula tracing label IDs at the application position the truth status true is stored, if the application position is part of the succedent and false otherwise.[2]

[1] As derived lists are maintained until the next rule application rewrites the formula, it can happen that the same expression is computed multiple times.

[2] The check for closing rules is required and not covered by the check for rewriting into true or false as Java DL offers closing rules which close a branch without rewriting a formula into true or false.

Input : Proof tree leaf l and ancestor node n.
Output: A mapping of formula IDs to their evaluated truth statuses or expressions describing how to compute their truth statuses.

```
1  result ← ∅;
2  while n ≠ null do
3  |  child ← childOnPathToLeaf(n, l);
4  |  for each formulaTermLabel of listInvolvedFormulaTermLabels(n) do
5  |  |  expression ← computeExp(formulaTermLabel, child, isAppliedInAntecedent(n));
6  |  |  if expression ≠ null then
7  |  |  |  result ← result ∪ createExpressionEntry(formulaTermLabel, expression);
8  |  |  end
9  |  end
10 |  applicationTerm ← getApplicationTerm(n);
11 |  if isReplacedByTrue(n) then
12 |  |  for each formulaTermLabel of applicationTerm do
13 |  |  |  result ← result ∪ createTruthValueEntry(formulaTermLabel, true);
14 |  |  end
15 |  end
16 |  if isReplacedByFalse(n) then
17 |  |  for each formulaTermLabel of applicationTerm do
18 |  |  |  result ← result ∪ createTruthValueEntry(formulaTermLabel, false);
19 |  |  end
20 |  end
21 |  if isClosingRuleApplied(n) then
22 |  |  if isAppliedInAntecedent(n) then
23 |  |  |  for each formulaTermLabel of applicationTerm do
24 |  |  |  |  result ← result ∪ createTruthValueEntry(formulaTermLabel, false);
25 |  |  |  end
26 |  |  else
27 |  |  |  for each formulaTermLabel of applicationTerm do
28 |  |  |  |  result ← result ∪ createTruthValueEntry(formulaTermLabel, true);
29 |  |  |  end
30 |  |  end
31 |  end
32 |  n ← child;
33 end
34 return result;
```

Algorithm 5.2: Algorithm to Analyze Rule Applications for Truth Status Tracing

Consider for instance proof (5.4). The labeled left branch is shown in proof (5.5) and the labeled right branch in proof (5.6).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{p\text{«}F(0.0,\ 2.0)\text{»},\ q\text{«}F(1.0,\ 2.0)\text{»} \Longrightarrow \text{true}\,\text{«}F(3.0,\ 5.0)\text{»}}\,(5) \qquad \overset{*}{\vdots}}{p\text{«}F(0.0,\ 2.0)\text{»},\ q\text{«}F(1.0,\ 2.0)\text{»} \Longrightarrow q\text{«}F(3.0,\ 5.0)\text{»}}\,(4)}{p\text{«}F(0.0,\ 2.0)\text{»},\ q\text{«}F(1.0,\ 2.0)\text{»} \Longrightarrow (q\text{«}F(3.0)\text{»} \wedge p\text{«}F(4.0)\text{»})\,\text{«}F(5.0,\ 6.0)\text{»}}\,(3)}{(p\text{«}F(0.0)\text{»} \wedge q\text{«}F(1.0)\text{»})\,\text{«}F(2.0,\ 6.0)\text{»} \Longrightarrow (q\text{«}F(3.0)\text{»} \wedge p\text{«}F(4.0)\text{»})\,\text{«}F(5.0,\ 6.0)\text{»}}\,(2)}{\Longrightarrow ((p\text{«}F(0.0)\text{»} \wedge q\text{«}F(1.0)\text{»})\,\text{«}F(2.0)\text{»} \rightarrow (q\text{«}F(3.0)\text{»} \wedge p\text{«}F(4.0)\text{»})\,\text{«}F(5.0)\text{»})\,\text{«}F(6.0)\text{»}}\,(1) \tag{5.5}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\text{false}\,\text{«}F(0.0,\ 2.0)\text{»},\ q\text{«}F(1.0,\ 2.0)\text{»} \Longrightarrow p\text{«}F(4.0,\ 5.0)\text{»}}\,(7) \qquad \overset{*}{\vdots}}{p\text{«}F(0.0,\ 2.0)\text{»},\ q\text{«}F(1.0,\ 2.0)\text{»} \Longrightarrow p\text{«}F(4.0,\ 5.0)\text{»}}\,(6)}{p\text{«}F(0.0,\ 2.0)\text{»},\ q\text{«}F(1.0,\ 2.0)\text{»} \Longrightarrow (q\text{«}F(3.0)\text{»} \wedge p\text{«}F(4.0)\text{»})\,\text{«}F(5.0,\ 6.0)\text{»}}\,(3)}{(p\text{«}F(0.0)\text{»} \wedge q\text{«}F(1.0)\text{»})\,\text{«}F(2.0,\ 6.0)\text{»} \Longrightarrow (q\text{«}F(3.0)\text{»} \wedge p\text{«}F(4.0)\text{»})\,\text{«}F(5.0,\ 6.0)\text{»}}\,(2)}{\Longrightarrow ((p\text{«}F(0.0)\text{»} \wedge q\text{«}F(1.0)\text{»})\,\text{«}F(2.0)\text{»} \rightarrow (q\text{«}F(3.0)\text{»} \wedge p\text{«}F(4.0)\text{»})\,\text{«}F(5.0)\text{»})\,\text{«}F(6.0)\text{»}}\,(1) \tag{5.6}$$

After the first rule application (1) label *F(6.0)* is removed and its ID is contained in the derived lists of top level formulas of the premiss. As the rule is applied to the succedent, the expression to derive the truth status is $(p\text{«}F(0.0)\text{»} \wedge q\text{«}F(1.0)\text{»})\,\text{«}F(2.0,\ 6.0)\text{»} \rightarrow (q\text{«}F(3.0)\text{»} \wedge p\text{«}F(4.0)\text{»})\,\text{«}F(5.0,\ 6.0)\text{»}$. The second rule application (2) is applied to the antecedent. The not simplified expression of *F(2.0)* is $(p\text{«}F(0.0,\ 2.0)\text{»} \wedge q\text{«}F(1.0,\ 2.0)\text{»}) \wedge \neg\,\text{false}$ which can be simplified to $p\text{«}F(0.0,\ 2.0)\text{»} \wedge q\text{«}F(1.0,\ 2.0)\text{»}$.[3]

The rule application (3) is applied to the succedent and splits the proof. The unsimplified expression for *F(5.0)* in the left branch is $\text{true} \rightarrow q\text{«}F(3.0,\ 5.0)\text{»}$ which can be simplified to $q\text{«}F(3.0,\ 5.0)\text{»}$. The expression for the right branch is analog $p\text{«}F(4.0,\ 5.0)\text{»}$.

Rule applications (4) and (6) rewrite a term to true and false. Consequently, t is stored for *F(3.0)* on the left branch and f is stored for *F(0.0)* on the right branch. The closing rule applications (5) and (7) evaluate the terms again with the same truth statuses. The results are shown in Table 5.1 and Table 5.2.

ID	Truth Status	Expression
6.0		$(p\text{«}F(0.0)\text{»} \wedge q\text{«}F(1.0)\text{»})\,\text{«}F(2.0,\ 6.0)\text{»} \rightarrow (q\text{«}F(3.0)\text{»} \wedge p\text{«}F(4.0)\text{»})\,\text{«}F(5.0,\ 6.0)\text{»}$
2.0		$p\text{«}F(0.0,\ 2.0)\text{»} \wedge q\text{«}F(1.0,\ 2.0)\text{»}$
5.0		$q\text{«}F(3.0,\ 5.0)\text{»}$
3.0	t	

Table 5.1.: Result of Algorithm 5.2 Applied on Proof (5.5)

ID	Truth Status	Expression
6.0		$(p\text{«}F(0.0)\text{»} \wedge q\text{«}F(1.0)\text{»})\,\text{«}F(2.0,\ 6.0)\text{»} \rightarrow (q\text{«}F(3.0)\text{»} \wedge p\text{«}F(4.0)\text{»})\,\text{«}F(5.0,\ 6.0)\text{»}$
2.0		$p\text{«}F(0.0,\ 2.0)\text{»} \wedge q\text{«}F(1.0,\ 2.0)\text{»}$
5.0		$p\text{«}F(4.0,\ 5.0)\text{»}$
0.0	f	

Table 5.2.: Result of Algorithm 5.2 Applied on Proof (5.6)

After the (initial) mapping is computed by Algorithm 5.2, the truth status of a formula tracing label can be looked up using Algorithm 5.3.

[3] Derived lists of labels are not relevant anymore, but not removed to avoid additional term instances.

Input : Mapping m of formula IDs to truth statuses or expressions and ID of formula tracing label id to look up.
Output: The truth status of formula labeled with ID id.

```
1  truthValue ← getTruthValue(m, id);
2  if truthValue ≠ null then
3  |  return truthValue ;
4  else
5  |  expression ← getExpression(m, id);
6  |  if expression ≠ null then
7  |  |  result ← getUnknownTruthValue();
8  |  |  if isAnd(expression) then
9  |  |  |  result ← lookup(m, idOfChild(expression, 0)) ∧
   |  |  |   lookup(m, idOfChild(expression, 1));
10 |  |  end
11 |  |  if isOr(expression) then
12 |  |  |  result ← lookup(m, idOfChild(expression, 0)) ∨
   |  |  |   lookup(m, idOfChild(expression, 1));
13 |  |  end
14 |  |  if isImplication(expression) then
15 |  |  |  result ← lookup(m, idOfChild(expression, 0)) →
   |  |  |   lookup(m, idOfChild(expression, 1));
16 |  |  end
17 |  |  if isNot(expression) then
18 |  |  |  result ← ¬lookup(m, idOfChild(expression, 0));
19 |  |  end
20 |  |  m ← m ∪ createTruthValueEntry(id, result);
21 |  |  return result ;
22 |  else
23 |  |  return getUnknownTruthValue();
24 |  end
25 end
```

Algorithm 5.3: Algorithm to Look Up the Truth Status of a Formula Label

If the map contains a truth status for a given ID, it is directly returned at line 3. Otherwise, the truth status is computed with help of the expression at lines 8 to 21. Expressions are of the form $(\bigwedge at) \wedge (\neg(\bigvee st))$ or $(\bigwedge at) \rightarrow (\bigvee st)$ where st and at are labeled formulas. The only required operations in Kleene logic are thus $\neg$ (see Table 5.3), $\vee$ (see Table 5.4), $\wedge$ (see Table 5.5) and $\rightarrow$ (see Table 5.6) as the truth statuses of labeled terms can be looked up by recursively calling Algorithm 5.3 (`lookup`). To avoid that an expression is evaluated multiple times, the result is added to the map. IDs not contained in the map are treated as unknown because they are not considered by the proof (line 23).

$\neg$	
f	t
u	u
t	f

Table 5.3.: Truth Statuses of $\neg$

$\vee$	f	u	t
f	f	u	t
u	u	u	t
t	t	t	t

Table 5.4.: Truth Statuses of $\vee$

$\wedge$	f	u	t
f	f	f	f
u	f	u	u
t	f	u	t

Table 5.5.: Truth Statuses of $\wedge$

$\rightarrow$	f	u	t
f	t	t	t
u	u	u	t
t	f	u	t

Table 5.6.: Truth Statuses of $\rightarrow$

It can happen that a formula evaluates to true or false even if the truth status of sub formulas is unknown. Rules performing a cut are an example for such a situation. Consider for instance formula $a \wedge b$. Applying a cut with the same formula evaluates the $\wedge$ to true in one branch and to false in another. But the truth status of a and b in both branches remains unknown.

The results after looking up each ID without a truth status in Table 5.1 are shown in Table 5.7. Please observe that only parts of an expression colored in red are considered. The expression of ID *5.0* is q«*F(3.0, 5.0)*». Looking up the truth status of ID *3.0* reveals t. Looking up truth statuses of the expression of ID *2.0* reveals $u \wedge u$ as ID *0.0* and *1.0* are not contained in the table. $u \wedge u$ evaluates finally to u. Looking up truth statuses of the expression of ID *6.0* reveals $u \rightarrow t$ which evaluates to t. Similarly, looking up the remaining truth statuses of Table 5.2 produces the results shown in Table 5.8. How the SED presents these results to the user is discussed in Section 6.7.

ID	Truth Status	Expression
6.0	t	$(p$«*F(0.0)*»$\wedge q$«*F(1.0)*»$)$«*F(2.0, 6.0)*»$\rightarrow (q$«*F(3.0)*»$\wedge p$«*F(4.0)*»$)$«*F(5.0, 6.0)*»
2.0	u	p«*F(0.0, 2.0)*»$\wedge q$«*F(1.0, 2.0)*»
5.0	t	q«*F(3.0, 5.0)*»
3.0	t	

Table 5.7.: Truth Statuses of the Left Branch of Proof (5.5)

ID	Truth Status	Expression
6.0	t	$(p$«*F(0.0)*»$\wedge q$«*F(1.0)*»$)$«*F(2.0, 6.0)*»$\rightarrow (q$«*F(3.0)*»$\wedge p$«*F(4.0)*»$)$«*F(5.0, 6.0)*»
2.0	f	p«*F(0.0, 2.0)*»$\wedge q$«*F(1.0, 2.0)*»
5.0	u	p«*F(4.0, 5.0)*»
0.0	f	

Table 5.8.: Truth Statuses of the Right Branch of Proof (5.6)

5.2 Slicing

Weiser [132] introduced a *program slice* as a subset of a program which contains (only) the parts that are relevant to observe the behavior of a given *slicing criterion*. The slicing criterion are usually memory locations at a statement of interest. An overview of basic program slicing approaches and applications are given by Tip [124].

According to Weiser [131], a developer breaks a program into slices during debugging to follow the data of interest. In the context of debugging, Sridharan et al. [118] suggest *Thin Slicing* because a *thin slice* is usually smaller compared to a traditional slice and thus better suited for human comprehension. But compared to a traditional slice, there is no guarantee that the thin slice contains the desired statement. However, Sridharan et al. [118] found out that the desired statement is in many cases part of the slice or otherwise, usually located near to a statement in the slice. A thin slice consists only of *producer statements* which help to compute or assign the value to a location of the slicing criterion. Other statements explaining for instance why a producer statement is executed are excluded.

The following Section 5.2.1 applies slicing to proof trees. The applicability of slicing to symbolic execution trees created by KeY's Symbolic Execution Engine (Chapter 4) is then discussed in Section 5.2.2.

5.2.1 Slicing a Proof Tree

Slicing a proof tree is comparable to dynamic slicing, because in the Java DL calculus, the sequent at a proof tree node of interest contains the full proof context. In particular, the proof context contains the precondition and the path condition similar to symbolic execution (Chapter 3). The difference is that symbolic values are used in lieu of concrete values. Consequently, a taken path may represent infinitely many concrete executions paths. However, pointers might be aliased resulting in different memory layouts fulfilling the path condition. Aliasing can cause different slices, as for instance thin slicing considers only a field access on an object and not how the object is accessed.

The slicing criterion to slice a symbolic execution tree is defined as follows:

Definition 5.5 (Proof Tree Slicing Criterion)**.** *The* slicing criterion *of a proof tree is a triple* (m, n, c) *with the memory location* m *of interest at proof tree node* n *and an optional condition* c *specifying a memory layout.*

The memory layout condition c *lists all aliased memory locations and is of the form* $\bigwedge_{i,j} l_i \doteq l_j$ *where* l_i, l_j *are memory locations.*

A backward slice is created by traversing over the ancestors of n and consists of nodes m depends on. Furthermore, a forward slice is created by traversing over the descendants of n and consists of nodes depending on m.

An example of slicing criteria and the resulting backward slices is given in Figure 5.2. The proof symbolically executes method `magic` under precondition `a != null & b != null`. Applied rules performing a symbolic execution step are highlighted in blue.[4] Other proof tree nodes operate on first-order formulas and are not relevant for slicing. The slicing criteria SC_1 and SC_2 are interested in location `a.value` at proof tree node *29: return a.value+b.value*. The additional condition is used to separate the cases that method parameters `a` and `b` are aliased. Please observe that the memory layout condition `a` $\doteq$ `b` is only given for readability. The resulting backward slice of SC_1 consists only of proof tree node *15: a.value = 2* and the backward slice of SC_2 consists only of proof tree node *22: b.value = 3*.

[4] All highlighted nodes continue symbolic execution in the modality of interest.

Figure 5.2.: Proof Tree Slicing Example

The basic structure of an algorithm performing backward slicing of a proof tree is shown in Algorithm 5.4. A central aspect of the algorithm is the treatment of aliased pointers. Method `computeAliases` computes the aliased pointers. Two locations a and b are considered to be aliased, if

1. $\texttt{a} := \texttt{b}$ is part of the update at the application position,
2. $store(h, a_o, a_f, b)$ is latest assignment to location $a = a_o.a_f$ on heap $h = \texttt{heap}$ in the update at the application position[5],
3. $a \doteq b$ is a top level formula in the antecedent of n[6], or
4. $a \doteq b$ is part of the memory layout condition c.

[5] To deal with the *anon* function, a side proof as discussed later is used.

[6] It is assumed that the strategy ensures a 'normal form' between each symbolic execution step, see also Section 4.1

The result is a set of ordered sets in which each ordered set contains all aliased locations pointing to the same memory location. Instead of the analysis of a sequent a side proof (according to Section 4.7) could be used instead. Nevertheless, the analysis is sufficiently precise and comes with less overhead.

```
   Input  : A memory location m at proof tree node n and the memory layout condition c
   Output: A set of proof tree nodes representing the computed slice
 1 result ← ∅;
 2 relevantLocations ← {normalize(m, computeAliases(n, c))};
 3 oldAliases ← null ;
 4 while n ≠ null ∧ relevantLocations ≠ ∅ do
 5     if isSEStep(n) then
 6         aliases ← computeAliases(n, c);
 7         if accept(n, aliases, relevantLocations) then
 8             result ← result ∪ {n};
 9         end
10         if oldAliases ≠ null ∧ isAssignmentPerformed(n) then
11             relevantLocations ← renormalizeOutdatedLocations(n, aliases, oldAliases);
12         end
13         oldAliases ← aliases ;
14     end
15     n ← parent(n);
16 end
17 return result;
```

Algorithm 5.4: Basic Algorithm for Backward Slicing of a Proof Tree

To avoid that all aliased locations need to be considered when two locations are compared, a normalization is performed first. Method `normalize` gets as parameters the memory location to normalize and the available aliases. For an aliased location the representative is returned, which is the first entry in the ordered set. Otherwise, if the location is not aliased, the location itself is returned. If the location consists of multiple parts, the representative replacement is performed iteratively in the order of access.

Lines 1 to 3 initialize used variables. This includes to initialize the relevant locations with the normalized form of m. During the iteration over proof tree nodes, a node will be accepted as part of the slice if it accesses a relevant location in the desired way according to the slicing algorithm. If a node is accepted, function `accept` might add locations to, or remove locations from, relevantLocations.

The **while** loop at line 4 iterates over proof tree nodes as long as more parent nodes are available and the set of relevant locations is not empty. Method `isSEStep` in line 5 checks if the current proof tree node performs symbolic execution[7] in the modality of interest[8]. If this is the case the current aliases are computed at line 6.

Method `accept` at line 7 checks if the current proof tree node is part of the slice. An implementation is also responsible to update the relevant locations. If the current node is accepted, the result is updated at line 8.

If the current proof tree node executes an assignment, the affected relevant locations are normalized again at line 11 to ensure that the correct representative is used. Finally, lines 13 and 15 prepare the next loop iteration.

To realize thin slicing, a proof tree node is accepted by method `accept`[9],

[7] All rules modifying the program of a modality without the distinction of statements which are part of the source code or caused by decomposition as done by KeY's Symbolic Execution Engine (Chapter 4).

[8] A modality with an *SE* label, see Section 4.1.

[9] In contrast to the original thin slicing, n is only part of the slice if it writes to m.

- if an assignment writing to a relevant location is executed. The written location is removed and all read locations are added to the relevant locations.
- if a method inlining is performed and the result variable is a relevant location. The result location is removed from the relevant locations.
- if a loop specification or method contract is applied and at least one assignable location is relevant. All assignable locations are removed from the relevant locations.

  In Java DL, an anonymization of heap locations is achieved by the *anon* function, see Section 2.3.2. What is challenging is, that the anonymized locations are not directly listed. Instead, a term-structure is used to specify them. Without knowing the details, the relevant locations can be computed by a side proof of the form:

  $$\Gamma \Longrightarrow \mathsf{c} \rightarrow \{\mathscr{U}\}\mathit{ResultPredicate}(\mathsf{relevantLocations}), \Delta$$

  It computes the value of each location of relevantLocations in the state defined by the updates $\mathscr{U}$ and in case that the memory layout condition c holds. Γ and Δ are all first-order top-level formulas of the original sequent, representing the context knowledge. A relevant location is part of the assignable clause if it has an anonymous value in at least one goal of the side proof.

To realize forward slicing, Algorithm 5.4 can be adjusted. Instead of traversing the parent nodes, the children on the path down to a given leaf are visited. The traversing can stop additionally, in case that after a split c does not hold anymore. Additionally, read and written locations need to be swapped in the implementation of method `accept`.

5.2.2 Slicing a Symbolic Execution Tree

The slicing criterion of a proof tree is defined analogous to Section 5.2.1 as follows:

Definition 5.6 (Symbolic Execution Tree Slicing Criterion)**.** *The* slicing criterion *of a symbolic execution tree is a triple* (m, n, c) *with the memory location* m *of interest at symbolic execution tree node* n *and an optional condition* c *specifying a memory layout.*

The memory layout condition c *lists all aliased memory locations and is of the form* $\bigwedge_{i,j} l_i \doteq l_j$ *where* l_i, l_j *are memory locations.*

An example of slicing criteria and the resulting backward slices is given in Figure 5.3. The symbolic execution tree, as visualized by the SED (Chapter 6), is created by executing method `magic` under pre-condition `a != null & b != null`. The slicing criteria are interested in location `a.value` at symbolic execution tree node (*return a.value + b.value*). The additional condition is used to separate the cases that method parameters `a` and `b` are aliased. Please observe that the memory layout condition `a` $\doteq$ `b` is only given for readability. The resulting backward slices are colored in red and green.

Algorithm 5.4 is also applicable to symbolic execution trees, with the simplification that the case distinction at line 5 is not needed anymore. However, the decision if a symbolic execution tree node is part of the slice depends on the read and written locations. To offer this information as part of the symbolic execution engine presented in Chapter 4 is not trivial. First, the symbolic execution tree is optimized for a human inspection. This includes that intermediate states caused by decomposition of complex statements are filtered out, but they need to be considered for slicing. Second, executing a statement might split into multiple branches on which different locations are accessed (by the filtered out nodes).

Slicing performed on the proof tree on the other hand does not have these complicating factors as all states are available and each performs only a single atomic step. For this reason, the proof from which

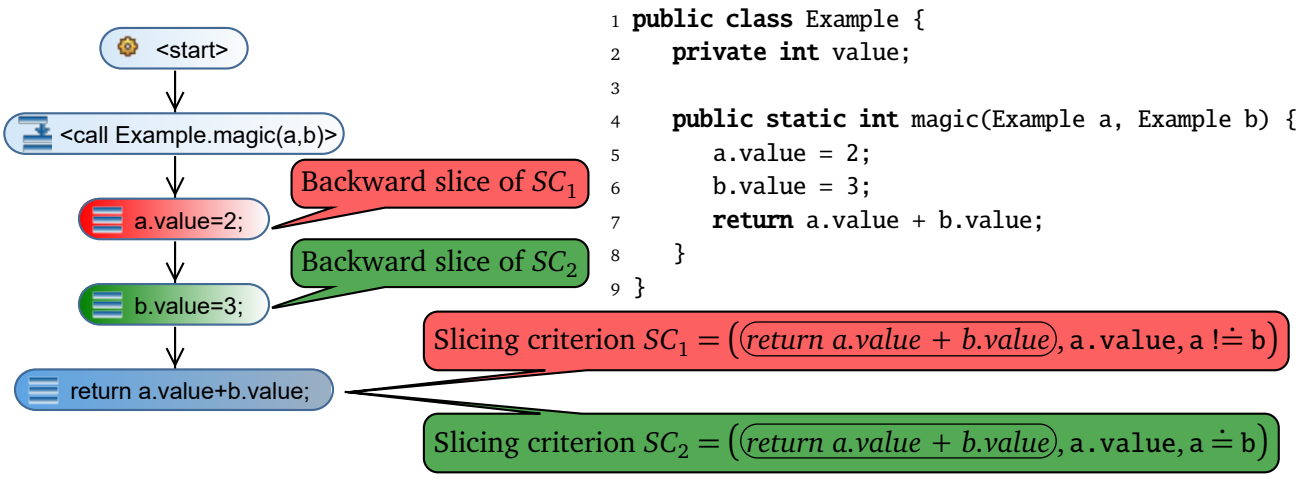


Figure 5.3.: Symbolic Execution Tree Slicing Example

the symbolic execution tree is generated is sliced according to Section 5.2.1. Proof tree nodes part of the slice are then mapped to a symbolic execution tree node.

The mapping from a symbolic execution tree node to a proof tree node and vice versa is realized as follows. Each symbolic execution tree node has a pointer to the proof tree node it represents. A proof tree node without a representation in the symbolic execution tree is mapped to the symbolic execution tree node of its closest ancestor with a representation.

6 Symbolic Execution Debugger (SED)

Out of the four papers ([28, 25, 84, 85]) that introduced independently symbolic execution as a program analysis technique during the mid 1970s, no less than three mentioned debugging as a motivation. Indeed, symbolic execution has a number of natural properties which make it attractive in helping to debug programs:

- A time-consuming task for users of classical interactive debuggers is to set up a (small) initial program state which leads to an execution that exhibits the failure. It is usually non-trivial to build the required, complex data structures. Symbolic program execution, on the other hand, permits to execute any method or any statement directly without setting up an initial state. This is possible by using symbolic values instead of concrete ones. The capability to start debugging from any code location makes it also easy to debug incomplete programs.

- Not only is it time-consuming to build concrete initial states, it is often also difficult to determine under which exact conditions a failure will occur. This can be addressed by symbolic execution, which allows one to specify initial states only partially (or not at all) and which generates *all* reachable symbolic states up to a chosen depth.

- Classical debuggers typically pass through a vast number of program states with possibly large data structures before interactive debugging mode is entered. Once this happens, it is often necessary to visit previous states, which requires to implement reverse (or omniscient) debugging, which is non-trivial to do efficiently as discussed by Pothier et al. [108]. In a symbolic execution environment reverse debugging causes only little overhead, because (i) symbolic execution can be started immediately in the code area where the defect is suspected and (ii) symbolic states are small and contain only program variables encountered during symbolic execution.

- The code instrumentation typically required by standard debuggers can make it impossible to observe a failure that shows up in the unaltered program, so-called *Heisenbugs* [61]. This can be avoided by symbolic execution of the unchanged code.

Given these advantages of symbolic execution, plus the fact that the idea to combine it with debugging has been around for 40 years, the question is then why all widely used debuggers are still based on interpretation of programs with concrete start states? Stable mainstream debugging tools evolved slowly and their feature set remained more or less stable in the last decades, providing mainly the standard functionality for stepwise execution, inspection of the current program state, and suspension of the execution before a marked statement is executed. This is all the more puzzling, since debugging is a central, unavoidable, and time-consuming task in software development with an accordingly huge saving potential.

The probable answer is that, until relatively recently, standard hardware simply was insufficient to realize a debugger based on symbolic execution for real-world programming languages. On a closer look, there are three aspects to this. First, symbolic execution itself: reasonably efficient symbolic execution engines for interesting fragments of real-world programming languages are available only since ca. 2006 (e.g. KeY [22] for Java, XRT [62] for .NET or VeriFast [74] for C). Second, and this is less obvious, to make good use of the advantages of symbolic execution pointed out above, it is essential to *visualize* symbolic execution paths and symbolic states and navigate through them. Otherwise, the sheer amount and the symbolic character of the generated information make it impossible to understand what is happening. The third obstacle to adoption of symbolic execution as a debugging technology is lack of integration. Developers expect that a debugger is smoothly integrated into the development environment of their

choice, so that debugging, editing, testing, and documenting activities can be part of a single workflow without breaking the tool chain.

These issues were for the first time addressed in a prototypic symbolic state debugger by Hähnle et al. [63]. However, that tool was not very stable and its architecture was tightly integrated into the KeY system. As a consequence, the *Symbolic Execution Debugger* (SED) [66] presented in this chapter was completely rewritten, much extended and realized as a reusable Eclipse extension.

The SED extends the Eclipse debug platform by adding an API for symbolic execution and by visualization capabilities for symbolic execution. Although different symbolic execution engines can be integrated into the SED platform, in the following only the integration of KeY as symbolic execution engine (Chapter 4) is presented. An experimental integration of JPF–SE [10] also exists. The main goal of the tool is to help program understanding. Like a traditional debugger it allows the user to control the execution, to inspect states and to suspend execution at defined breakpoints.

In contrast to the KeY system as used for verification, the SED can be used without any specialist knowledge, exactly like a standard debugger. To make full usage of its capabilities, however, it is of advantage to know the basic concepts of symbolic execution. An introduction into the SED's notion of a symbolic execution tree is given in Section 6.1. The basic usage of the SED is explained in tutorial style in Section 6.2. How to apply the SED profitably in various use cases, including tracking the origin of failures (Section 6.3 and Section 6.4), help in program understanding (Section 6.5), and even actual program verification (Section 6.6) is presented next. How the truth status tracing (Section 5.1) and slicing (Section 5.2) is integrated into the SED is shown by Section 6.7 and Section 6.8. Section 6.9 explains the architecture of the SED, which has a highly modular design and allows one to integrate other symbolic execution engines than KeY into SED. Finally, Section 6.10 presents the symbolic debug model which needs to be implemented in order to integrate a symbolic execution engine and Section 6.11 discusses the annotation model which can be used to present results of an analysis based on symbolic execution by highlighting nodes.

6.1 Visualization of Symbolic Execution Trees

This section explains the notion of a symbolic execution tree used by the SED by way of examples. Listing 6.1 shows Java method `min`, which computes the minimum of two given integers.

```
1 public static int min(int x, int y) {
2    if (x < y) {
3       return x;
4    }
5    else {
6       return y;
7    }
8 }
```

Listing 6.1: Minimum of Two Integers

The complete symbolic execution tree of method `min` is shown in Figure 6.1. The root of each symbolic execution tree in the SED's notion of symbolic execution is a *start node*, usually followed by a call of the method to execute.

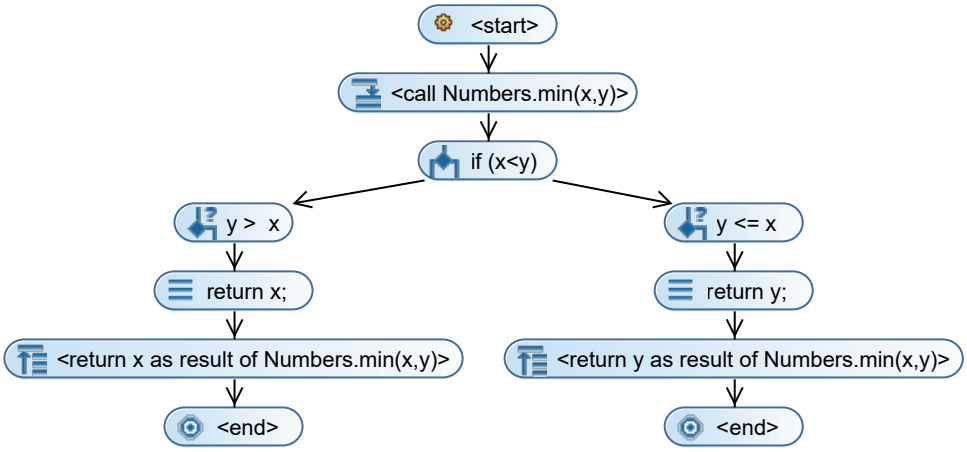


Figure 6.1.: Symbolic Execution Tree of Static Method `min` Defined in Class `Numbers`

Typically, an **`if`**-statement splits execution. For this reason it is represented as a *branch statement*. Its child nodes are *branch conditions* representing the condition when a branch is taken. Branch conditions occur after branch statements if and only if execution splits. If a branch statement does not split because only one execution path is feasible, then its child is the next statement to execute. But also other statements than explicit branch statements can split execution, for instance, an object access that may throw a `NullPointerException`. Whenever a statement splits execution, its children show the relevant branch conditions and continue execution.

In the example, on each branch a return statement is executed which causes a method return and lets the program terminate normally (without an uncaught exception).

Loop statements are unwound by default, similar as in a concrete program execution. The first time when a loop is entered it is represented as a *loop statement* in the symbolic execution tree. Whenever the loop guard is executed, it will be represented as *loop condition* node and may split execution into two branches. One where the guard is false and execution is continued after the loop and one where it is true and the loop body is executed once and the loop guard is checked again. As a consequence, unwinding a loop can result in symbolic execution trees of unbounded depth. As an illustration the method in Listing 6.2 is used which computes the sum of array elements.

```
1 public static int sum(int[] array) {
2    int sum = 0;
3    for (int i = 0; i < array.length; i++) {
4       sum += array[i];
5    }
6    return sum;
7 }
```

Listing 6.2: Sum of All Array Elements

The beginning of a symbolic execution tree resulting from execution of `sum` with precondition `array != null` is shown in Figure 6.2. The left branch stops before the loop guard is evaluated the second time, whereas the right branch terminates after the computed sum is returned. When symbolic execution is continued on the left branch, similar child branches will be created until `Integer.MAX_VALUE` is reached.

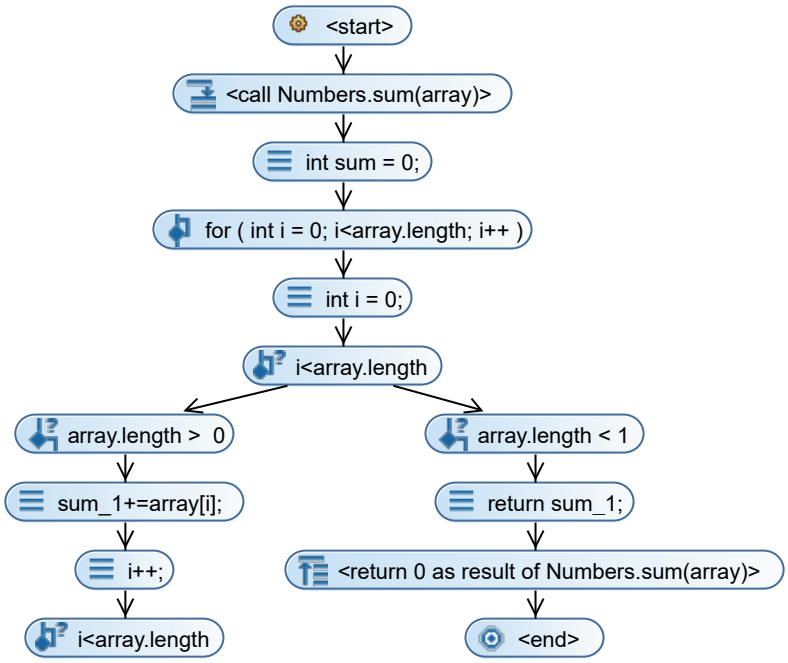


Figure 6.2.: Symbolic Execution Tree of Static Method `sum` Defined in Class `Numbers`

To render symbolic execution trees finite in presence of loops, optionally, a loop specification can be supplied (see Section 3.2). In this case a *loop invariant* node is shown in the symbolic execution tree splitting execution into two branches. The first *body preserves invariant* branch represents all possible loop iterations ending in *loop body termination* nodes.[1] The second *use case* branch continues execution after the loop. It is possible that the loop invariant was initially not valid or that it is not preserved by the loop body. This would be a problem in a verification scenario, but a violated loop invariant should not stop one from debugging a program. Therefore, different icons indicate whether the loop invariant holds initially and in a *loop body termination* node.

The sum example in Listing 6.2 is extended by a weak (and wrong) loop invariant in Listing 6.3. A correct loop invariant would treat the case that `i` can be zero. For verification it would also be required to specify how the value of `sum` is changed by the loop.

```
1 /*@ loop_invariant i > 0 && i <= array.length;
2   @ decreasing array.length - i;
3   @ assignable \strictly_nothing;
4   @*/
5 for (int i = 0; i < array.length; i++) { /* ... */ }
```

Listing 6.3: Wrong and Weak Loop Invariant of Loop from Listing 6.2

The resulting symbolic execution tree using the loop specification and precondition `array != null` is shown in Figure 6.3. The icon of the loop invariant node indicates by the red cross that it is initially not fulfilled.

[1] In case an exception is thrown or a jump outside of the loop is initiated by a **`return`**, **`break`** or **`continue`** statement, execution is continued directly in the *body preserves invariant* branch.

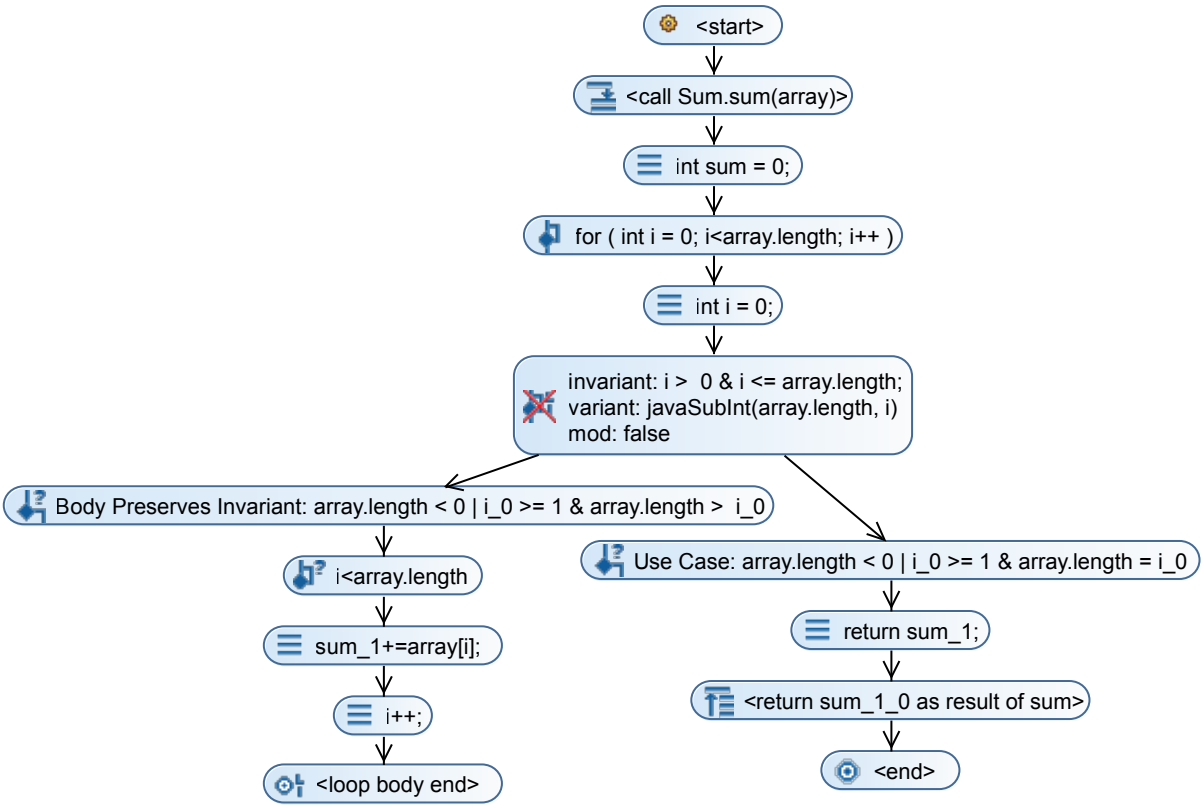


Figure 6.3.: Symbolic Execution Tree of Static Method `sum` Using a Loop Specification

Method calls are handled by default by inlining the body of the called method. In case of inheritance, symbolic execution splits to cover all possible implementations indicated by *branch condition* nodes in front of the *method call* node.

The usage of inlined methods is explained with help of the example in Listing 6.4 which calls in method `run` of class `Main` the `run` method of an `IOperation`. Two different `IOperation` implementations are available (`FooOperation` and `BarOperation`).

The resulting symbolic execution tree under precondition `operation != null` is shown in Figure 6.4. First, the target method is inlined and its body is executed between the *method call* and the corresponding *method return* node. The only statement calls method `run` on the argument `operation`. As the actual implementation which is executed after the dynamic dispatch is unknown, symbolic execution has to split to consider the one implemented by `BarOperation` and the one implemented by `FooOperation`. The implementation taken in a branch is shown by a *branch condition* node. Here, the left branch continues execution in case that `operation` is an instance of `BarOperation` and the right one in the other case. Then, both branches inline the target method, execute the return statement, return from the called method, and finally terminate normally.

```
1  public class Main {
2     public static String run(IOperation operation) {
3        return operation.run();
4     }
5  }
6
7  interface IOperation {
8     public String run();
9  }
10
11 class FooOperation implements IOperation {
12    public String run() {
13       return "foo";
14    }
15 }
16
17 class BarOperation implements IOperation {
18    public String run() {
19       return "bar";
20    }
21 }
```

Listing 6.4: Method Call with Inheritance

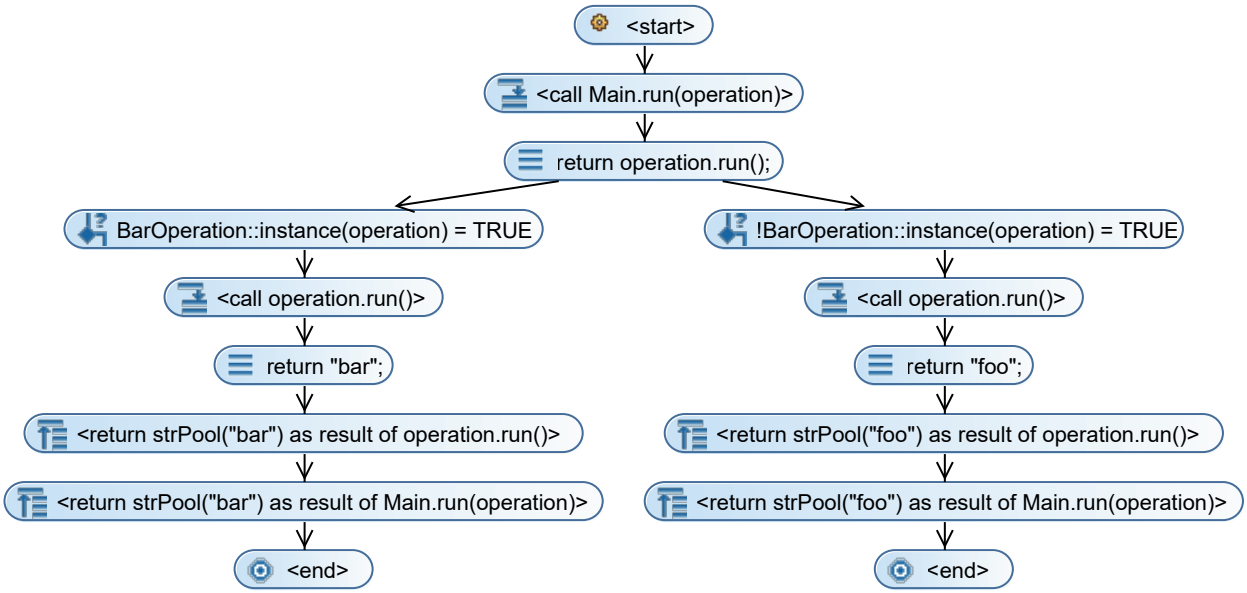


Figure 6.4.: Symbolic Execution Tree of Static Method `run`

As in the case of loops, recursive method calls can lead to unbounded symbolic execution trees. But even unfolding non-recursive calls can quickly lead to infeasible large symbolic execution trees. To address this issue, instead of inlining the method body, it is possible to replace a method call by a *method contract* (see Section 3.3). This can also be useful when the source code of a method implementation is not available. For example, if it is proprietary code or simply unfinished.

Upon application of a method contract, symbolic execution is continued separately for the specification cases corresponding to normal and to exceptional behavior. As in the case of loop invariants, node icons are used to indicate if certain conditions like preconditions or that the callee is not **null** could not be established.

Listing 6.5 shows the contract of method sum from Listing 6.2. The sum method is used to compute the average of all array elements in Listing 6.6.

```
1  /*@ normal_behavior
2    @ requires array != null;
3    @ ensures \result == (\sum int i; i >= 0 && i < array.length; array[i]);
4    @
5    @ also
6    @
7    @ exceptional_behavior
8    @ requires array == null;
9    @ signals_only NullPointerException;
10   @ signals (NullPointerException) true;
11   @*/
12 public static /*@ pure @*/ int sum(/*@ nullable @*/ int[] array) {
13     // ...
14 }
```

Listing 6.5: Method Contract of Method sum from Listing 6.2

```
1 public static int average(/*@ nullable @*/ int[] array) {
2    return sum(array) / array.length;
3 }
```

Listing 6.6: Average of All Array Elements

The symbolic execution tree resulting from the execution of method average, where the contract of sum is used to handle the call to sum, is shown in Figure 6.5. The left branch terminates with an uncaught ArithmeticException in case that the array is empty whereas the middle branch terminates normally after the computed average is returned. The right branch terminates with an uncaught Throwable in case the array is **null**.

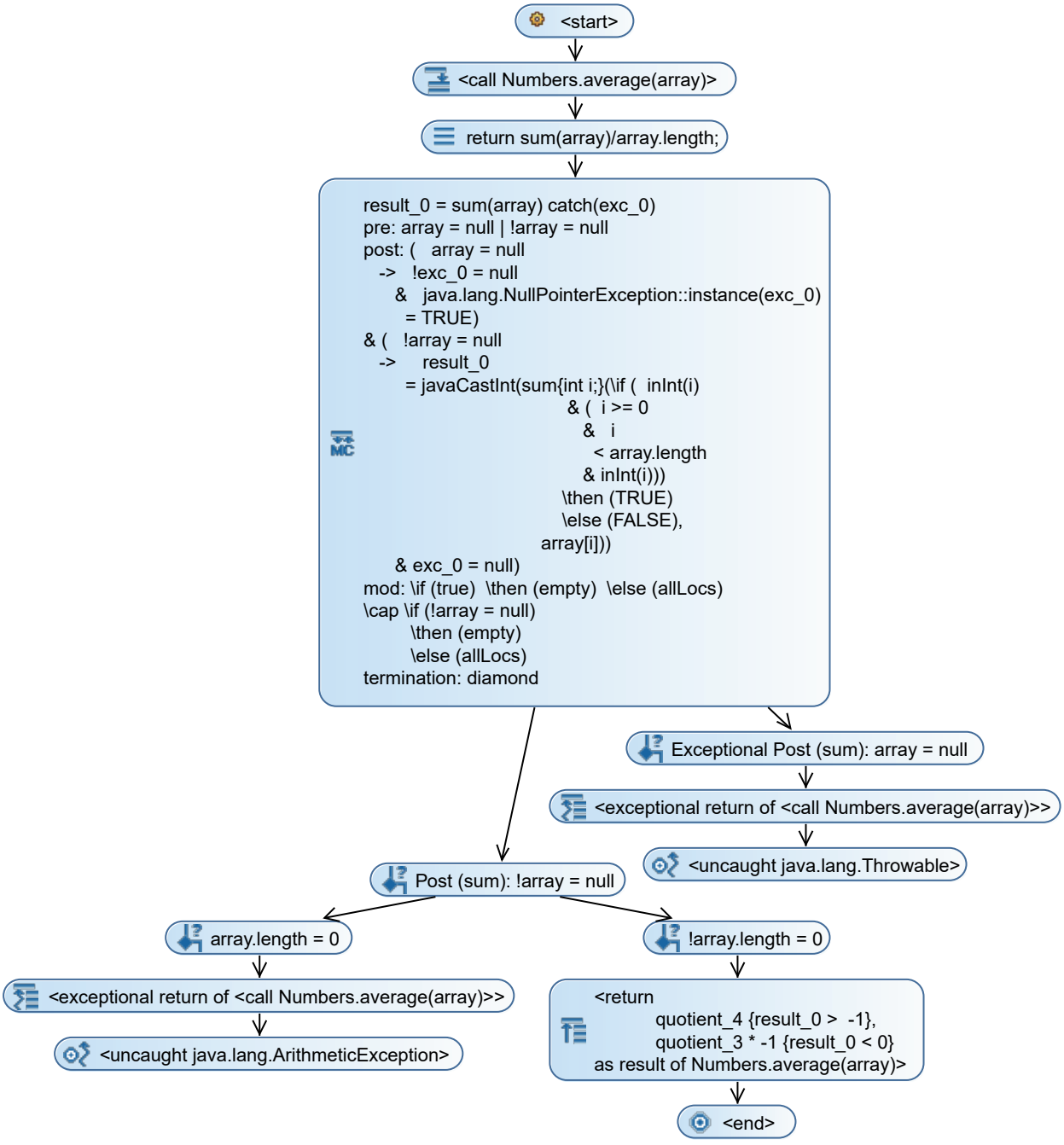


Figure 6.5.: Symbolic Execution Tree of Method `average` Using a Contract for the Called Method

Table 6.1 summarizes the different nodes which are used in the SED's notion of a symbolic execution tree. Please observe that icons of start and statement nodes are compatible with the Eclipse usage.

Icon & Node Type	Description
Start	The root of a symbolic execution tree.
Branch Statement	The program state before a branch statement (**`if`** and **`switch`** in Java) is executed.
Loop Statement	The program state before a loop (**`while`**, **`do`**, **`for`** and for-each loop in Java) is executed. It occurs only once when the loop is entered the first time.
Loop Condition	The program state before a loop condition is executed. It is repeated in every loop iteration.
Statement	Any statement which is not a branch statement, loop statement and loop condition.
Branch Condition	The condition under which a branch is taken.
Termination	The last node of a branch indicating that the program terminates normally. If the postcondition does not hold, icon is used instead.
Exceptional Termination	The last node of a branch indicating that the program terminates with an uncaught exception. If the postcondition does not hold, icon is used instead.
Method Call	The event that a method body is inlined and will be executed next.
Method Return	The event that a method body is completely executed. Execution will be continued in the caller of the returned method.
Exceptional Method Return	The event that a method returns by throwing an exception. Execution will be continued where the exception is caught. Otherwise, execution finishes with an exceptional termination node.
Method Contract	A method contract is applied to treat a method call. If the object on which the method is called can be **`null`**, icon is used instead. If the precondtion does not hold, icon shows this circumstance. If both do not hold, icon is used.
Loop Invariant	A loop specification is applied to treat a loop. If the loop invariant is initially not fulfilled, the icon is used instead.
Loop Body Termination	The branch of a loop invariant node which executes only loop guard and loop body once is completed. If the loop invariant does not hold, the icon is used instead.

Table 6.1.: Symbolic Execution Tree Nodes

6.2 Basic Usage

The main use case of the SED using KeY is to execute a Java method symbolically. It can be achieved by opening the context menu of a method and by selecting **Debug As, Symbolic Execution Debugger (SED)** in Eclipse. Alternatively, it is possible to execute individual Java statements by selecting them first in the Java text editor and then by selecting the same context menu entry. Additional knowledge to limit feasible execution paths can be supplied as a precondition in the *Debug Configuration*. Also a full method contract can be selected instead of specifying a precondition.[2] In this case icons of termination nodes

[2] The use of a method contract activates full JML support including **`non_null`** defaults.

will indicate whenever the postcondition is not fulfilled. After starting execution, it is recommended to switch to the perspective **Symbolic Debug** which contains all relevant views explained in Table 6.2.

View	Description
Debug	Shows symbolic execution trees of all launches, allows one to switch between them and to control execution.
Symbolic Execution Tree	Visualizes symbolic execution trees of selected launches.
Symbolic Execution Tree (Thumbnail)	Miniature view of the **Symbolic Execution Tree** view for navigation purposes.
Variables	Shows the visible variables and their symbolic values.
Breakpoints	Manages the breakpoints.
Properties	Shows all information of the currently selected element (in particular a symbolic execution tree node or a variable).
Symbolic Execution Settings	Allows one to customize symbolic execution, e.g., defines how to treat method calls and loops.

Table 6.2.: Views of Perspective **Symbolic Debug**

Figure 6.6 shows a screenshot of the **Symbolic Debug** perspective in which the symbolic execution tree of method `equals`, whose implementation is shown in the bottom right editor, is visualized. The method checks whether its `Number` argument instance has the same content as **`this`**, which is named `self` in KeY. The left branch represents the case when both instances have the same content, whereas the content is different in the middle branch. The right branch terminates with an uncaught `NullPointerException`, because the argument is **`null`**.

The additional frames[3] (blue rectangles) displayed in view **Symbolic Execution Tree** of Figure 6.6 represent the bounds of code blocks. Such frames can be independently collapsed and expanded to abstract away from the inner structure of code blocks, thus achieving a cleaner representation of the overall code structure by providing only as much detail as required for the task at hand. A collapsed frame contains only one branch condition node per path (namely the conjunction of all branch conditions of that particular path), displaying the constraint under which the end of the corresponding code block is reached. In Figure 6.7, the method call node is collapsed. The green color of the frame indicates that all execution paths reach the end of the frame. Otherwise, the frame would be colored in orange.

The symbolic program state of a selected node is shown in the view **Variables**. The details of a selected variable (e.g. additional constraints) or symbolic execution tree node (e.g. path condition, call stack, etc.) are available in the **Properties** view. The source code line corresponding to the selected symbolic execution tree node is highlighted in the editor. Additionally, the editor highlights statements and code members reached during symbolic execution.

The **Symbolic Execution Settings** view lets one customize symbolic execution, e.g., one can choose between method inlining and method contract application. Breakpoints suspend the execution and are managed in the **Breakpoints** view.

In Figure 6.6 the symbolic execution tree node (*return true;*) is selected, which is indicated by a darker color. The symbolic value of field `content` of the current instance `self` and of the argument instance `n` are identical. This is not surprising, because this is exactly what is enforced by the path condition. A fallacy and source of defects is to implicitly assume that `self` and `n` refer to different instances as they are named differently and here also because that an object is passed to itself as a method argument. This is because the path condition is also satisfied if `n` and `self` reference the same object. The SED helps to detect and locate unintended aliasing by determining and visualizing all possible memory layouts w.r.t. the current path condition.

[3] The visualization of code blocks using frames was implemented as part of the Bachelor's thesis by Möller [102].

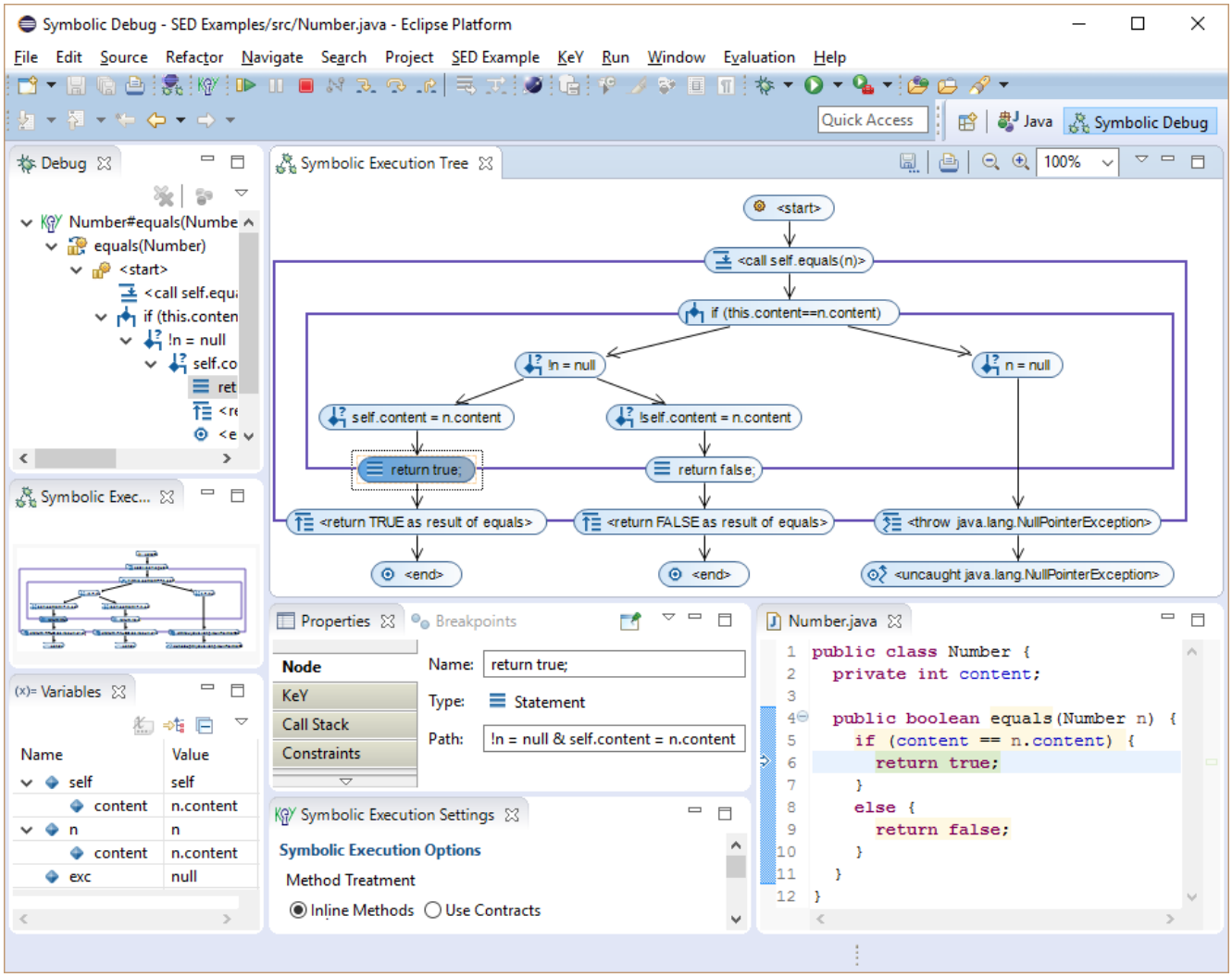


Figure 6.6.: Symbolic Execution Debugger: Interactive Symbolic Execution

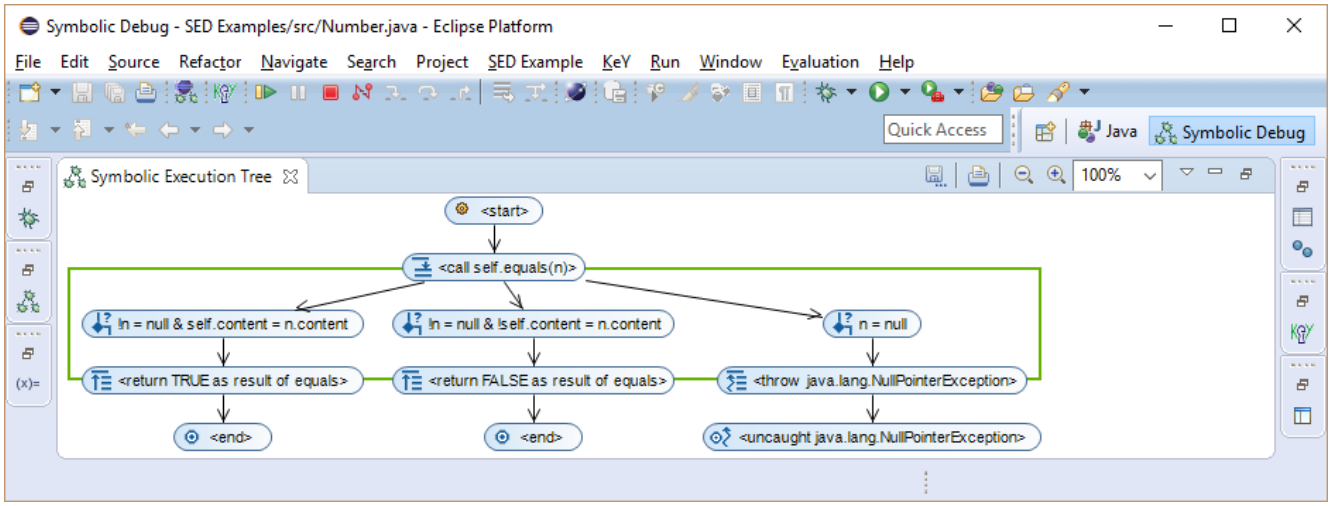


Figure 6.7.: Symbolic Execution Debugger: Collapsed Frame

Selecting context menu item **Visualize Memory Layouts** of a symbolic execution tree node creates a visualization of possible memory layouts as a *symbolic object diagram*, see Figure 6.8. It resembles a UML object diagram and shows (i) the dependencies between objects, (ii) the symbolic values of object fields and (iii) the symbolic values of local variables of the current state.

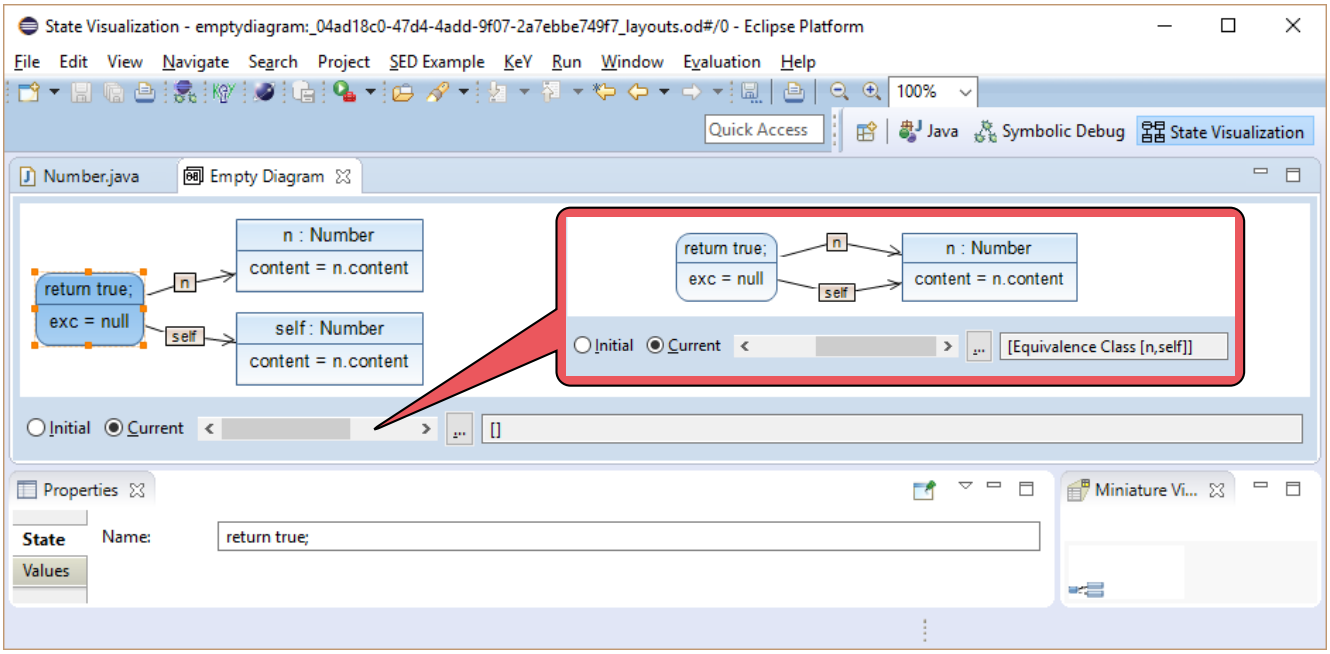


Figure 6.8.: Symbolic Execution Debugger: Possible Memory Layouts of a Symbolic State

The root of the symbolic object diagram is visualized as a rounded rectangle and shows all local variables visible at the current node. In Figure 6.8, the local variables `n` and `self` refer to objects visualized as rectangles. The symbolic value of the instance field `content` is shown in the lower compartment of each object. The local variable `exc` is used by KeY to distinguish among normal and exceptional termination.

The toolbar (near the origin of the callout) allows one to select different possible layouts and to switch between the current and the initial state of each layout. The initial state shows how the memory layout looked before the execution started resulting in the current state. Figure 6.8 shows both possible layouts of the selected node (*return true;*) in the current state. The second memory layout (inside the callout) represents the situation, where `n` and `self` are aliased.

6.3 Debugging with Symbolic Execution Trees

The Symbolic Execution Debugger allows one to control execution like a traditional debugger and can be used in a similar way. A major advantage of symbolic execution is that it is not required to start at a predefined program entry point and to run the program until the point of interest is reached. Instead, the debug session can start directly at the point of interest. This avoids to build up large data structures and the memory will contain only the variables used by the code of interest. If knowledge about the conditions under which a failure can be observed is available, this can be given as a precondition to limit the number of explored execution paths.

The main task of the user is, like in a traditional debugger, to control execution and to comprehend each performed step. It is helpful to focus on a single branch where it is expected to reach a faulty state. If this is not the case, the focus can be changed to a different branch. There is no need for a new debugging session or to find new input values resulting in a different execution path. It is always possible to go back to previous steps, because each node in the symbolic execution tree provides the full symbolic state.

Of special interest are splits, because their explicit rendering in the symbolic execution tree constitutes a major advantage of the SED over traditional debuggers. Unexpected splits or missing expected splits are good candidates for possible sources of defects. This is explained by example. Listing 6.7 shows a defective part of a *Mergesort* implementation for sorting an array called `intArr`. The exception shown in Listing 6.8 was thrown during a concrete execution of a large application that contained a call to `sort`. It seems that method `sortRange` calls itself infinitely often in line 9 until the call stack is full, which happened in line 7.

```
1  public class Mergesort {
2     public static void sort(int[] intArr) {
3        sortRange(intArr, 0, intArr.length - 1);
4     }
5
6     public static void sortRange(int[] intArr, int l, int r) {
7        if (l <= r) {
8           int q = (l + r) / 2;
9           sortRange(intArr, l, q);
10          sortRange(intArr, q + 1, r);
11          merge(intArr, l, q, r);
12       }
13    }
14
15    private static void merge(int[] intArr, int l, int q, int r) {
16       int[] arr = new int[intArr.length];
17       int i, j;
18       for (i = l; i <= q; i++) {
19          arr[i] = intArr[i];
20       }
21       for (j = q + 1; j <= r; j++) {
22          arr[r + q + 1 - j] = intArr[j];
23       }
24       i = l;
25       j = r;
26       for (int k = l; k <= r; k++) {
27          if (arr[i] <= arr[j]) {
28             intArr[k] = arr[i];
29             i++;
30          }
31          else {
32             intArr[k] = arr[j];
33             j--;
34          }
35       }
36    }
37 }
```

Listing 6.7: Defective Part of a Mergesort Implementation[4]

The value of `r` is the termination criterion. Using a traditional debugger the user has to execute the whole program with suitable input values until method `sort` is executed. From this point onwards, she may control the execution, observe how the `r` value is computed and try to find the origin of the failure. With the Symbolic Execution Debugger, however, she can start execution directly at

[4] Modified version of the Mergesort implementation by Jörg Czeschla, see javabeginners.de/Algorithmen/Sortieralgorithmen/Mergesort.php

```
Exception in thread "main" java.lang.StackOverflowError
   at Mergesort.sortRange(Mergesort.java:7)
   at Mergesort.sortRange(Mergesort.java:9)
   at Mergesort.sortRange(Mergesort.java:9)
   at Mergesort.sortRange(Mergesort.java:9)
   ...
```

Listing 6.8: Exception Thrown by the Mergesort Implementation in Listing 6.7

method `sort`. Clearly, the array `intArr` needs to be not **`null`**. This knowledge can be expressed as precondition `intArr != null`. The resulting symbolic execution tree in Figure 6.9 shows already after a few steps that the **`if`**-statement is not branching in case that `intArr` is not empty and the defect is found (the comparison should have been `l < r`).

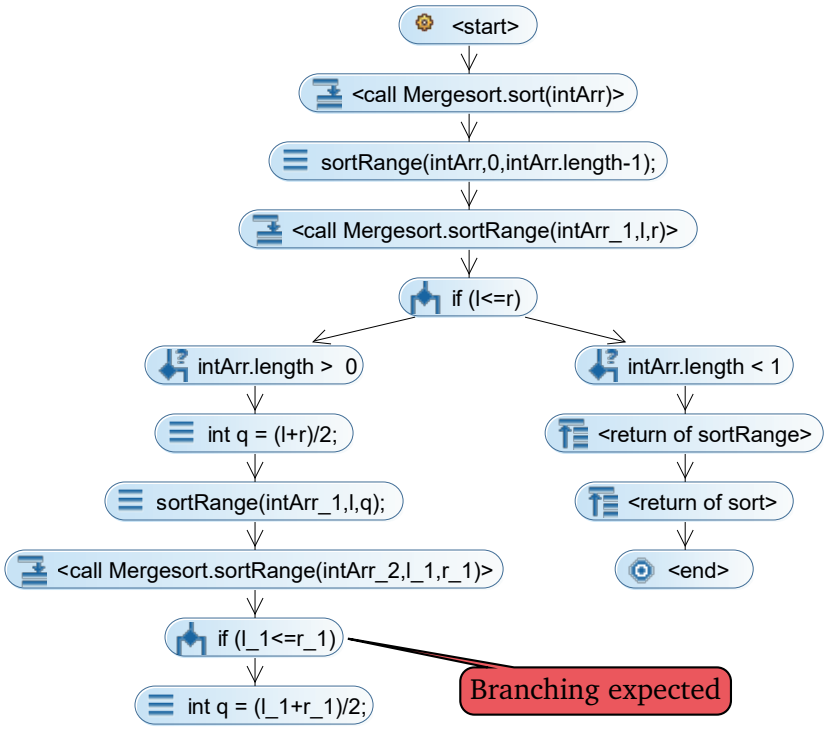


Figure 6.9.: Symbolic Execution Tree of the Mergesort Implementation in Listing 6.7

6.4 Debugging with Memory Layouts

It is easy to make careless mistakes in operations which modify data structures. To find them with a traditional debugger can be time consuming, because large data structures have to be inspected after each execution step. A complication is that a program state contains not only the data structure of interest, but all information computed before the state of interest is reached. Traditional debuggers present the current state typically as variable-value pairs in a list or tree. This representation makes it very hard to figure out data structures.

With the Symbolic Execution Debugger it is possible to visualize the current state as well as the initial state from which the execution started in the form of a symbolic object diagram. As an example, consider the rotate left operation of an AVL tree. Each node in such a tree has a left and a right child and it knows its parent as well. Again, symbolic execution is started directly in the method of interest, here the `rotateLeft` method and the SED is asked to compute all memory layouts for one of its return nodes.

The initial state that looks like the one in Figure 6.10 is of interest. The node to rotate is named `current` and it is the root of the tree because its parent is **`null`**. It has a right child, which in turn has a left child. The AVL tree itself is named `self`. Additionally, precondition `current != null && current.right != null` is used to ensure that the nodes to rotate exist.

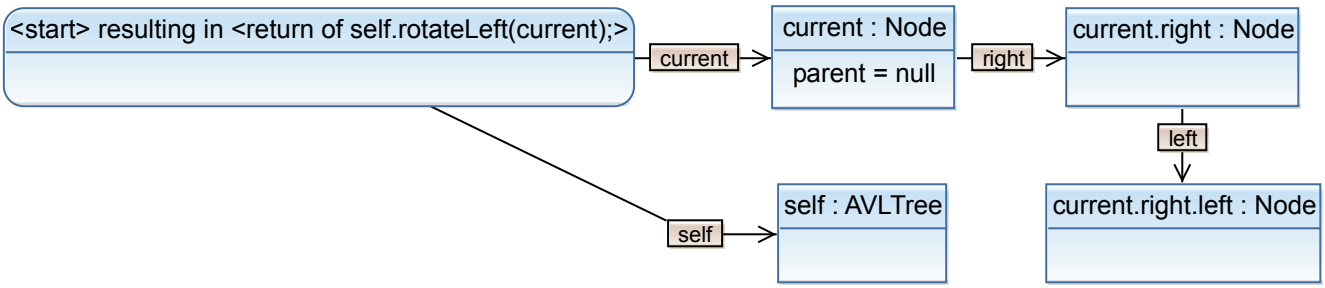


Figure 6.10.: Initial Symbolic Object Diagram of an AVL Tree Rotate Left Operation

The symbolic state automatically computed and visualized by the SED after performing the rotation is shown in Figure 6.11. It shows the initial objects with all performed changes. By inspecting this diagram it is obvious that the parent of object `current.right.left` was not correctly updated because its parent is now the node itself.

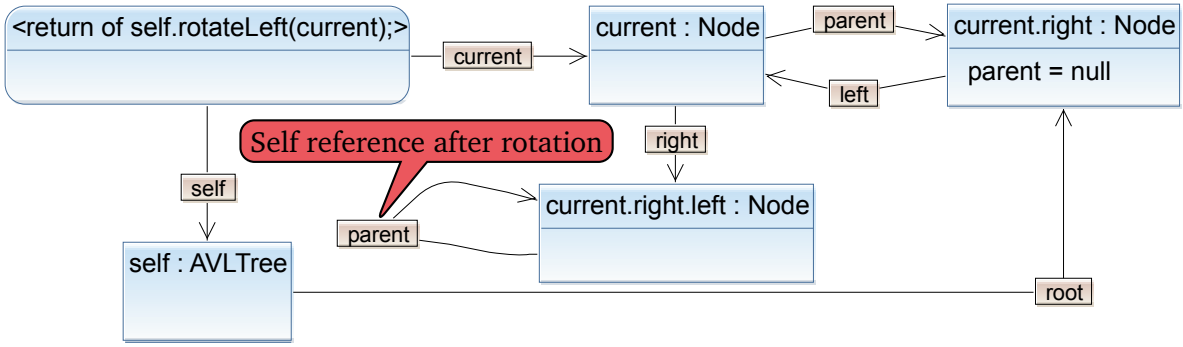


Figure 6.11.: Current Symbolic Object Diagram of an AVL Tree Rotate Left Operation

6.5 Help Program and Specification Understanding

An important feature of symbolic execution trees is that they show control and data flow at the same time. Thus they can be used to help understanding programs and specifications just by inspecting them. This can be useful during code reviews or in early prototyping phases, where the full implementation is not yet available. It works best, when some method contracts and/or loop specifications are available to achieve compact and finite symbolic execution trees. However, useful specifications can be much weaker than those that would be required for verification.

For example, Listing 6.9 shows a defective implementation of method `indexOf` with a basic method contract limiting the expected input values as well as a very simple loop specification.

The corresponding symbolic execution tree is shown in Figure 6.12. It captures the full behavior of `indexOf`. Without checking any details, one can see that the left-most branch terminates in a state where the loop specification does not hold. Closer inspection (e.g. by tracing truth statuses, see Section 6.7) shows that the variable `i` is not increased when the array element is found, hence the **`decreasing`** clause of the loop specification is violated. The two branches below the *use case* branch correspond to the code after the loop has terminated. In one case an element was found, in the other not. The return nodes show in both cases that instead of the `index` computed in the loop, the value of `i` is returned.

```
1  public class ArrayUtil {
2     /*@ normal_behavior
3       @ requires \invariant_for(filter);
4       @*/
5     public static int /*@ strictly_pure @*/ indexOf(Object[] array,
6                                                     Filter filter) {
7        int index = -1;
8        int i = 0;
9        /*@ loop_invariant i >= 0 && i <= array.length;
10         @ decreasing array.length - i;
11         @ assignable \strictly_nothing;
12         @*/
13       while (index < 0 && i < array.length) {
14          if (filter.accept(array[i])) {
15             index = i;
16          }
17          else {
18             i++;
19          }
20       }
21       return i;
22    }
23
24    public static interface Filter {
25       /*@ normal_behavior
26         @ requires true;
27         @ ensures true;
28         @*/
29       public boolean /*@ strictly_pure @*/ accept(/*@ nullable @*/ Object object);
30    }
31 }
```

Listing 6.9: A Defective and Only Partially Specified Implementation

As this example demonstrates, symbolic execution trees can be used to answer questions, for example, about thrown exceptions (none in the example) or returned values. Within the SED, the full symbolic state of each node is available and can be visualized. Thus it is easily possible to see whether and where new objects are created and which fields are changed when (comparison between initial and current layout). Using breakpoints, symbolic execution is continued until a breakpoint is hit on any branch. Thus it can be used to find execution paths (i) throwing a specified exception, (ii) accessing or modifying a specified field, (iii) calling or returning a specified method or (iv) causing a specified state.

6.6 Debugging Meets Verification

The SED allows one to perform symbolic execution interactively, to visualize the resulting symbolic execution tree, and to inspect symbolic states. Together, this results in a powerful debugging tool that in addition can be used to control symbolic execution and to present results of a symbolic execution based analysis.

Going beyond mere symbolic execution, the SED can also verify that a Java program satisfies a given specification written in JML, because it uses KeY as its underlying symbolic execution engine.[5] A program is correct with respect to its specification if and only if each branch in the symbolic execution tree ends

[5] The debug configuration allows one to select a method contract alternatively to a precondition.

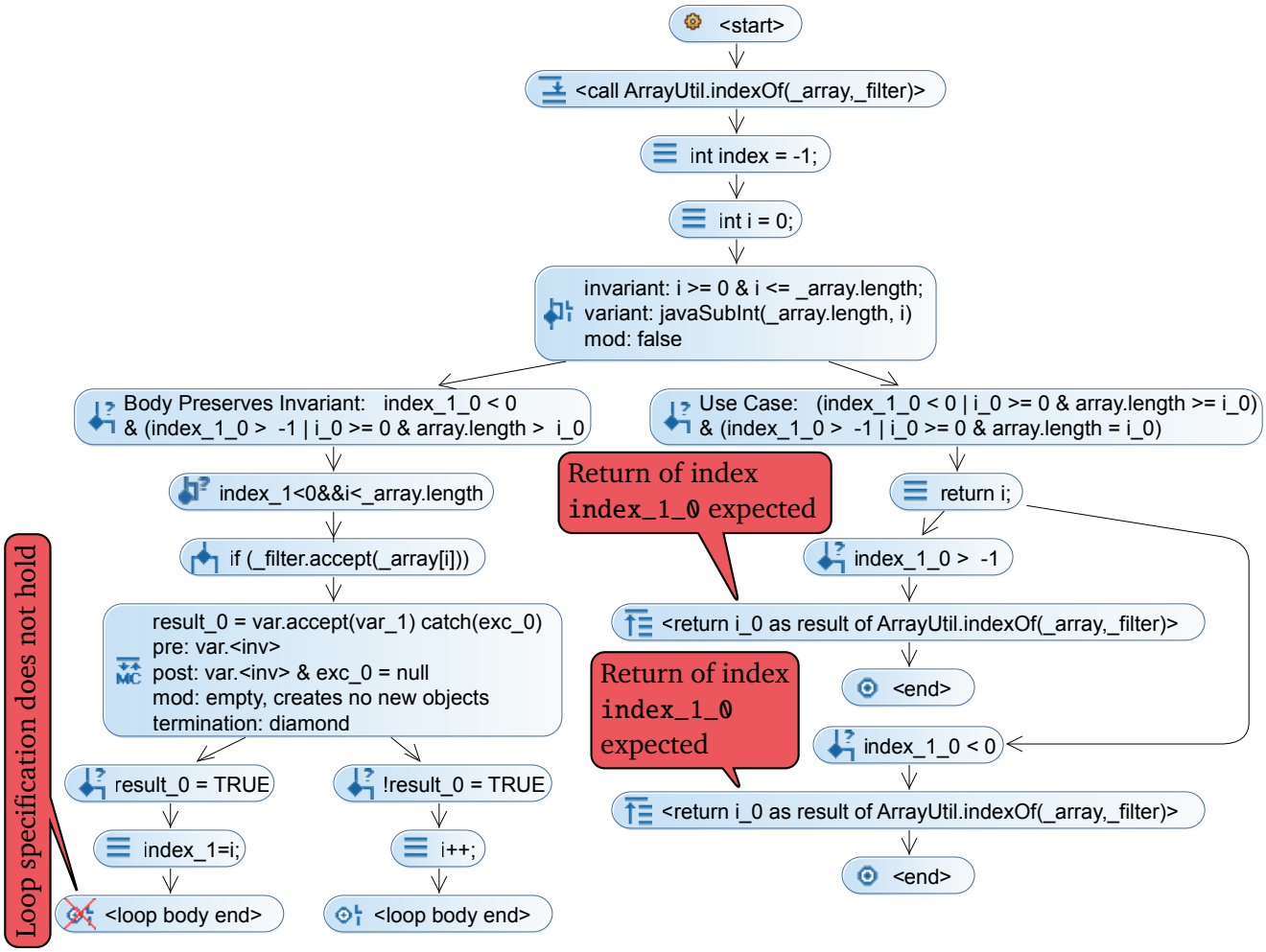


Figure 6.12.: Symbolic Execution Tree of Listing 6.9

with a termination node and no icons are crossed out in red are displayed in the whole tree. In this case all branches terminate in a state where the given postcondition (i.e., JML **ensures** clause) is fulfilled. If a method call is approximated by a method contract, the precondition- and caller-no-null checks must have been successful, too. In addition, all applied loop invariants are valid at the start of their loop and are preserved by the loop body.

A symbolic execution tree produced by the SED displays considerably less information than a full proof tree in KeY: whereas the former contains only nodes that correspond to reachable program states, the latter shows all intermediate symbolic execution steps performed during proof construction, including the proof steps for pure first-order verification conditions. Hence, the SED provides a software developer's view on a KeY proof, hiding intermediate and non symbolic execution related steps. Program states are visualized in a user-friendly way and are not encoded as formulas often distributed and hidden within large proof goals.

A major limitation of the SED compared to the KeY system is that it is currently not possible within the SED to continue a proof interactively in case KeY's proof strategy was not powerful enough to close some goal automatically. But it provides still the means to interact with the prover by adapting or inserting additional JML assertions and thus to use KeY with an auto-active flavor [125, 95].

Another advantage of SED over the KeY GUI is that insufficient or wrong specifications are directly highlighted. Whenever a symbolic execution tree node is crossed out in red, then something went wrong in proving the verification conditions for that path. The user can then inspect the truth statuses evaluated by the proof (see Section 6.7) and the ancestor symbolic execution nodes to check whether

the implementation or the specifications contain a defect. More specifically, if the postcondition in a termination node is not fulfilled, then the symbolic program state at that point should be inspected. Wrong values relative to the specified behavior indicate a defect in the implementation. Values that have been changed as expected, but which are not mentioned in the specification indicate that the specification has to be extended. Moreover, method call and loop invariant nodes crossed out in red indicate that the precondition of the proven specification is too weak or that something went wrong during execution in order to establish the needed conditions. If a loop invariant is not preserved, the state of the loop body at the termination nodes gives hints on how to adjust the loop invariant.

6.7 Inspecting Evaluated Truth Statuses

Whenever the SED is used to verify a method contract (see Section 6.6) or a specification is applied during symbolic execution, then KeY tries to show that the corresponding proof obligation hold. The **Properties** view offers a special tab in which the evaluated truth statuses (see Section 5.1) are shown. This tab is available for all kinds of symbolic execution tree nodes which might be red crossed out. The tab is named after the verified property. The used names are **Postcondition** for termination and exceptional termination nodes, **Loop Invariant** for loop body termination and loop invariant nodes, and **Precondition** for method call nodes.

All these nodes represent the state in the proof tree just before the evaluation of the proof obligation starts. During evaluation, splits might be performed to continue evaluation under additional conditions. This offers also the opportunity to close a branch if the conditions all together are contradictory.

The inferred truth statuses of each branch are shown to the user as a sequent. The antecedent of the sequent is the path condition from the current node down to the leaf node and the succedent is the formula of interest evaluated in the state specified by the updates. A branch is closed if the antecedent evaluates to false (contradiction in path condition) or if the succedent evaluates to true (formula of interest holds). A branch might also be closed if the current symbolic execution tree path is infeasible (contradiction in formulas representing context knowledge). As the presented sequent consists only of the path condition and the postcondition, a contradiction in the context knowledge cannot be seen. However, such a situation can only occur in interactively performed proofs as KeY's proof search strategy closes branches with contradictory formulas as soon as possible.

Different colors are used to visualize the truth status of each formula in the sequent. Green indicates that a formula has been evaluated to true and red that a formula has been evaluated to false. Formulas part of the proof obligation not (yet) evaluated into true or false are colored in orange. Updates and additional formulas not relevant for the evaluation are colored in black. Not relevant formulas can occur in the path condition if a split is based on unlabeled formulas representing the context knowledge, e.g. conditions gained during symbolic execution. KeY's strategy guesses splits to close a branch which can cause that such formulas are part of the path condition.

The source code in Figure 6.13 specifies that method `doNothing` terminates without a thrown exception (**`normal_behavior`**) and in a state fulfilling the postcondition `(p & q) ==> (q & p)` (see Section 5.1.2). View **Properties** shows the evaluated truth statuses of the postcondition at the selected termination node. Each of the two groups evaluates the formula of interest (succedent) under a different condition (antecedent). The formula is successfully verified as each sequent evaluates to true.

The formula of interest shown in the succedent is the direct representation of the JML specification. It is easy to read and contained formulas can be easily mapped back to the related part of the JML specification.

The path condition shown in the antecedent is not that easy to read as the contained formulas occur somewhere in the proof and have not necessarily a counterpart in the JML specifications. In particular, formulas of interest are rewritten, maybe used by a split and thus be part of the branch condition in rewritten form. But the rewritten form can look completely different compared to the original formula or contain additional formulas and terms.

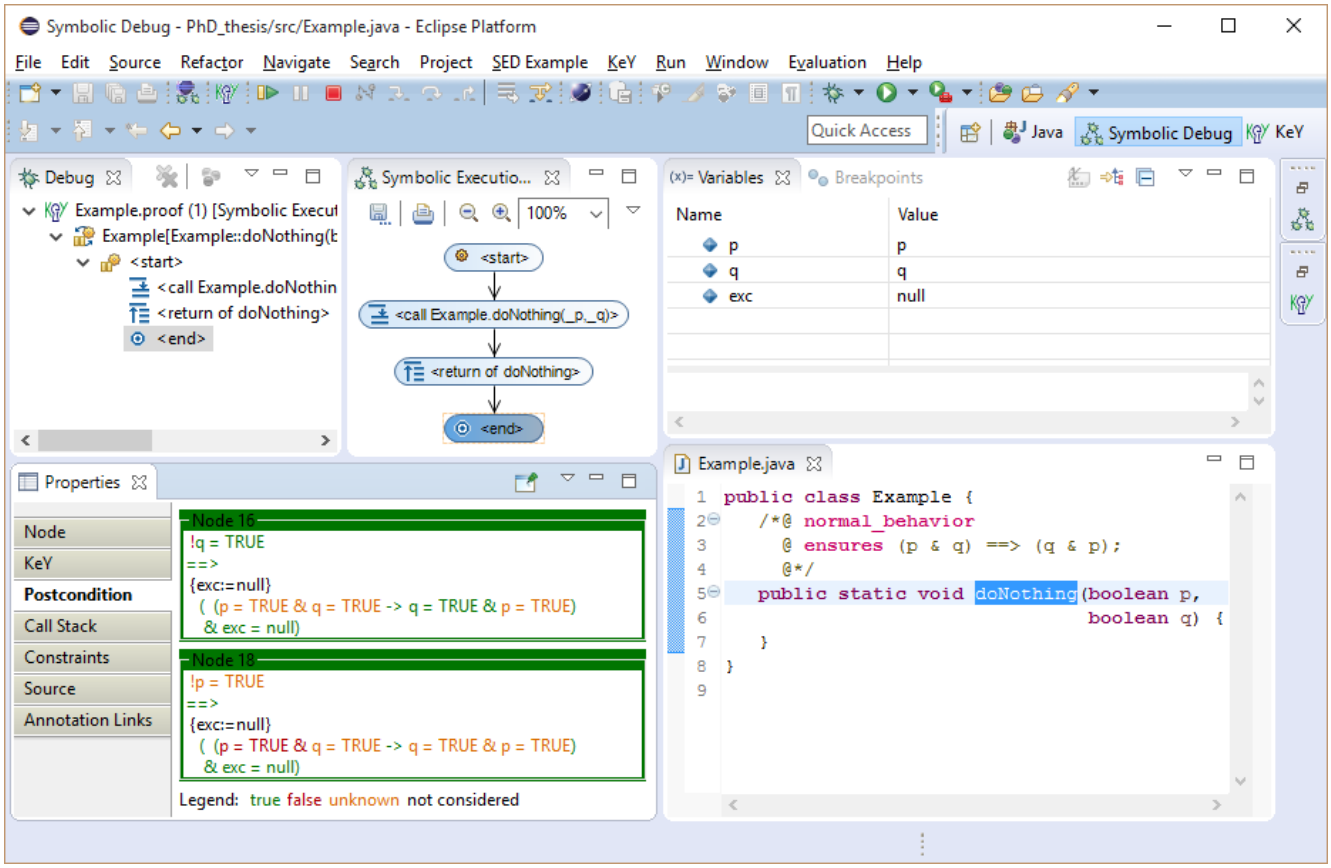


Figure 6.13.: Truth Statuses of the Example from Section 5.1.2

The truth statuses of a defective specification are shown in Figure 6.14. The method add adds `s.value` to **`this`**`.value`. The succedent of Node 190 evaluates to false, shown by the red colored AND operators (&), indicating that something is wrong. KeY was able to verify that the invariant is preserved and that no exception is thrown, because these formulas are colored in green. The formulas representing the postcondition and the assignable clause are colored in orange, so KeY did not completely evaluate them or did never try to evaluate them. The path condition of the antecedent of Node 190 indicates that **`this`** and `s` are aliased (`self = s`) which is suspicious. Additionally, the current state shown in view **Variables** indicates that in case of aliasing the value of `s.value` is doubled. Consequently, the postcondition does not hold in case of aliased objects. If the initial state is for instance **`this`** `== s &` **`this`**`.value == 1`, then the postcondition as specifies evaluates to `2 == 1 + 2`. The correct version of the postcondition would be `value ==` **`\old`**`(value + s.value)`. An alternative to changing the postcondition is to add the precondition **`this`** `!= s`.

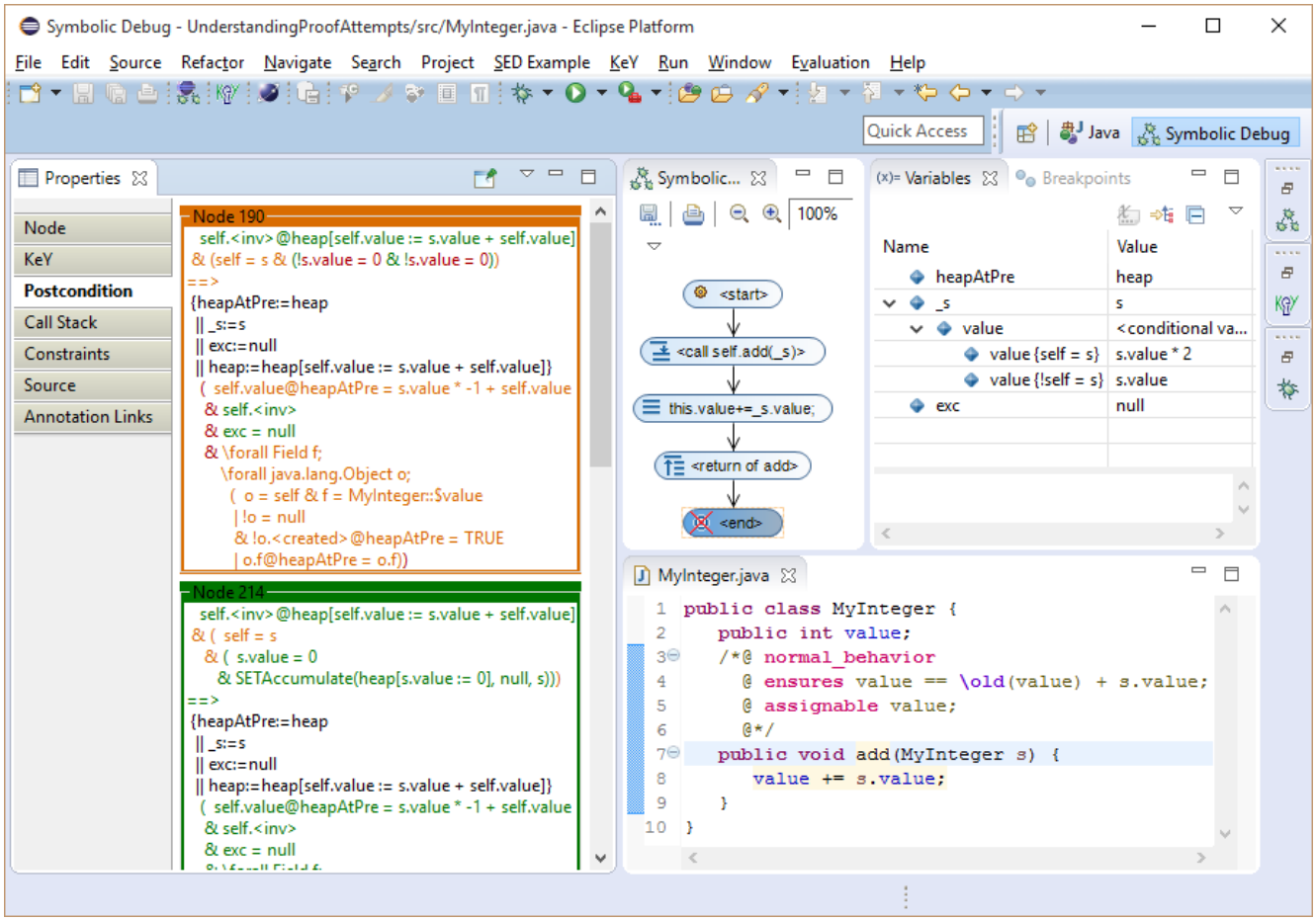


Figure 6.14.: Truth Statuses of a Defective Specification

6.8 Symbolic Execution Tree Slicing

The depth of a symbolic execution tree grows with the number of executed statements. In particular, this is the case when specifications are not considered, thus methods are inlined and loops are unrolled. A typical task in debugging is to find statements which contribute to a value of interest. With slicing (see Section 5.2), this can be done automatically. In the SED, slicing is triggered by selecting context menu item **Slice Symbolic Execution Tree** of a variable in view **Variables**. This opens a dialog in which the user can select the slicing technique to perform. Future work will also allow one to select a memory layout of interest. All symbolic execution tree nodes which are part of the slice are finally highlighted in a user defined color.

Figure 6.15 shows the thin slice of the example presented by Sridharan et al. [118]. For the location `z.f` at node $\textit{B v = z.f;}$ the slice contains nodes $\textit{w.f = y;}$ and $\textit{B y = new B();}$ which are colored in pink.

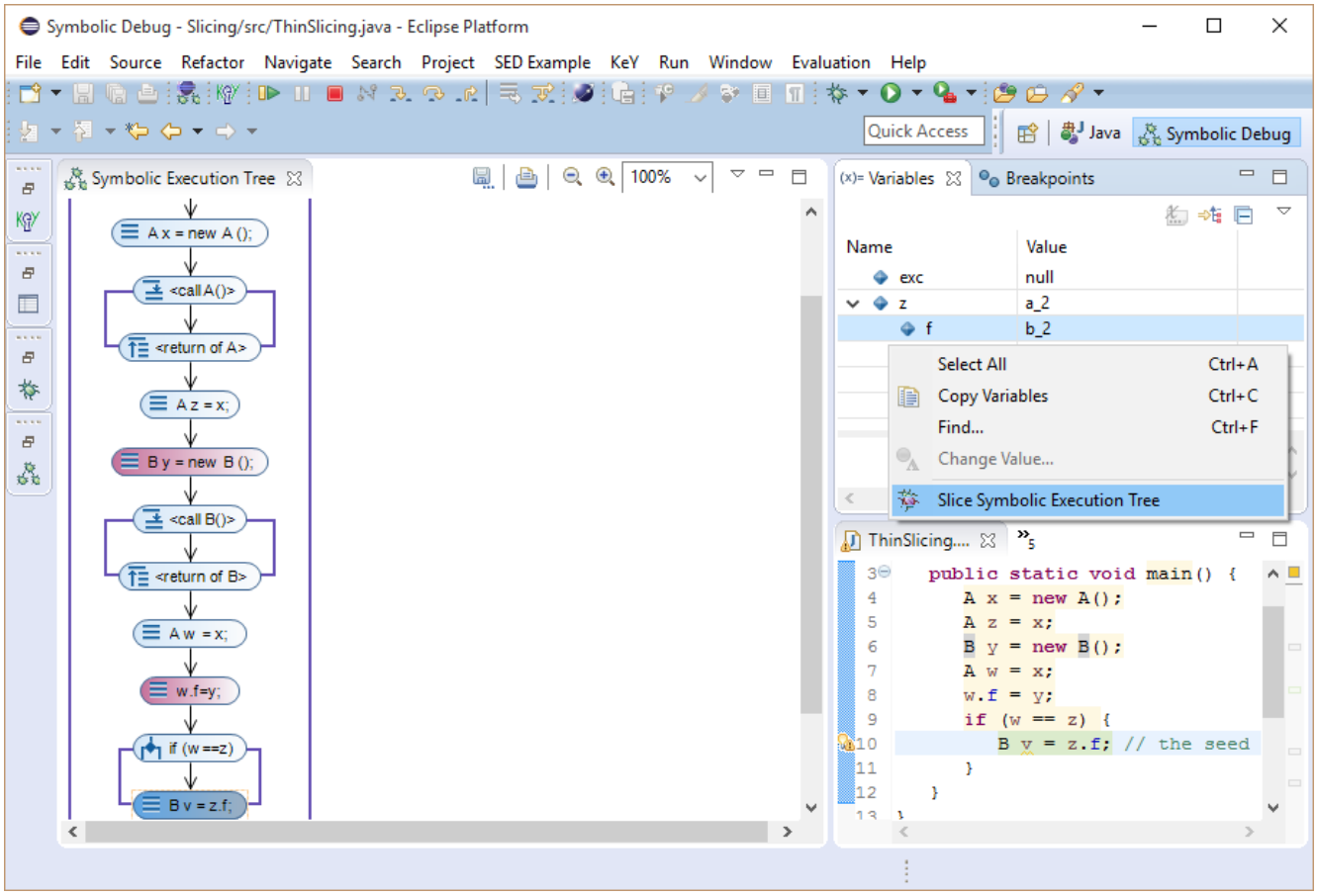


Figure 6.15.: Slicing a Symbolic Execution Tree (Source Code Taken from [118])

6.9 Architecture

The Symbolic Execution Debugger (SED) is an Eclipse extension and can be added to existing Eclipse-based products. In particular, SED is compatible with the Java Development Tools (JDT) that provide the functionality to develop Java applications in Eclipse. To achieve this and also a user interface that seamlessly integrates with Eclipse, SED needs to obey a certain architecture, which is shown in Figure 6.16. The grey-colored components are part of Eclipse (see Section 2.4.1), whereas the remaining components are part of the SED extension.

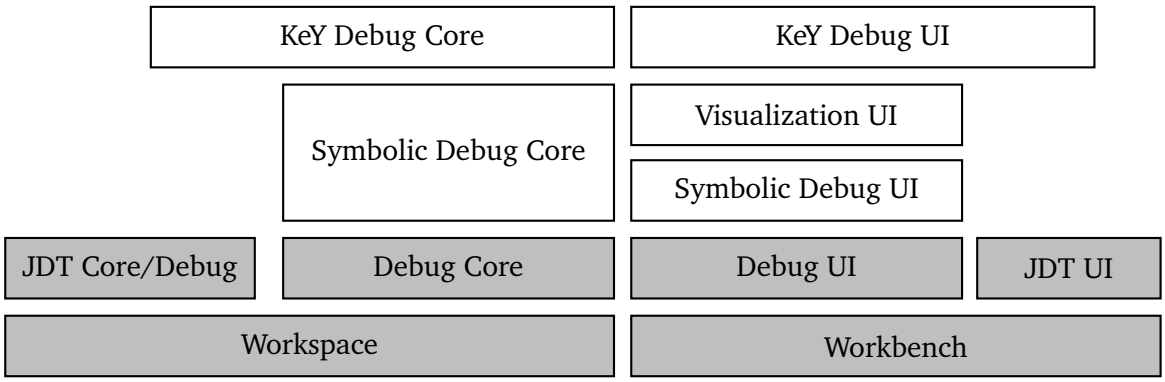


Figure 6.16.: Architecture of the Symbolic Execution Debugger (SED)

The foundation is the *Workspace* which provides resources such as projects, files and folders, and the *Workbench* which provides the typical Eclipse user interface with perspectives, views and editors. On top of these, Eclipse implements the debug platform, which defines language-independent features and mechanisms for debugging. Specifically, *Debug Core* provides a language-independent model (see Section 2.4.2) to represent the program state of a suspended execution. This includes threads, stack frames, variables, etc. *Debug UI* is the user interface to visualize the state defined by the debug model and to control execution. *JDT Core* defines the functionality to develop Java applications, including the Java compiler and a model to represent source code, whereas *JDT Debug* uses the debug platform to realize the Java debugger. Finally, *JDT UI* provides the user interface which includes the editor for Java source files.

The Symbolic Execution Debugger is based on the components provided by Eclipse. First, it extends the debug model (see Section 2.4.2) to be able to represent symbolic execution trees (see Section 6.10). Second, it provides a specific implementation based on KeY's symbolic execution engine (see Chapter 4), called KeY's SED integration. An experimental implementation based on JPF–SE [10] also exists.

Symbolic Debug Core extends the debug model to represent symbolic execution trees. This is done in a way that is independent of the target programming language and of the used symbolic execution engine. It is also necessary to extend the debugger user interface, which is realized in *Symbolic Debug UI*. It contains in particular the tree-based representation of the symbolic execution tree that is displayed in the **Debug** view (see top left view in Figure 6.6). The graphical representation of the symbolic execution tree shown in the **Symbolic Execution Tree** view as well as the visualization of memory layouts is provided language-independently by *Visualization UI*. Finally, *KeY Debug Core* implements the symbolic debug model with help of KeY's symbolic execution engine. The functionality to debug selected code and to customize the debug configuration is provided by *KeY Debug UI*.

The extendable architecture of SED allows one to reuse the symbolic debug model to implement alternative symbolic debuggers while profiting from the visualization functionality. All that needs to be done is to provide an implementation of the symbolic debug model for the target symbolic execution engine.

6.10 Symbolic Debug Model

The Eclipse Debug Platform[6] provides language independent facilities for debugging. This is achieved by a language independent debug model (see Section 2.4.2) which is implemented for different languages like Java. Once a program is launched, it is represented as `ILaunch` and provides access to the debuggable execution context. The debuggable execution context is defined by `IDebugTarget` instances and allows one for instance to list the currently running threads (`IThread`). How to write an Eclipse debugger is explained by Wright and Freeman-Benson [136] in detail.

The debug model reflects the structure of running programs and is not designed for symbolic execution by default. But it can be reused and extended for symbolic execution as shown in Figure 6.17.

If something is launched symbolically (`ILaunch`), the debuggable execution context is defined by `ISEDebugTarget` instances which is a subtype of `IDebugTarget`. It provides access to the root of a symbolic execution tree represented as `ISEThread` instance, a subtype of `IThread` for compatibility reasons. All nodes within a symbolic execution tree are subtypes of `ISENode` that allows one to access child nodes and the parent node. An `ISENode` is again a subtype of `IStackFrame` for compatibility reasons. `ISEStatement`, `ISEBranchStatement`, `ISELoopStatement` and `ISELoopCondition` represent different kinds of statements. A method call treated by inlining is represented by an `ISEMethodCall` and the return of the called method by `ISEMethodReturn` or `ISEExceptionalMethodReturn` instances. Alternatively, method calls can be treated by applying contracts (`ISEMethodContract`) and loops by applying a loop specification (`ISELoopInvariant`) instead of unrolling it. If execution splits into several branches `ISEBranchCondition` nodes show the condition under which each path is taken. Finally,

[6] www.eclipse.org/eclipse/debug

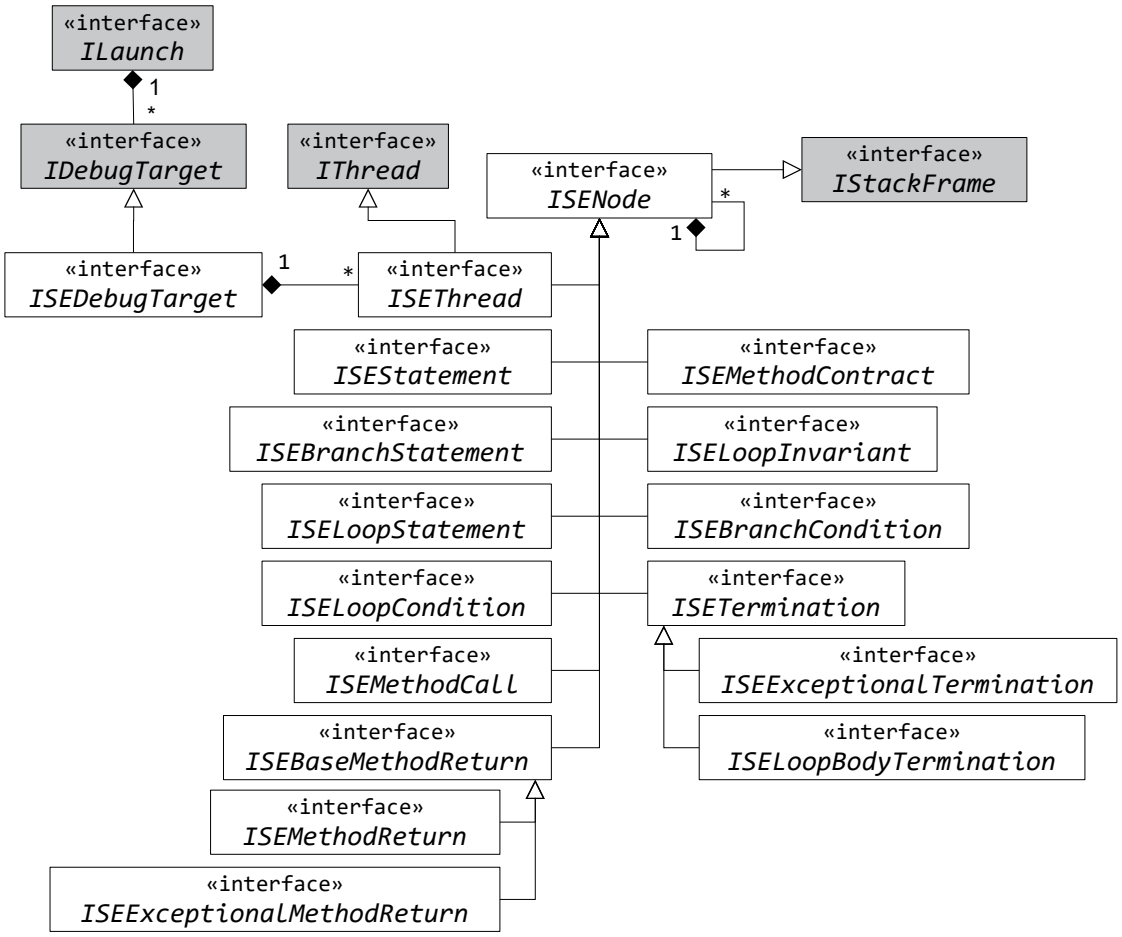


Figure 6.17.: Simplified Symbolic Debug Model

symbolic execution terminates normally (`ISETermination`) or exceptionally with an uncaught exception (`ISEExceptionalTermination`). Branches executing only the loop body after an applied loop specification end usually in an `ISELoopBodyTermination` node as long as the loop does not terminate abnormally.

An implementation of the symbolic debug model is registered and launched like any other debug model implementation (see [136]). The only difference is that the created instances also realize the interfaces of the symbolic debug model. Abstract classes exist covering already most of the work to be done. An example implementation launching a fixed symbolic execution tree as starting point for a new integration of a symbolic execution engine can be found at www.key-project.org/eclipse/SED/example.zip.

6.11 Annotation Model

Analyses based on symbolic execution need to present results to the user. The SED supports this by the annotation model. It allows one to highlight selected symbolic execution tree nodes and to attach additional information to them. The SED offers by default annotations for slicing, searching, comments and breakpoints. An example for each of them is shown in Figure 6.18. Method `fac` computes the factorial of `n`. Symbolic execution was started using precondition $n >= 0$ and the loop is unrolled twice. The annotations attached to the symbolic execution tree are shown in the **Properties** tab **Annotations** when an `ISEDebugTarget` or its parent `ILaunch` is selected in view **Debug**.

The first annotation in Figure 6.18 represents the slice of `f` at node $\textcircled{f *= i;}$ in the second loop iteration (leaf of left most branch). Nodes part of the slice are highlighted by a purple background. The second annotation defines that nodes hitting a breakpoint are highlighted by a green background. The third

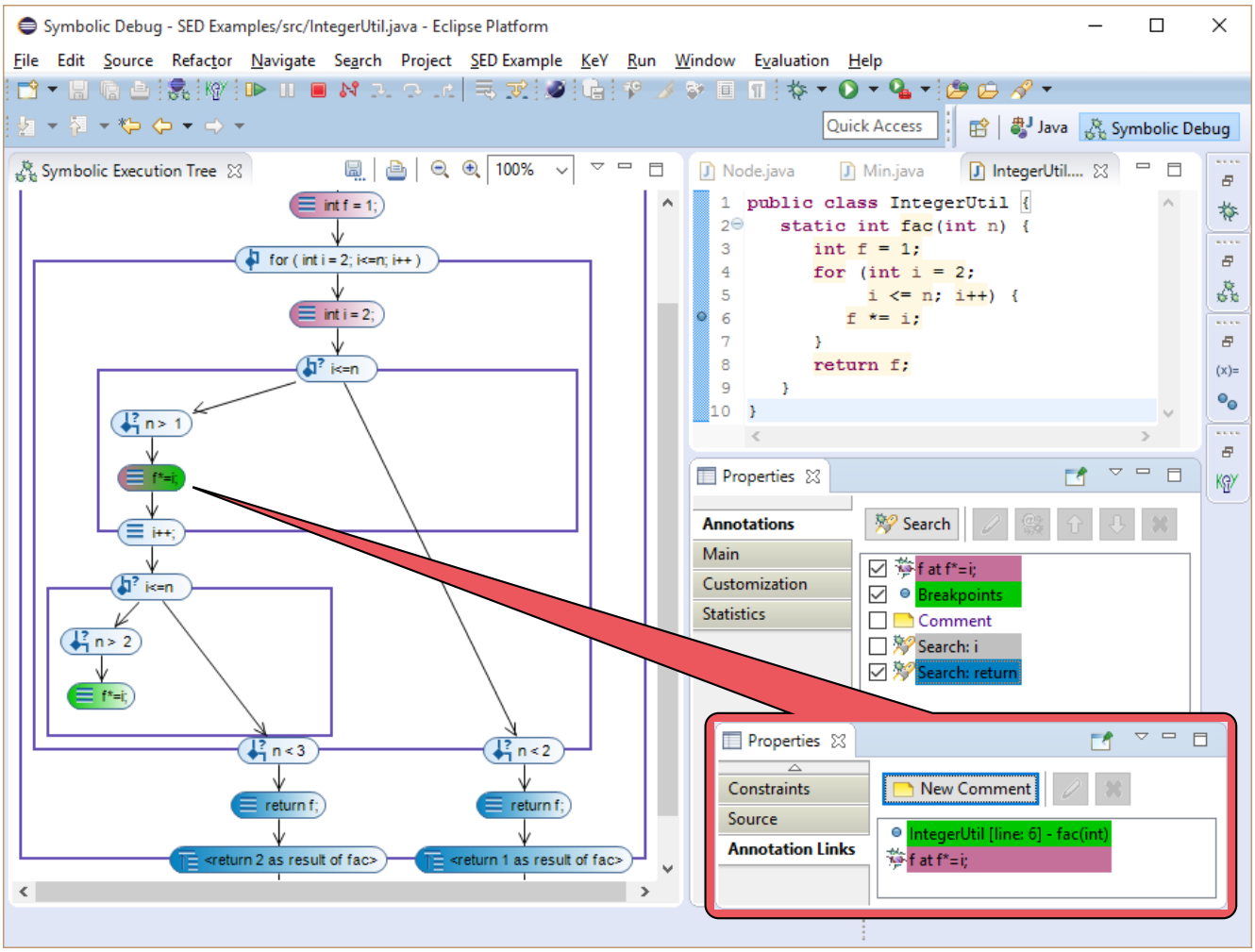


Figure 6.18.: Default Annotations of the SED

annotation allows one to attach comments to a node and the fourth contains the results of searching text "i" in the labels of symbolic execution tree nodes. Like the unchecked boxes indicate, the highlighting of nodes is disabled for both annotations. The last annotation represents results of searching text "return" highlighted by a blue background.

All annotated symbolic execution tree nodes are colored accordingly in view **Symbolic Execution Tree**. If a node has multiple annotations, a color gradient is used.

The callout in Figure 6.18 shows the annotations attached to node $\langle f *= i; \rangle$. The breakpoint at line 6 is hit and the node is part of the slice.

Figure 6.19 shows the annotation model. An `ISEAnnotation` defines what is highlighted and is attached to `ISEDebugTarget` instances. As it is a subtype of `ISEAnnotationAppearance` it defines also how nodes are highlighted. `ISEAnnotationLink` instances define which `ISENode` instances are annotated.

For each annotation implementation exists an `ISEAnnotationType` which is used as factory to create the `ISEAnnotation` and the related `ISEAnnotationLink` instances. The `ISEAnnotationType` is also responsible for persistence as each implementation can be arbitrarily parameterized.

Implementations of the annotation model can be but are in general not tied to a symbolic debug model implementation. The `ISEAnnotationType` implementation needs to be registered using extension point `org.key_project.sed.core.annotationTypes`. Besides the usual Eclipse extension points to extend the user interface allows extension point `org.key_project.sed.ui.annotationActions` and

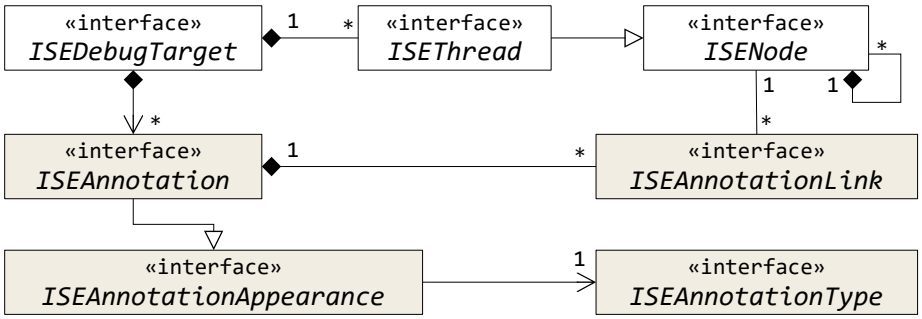


Figure 6.19.: Annotation Model

extension point `org.key_project.sed.ui.annotationLinkActions` to add additional buttons to the annotation properties tabs. Extension point `org.key_project.sed.ui.annotationTypeImages` allows one to register annotation type images. User interface controls to edit an `ISEAnnotation` instance are registered in extension points `org.key_project.sed.ui.annotationEditors`. At last, extension point `org.key_project.sed.ui.annotationLinkEditActions` contains actions opening a shell to edit `ISEAnnotationLink` instances.

The SED provides the user interface to select a slicing algorithm and to execute it. Each symbolic debug model implementation is responsible to offer the available slicing algorithms for the reasons discussed in Section 5.2.2.

7 An automatic proof manager to reduce user interaction

A major challenge in software development is to keep the different artefacts such as source code, comments, test cases and manuals up-to-date. Changes in the source code are not reflected in comments and as soon as artefacts are not managed in a single source, which is often the case for manuals or other technical documents, the gap widens.

Adding static analyses, such as deductive verification, to a software development process, aggravates the problem of keeping artefacts synchronized. A systematic analysis of the dependencies and how they can be resolved becomes mandatory for practical formal verification. So which additional tasks and artefacts does software verification add to the software development process? Obviously, there is the specification of the intended program behavior in a formal specification language like (Event-)B [1] or the Java Modeling Language (JML) [93]. Second, the proof that a program adheres to its specification. Constructing proofs is generally expensive. Depending on the complexity and expressivity of the specification language, program logic and precision of the underlying verification system, user interaction might be needed to complete the proofs. But also completely automatic approaches (which of course might fail to prove that a correct program is correct) for non-trivial properties take a long time. The artefacts produced are source code, specifications and proofs.

To illustrate the dependencies between the artefacts consider the Java program specified with JML (see Section 2.1) shown in Listing 7.1. The specification consists of two method contracts encompassing three specification cases. The first specification case of method `update` requires that `canUpdate` returns true and ensures that `balance` is updated by the given amount. Otherwise, if `canUpdate` returns false, the second specification case is applicable, which states that an exception is thrown without changing anything. The method contract of `canUpdate` is always applicable and returns true if the update will not exceed the overdraft limit.

The verification argument that the two methods satisfy their respective contracts gives rise to three verification conditions (proof obligations), and hence proofs[1].

As it happens, the given specification is defective. The method contract for `canUpdate` promises in its postcondition that the sum of the account balance and the amount argument is greater than the overdraft limit, whereas the implementation only guarantees greater-or-equal. Consequently, either the implementation or the specification needs to be changed. Changing the implementation requires to redo the proof that `canUpdate` satisfies its contract (and preserves the invariants), but all other proofs remain valid provided that they use the method's contract and not its implementation. Changing the contract of `canUpdate`, however, requires to redo not only the proof for `canUpdate`, but also the proof for `update` which uses the contract to represent the effect of the invocation of `canUpdate` in its conditional statement (and in its contract).

The example has a rather simple structure regarding implementation and specification, so all proofs should close automatically. In more realistic examples it might be necessary to assist the prover by performing some proof steps interactively. More complex specifications make also use of invariants and advanced framing concepts (**`assignable`** clause) which introduce additional verification conditions and a higher degree of dependencies among different proofs. In such a context it becomes challenging to keep all artifacts synchronized. In particular, in the presence of continuous changes to implementation and specification, the whole process has to be efficient. Otherwise, the user will turn the feature off. Hence, to determine the smallest set of proofs that need to be redone as well as to communicate the failure to find proofs (and thus potential defects) to the user is crucial for practical adoption of formal verification.

[1] Depending on the verification system certain verification conditions might be combined into one proof or split up into several subproofs.

```
1  package banking;
2
3  public class Account {
4     private /*@ spec_public @*/ int balance;
5     private /*@ spec_public @*/ int overdraftLimit;
6
7     /*@ normal_behavior
8       @ requires canUpdate(amount);
9       @ ensures balance == \old(balance) + amount;
10      @ assignable balance;
11      @ also
12      @ exceptional_behavior
13      @ requires !canUpdate(amount);
14      @ signals (Exception) true;
15      @ assignable \nothing;
16      @*/
17    public void update(int amount) throws Exception {
18       if (canUpdate(amount)) { balance += amount; }
19       else { throw new Exception(); }
20    }
21
22    /*@ normal_behavior
23      @ ensures \result == balance+amount > overdraftLimit;
24      @*/
25    public /*@ pure @*/ boolean canUpdate(int amount) {
26       return balance + amount >= overdraftLimit;
27    }
28 }
```

Listing 7.1: Example Java Class Specified with JML

Section 7.1 introduces the concepts implemented by (almost) any integrated development environment (IDE) to manage development projects. Then, Section 7.2 introduces *proof dependencies*, a new concept, which allows one to determine the (sub-)set of proofs that might become obsolete due to a change and need to be redone. Further, Section 7.3 presents the concept on how to integrate an interactive verification tool into an IDE. KeY Resources, the concrete implementation of the concept based on Eclipse and KeY is presented in Section 7.4. The evaluation of the approach and of the proposed optimizations is presented later in Section 9.3.

7.1 Basic IDE Concepts

A *project* has a unique name and is the root of a structured collection of resources. Resources are of different kinds: source code, libraries, or meta information such as the build path and settings. IDEs present the project structure usually as a tree resembling the standard rendering of file systems. Source code is usually displayed as a subtree of the project tree, where the inner nodes correspond to modules or packages and the leaves represent the actual classes or files.

A *marker* is a tag that can be attached to resources or content contained in a resource. Markers have a kind (e.g., *information*, *warning*, *error*), a position, and a description text. For instance, in case of a compilation error an error marker can be positioned at the statement or line of code causing the error. IDEs visualize markers in several ways: as a list of errors and warnings in a separate view or as an icon shown within an editor next to the marker's position.

7.2 Proof Dependencies

To determine efficiently which proofs must be redone in presence of a change, Definition 7.1 introduces the concept of proof dependencies. It assumes a program annotated with specifications given as context.

Definition 7.1 (Proof dependency)**.** *A* proof dependency *is a pair* (*proof obligation, target descriptor*) *linking a proof obligation to a target descriptor. A* target descriptor *represents a program or specification element used in the proof of the proof obligation. A proof dependency and a change to the context program is called*

- dangling*, if the program or specification element referred to by the target descriptor does no longer exist;*
- tainted*, if the change might effect the evaluation or execution of the program or specification element referred to by the target descriptor.*

Proof dependencies capture the dependencies of a particular proof to source code or specification elements and are thus proof dependent. Depending on the kind of program or specification element and its treatment by the calculus used as basis for the verification system, different kinds of proof dependencies can be distinguished, see Table 7.1.

Proof Dependency	Description
Method Invocation	Such proof dependencies link a proof obligation to a method invocation descriptor $(m, ct, ctxt)$ where m denotes the signature of the called method, ct the static type of the callee and $ctxt$ the class where the method invocation occurred. Such a proof dependency is created when a proof contains an explicit case distinction over all possible implementations of a method to evaluate a given method invocation statement. A change causes a tainted (or even dangling) proof dependency if it removes and/or adds a new binding for the described method invocation.
Method Inlining	Such proof dependencies link a proof obligation to a method implementation (m, ct) with ct the class containing an actual implementation of method m. Proof dependencies of this kind are created when the verification system inlines a method. Such a proof dependency becomes dangling when method m has been removed and tainted if its implementation has been changed.
Use Contract	Such proof dependencies link a proof obligation to a contract of a method and are created whenever the verification system uses a method contract instead of inlining the method. It becomes dangling if the contract has been removed and tainted in case the contract has been changed.
Field Access	Such proof dependencies link a proof obligation to a field access descriptor (fd, ct) with fd denoting a field declaration fd and ct the static type of the reference prefix. They are created whenever a field is accessed in source code or specifications. A change taints a field access proof dependency if the field declaration has changed or if the parent hierarchy has changed. A dangling proof dependency is caused if the field has been removed altogether.
Use Invariant	Such proof dependencies link a proof obligation to an instance or static invariant. They are created whenever a property assured by an invariant is used in a proof. They become dangling if the invariant has been removed and tainted in case the invariant has been changed.

Table 7.1.: Kinds of Proof Dependencies

7.3 Integrated Proof Management

This section describes the approach to integrated proof (or analysis) management. The concept was developed with semi-automatic verification systems in mind, but also automatic or completely interactive verification or static analysis systems in general can profit from it. The approach is designed for modular verification which gives rise to multiple proof obligations.

The concept consists of an appropriate file structure (Section 7.3.1) and of an automatic update process (Section 7.3.2). Section 7.3.3 summarizes the requirements on verification tool and IDE needed to realize the approach.

7.3.1 Proof Storage and Proof Markers

Proofs are the central artefacts produced by software verification tools. To manage proofs, the IDE project is extended with an additional resource kind for proofs. Managing proofs as part of a project is advantageous in many ways: (i) the user has direct access to the proofs and can inspect or manipulate them; (ii) if a version control system is used, then source code, specifications *and* proofs are committed and updated together ensuring their synchronization. Further, it is possible to compare different versions of a proof.

For a concise representation of verification results, markers as described in Section 7.1 are used to indicate the status of proof obligations. *Information markers* indicate successful (closed) proofs, possibly with a hint that not all used specifications (theorems) have yet been proven. *Warning markers* are created for open proofs and provide details of the reason why a proof was not successful (timeout or unclosable goal detected). *Error markers* with a failure description indicate syntax errors in specifications or failed global correctness checks (such as cyclic dependencies).

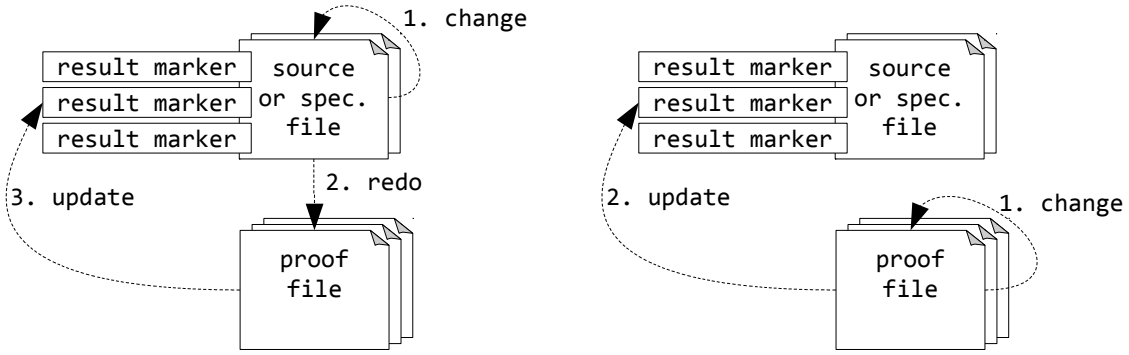
7.3.2 Update Process

The proof manager is responsible to keep (i) source code, specification and proofs synchronized as well as (ii) the verification status information (in the form of markers) up-to-date.

Changes to source code and/or specifications may invalidate existing proofs or give rise to additional proof obligations whose proofs (proof attempts) are stored as new proofs. The markers need then to be updated to reflect the correct status. This scenario is shown in Figure 7.1a.

The second source of change is a modified proof, for instance, after the user performed some interactive proof steps. In this case only the result markers need to be updated to reflect the new status (see Figure 7.1b). Changing the status of one proof might trigger status changes of dependent proofs, for example: the proof that a method m adheres to its specification could be closed but uses the contract of method n. Hence, the overall correctness of the proof for m depends on the proof that method n adheres to its specification.

Please observe that the update process is completely automatic and does not require any user interaction. If a proof cannot be closed automatically, the proof attempt is stored and a result marker informs the user about the proof state. After completion of the update process, the user can choose to finish a proof attempt interactively which will then trigger a new update process.



(a) Change of Code or Specifications **(b)** Change of Proofs

Figure 7.1.: Change Handling

The update process for source code or specification changes (first scenario) is now described in detail. It contains the process for proof changes (second scenario) as a subroutine. The update process is triggered whenever a change occurs, for instance, every time a file is saved. The naïve approach to simply redo *all* proofs upon each change turns out to be too inefficient and does not scale to larger projects. Instead Algorithm 7.1 is proposed to reduce the overall verification time. The algorithm provides several anchor points at which different optimizations can be plugged in.

input: A project p and a list of changes ci (change information) in p

```
1 allProofObligations ← listAllPO (p);
2 pendingProofObligations ← filter (allProofObligations, ci);
3 pendingProofObligations ← sort (pendingProofObligations);
4 foreach po in pendingProofObligations do
5 |   proofResult ← doProof (po);
6 |   showInfoOrWarning (proofResult);
7 end
8 cycles ← checkCycles (allProofObligations);
9 showError (cycles);
```

Algorithm 7.1: Update Process (Source Code or Specification Change)

Upon a change the IDE informs the proof manager and provides the affected project as well as additional information about the change. The detail of the provided change information depends on the IDE and the nature of the change. It can range from detailed information like renaming of a method or field to a simple list of files that have been changed.

When the update process is triggered by a source code or specification change, it first retrieves all proof obligations available for the project (line 1). In a second step (line 2) the relevant proof obligations for which proofs have to be redone are determined and (line 3) prioritized. The proof obligations are then processed in order of their priority and proof attempts are initiated (line 5). The status of each performed and completed proof attempt is updated (line 6).

When all proof attempts have been completed (successful or not), a global correctness check (line 8) is performed to ensure that no cyclic proof dependencies exist (e.g., to avoid that when proving total correctness of a recursive method its own contract is used to eliminate the recursion). The global correctness check must look at *all* proof obligations and their respective proof dependencies. Finally, the result markers are updated again (line 9) to reflect the result of the global correctness check. In case of a proof change only the global correctness check and an update of the result marker have to be performed.

For a naïve approach the implementation of the procedures `filter` (line 2) and `sort` (line 3) simply return the list of proof obligations given as their argument, whereas method `doProof` starts a new proof search for each proof obligation. In the following, an optimized implementation for each of these procedures is presented to achieve a significantly reduced proof effort and to provide faster feedback to the user.

Optimization Selection

Proofs are modular in the sense that they rely only on specific parts of the implementation and specification. Hence, method `filter` can remove all proof obligations for which the related proof is known not to be affected by the change. To this end method `filter` retrieves the stored proof dependencies for each proof obligation and checks whether the change caused a tainted or dangling proof dependency (see Section 7.2). Proof obligations with no tainted or dangling proof dependencies are filtered out. The achieved precision depends on the granularity with which changes are recorded (changed files, code fragments, specification elements, etc.). Proof obligations remaining from previous changes (e.g., if a previous change was not well-formed and caused compilation errors) have to be returned as well.

Optimization Prioritization

For usability reasons it is important to provide feedback about the status of the different proofs to the user in a timely manner. This is achieved by (i) updating the proof status after each completed proof attempt (line 6) and by (ii) prioritizing the proof-obligations to be proven (line 3). Prioritization takes into account that the user is not interested in all proofs to the same degree. A developer changing the specification of a method *m* has an immediate interest to know whether that method still satisfies the modified contract; once this is achieved, it is proven whether the modified contract is (still) sufficiently strong to prove the correctness of methods invoking *m*. A prioritization (in descending order) might be as follows: proof obligations for the currently selected element → proof obligations for the currently selected type → proof obligations for the currently selected file → proof obligations for other opened files → all other proof obligations.

Optimization Proof Replay

This optimization concerns the implementation of `doProof()`. Proof replay is usually faster than proof search for two reasons:

- if closing a proof required user interaction, the interactive steps are saved and performed automatically during proof replay (if still applicable);
- if the proof format stores each performed step (and not only a script), proof replay avoids expensive proof search strategies completely.

To benefit from proof replay, `doProof()` proceeds as follows: if no saved proof exists then proof search is used. Otherwise, proof replay is attempted. Proof replay may complete with two outcomes: (i) either the saved proof can be replayed completely or (ii) the proof replay stops at some point, because some proof step is no longer applicable (due to the performed change). In the latter case, proof search is initiated to attempt to close the proof. Note that even in case (i) the replayed proof might not be closed in case the change did not affect that proof and a previous proof attempt was unsuccessful. In that case, automatic proof search is not started, but the user is asked to finish the proof interactively.

If the underlying verification system provides more intelligent proof reuse strategies [111, 86] than simple proof replay, these can be used instead (or in combination) to reduce the verification effort.

Optimization Parallelization

The for-loop of Algorithm 7.1 can be parallelized to execute several proofs concurrently. The time required for proof search differs from proof to proof and thus it is not advisable to have a fixed assignment from proof obligations to threads, but to use pooling instead. The update process can be even distributed to other computers as long as it reduces the overall time.

7.3.3 Requirements

To implement the proposed proof management and update process, the IDE and the verification tool need to satisfy some minimal requirements:

The verification tool must be able to list all proof obligations for a project, to instantiate a proof and start the proof search. For proof replay, the verification system must be able to save and replay proofs. For the optimized selection, the proof format of the verification tool must be proof producing and the proof format must contain enough information to extract proof dependencies.

The IDE must be extensible to add support for managing proofs inside a project and to listen for project changes to trigger the update process. Native support for markers is advantageous for a seamless integration, but not a necessity.

7.4 KeY Resources

The concept is realized by integrating the verification system KeY into Eclipse. Figure 7.2 shows a screenshot of the Eclipse integration called KeY Resources[2,3].

The **Package Explorer** view provides access to all files organized in a *KeY project* which extends the original Java project with the features of the integrated proof management as described in Section 7.3. The project structure is extended by a proof folder *proofs* which stores the proofs together with meta files that contain in particular the proof dependencies. For ease of navigation the proof folder reflects the hierarchy of the source folder.

The editor shows the file **Account.java** with the program from Listing 7.1. The **Outline** view lists basic code members for navigation purposes. The information marker in front of line 17 indicates that both proofs of method `update` are closed whereas the warning marker in front of line 25 indicates that the proof of `canUpdate` is still open. The user can directly open a proof in KeY to inspect or continue the proof interactively by using Eclipse's *quick fix* functionality. Additional quick fixes allow the user to inspect a proof in the Symbolic Execution Debugger (Section 6.7) and to generate test cases or counterexamples (Section 7.4.2). In addition to the markers, view **Problems** is used to summarize the proof results together with other detected issues.

A detailed overview of the verification progress is offered by view **Verification Status** as shown in Figure 7.3. Its tab **Proofs and Specifications** shows the project structure including all proof obligations. For an easy overview of the verification status of components, the *worst* proof result is propagated to ancestors. It is defined as the minimal element in the following ordered list (worst to best): (i) *cyclic proofs* (the usage of specifications forms a cycle), (ii) *open proof*, (iii) *unspecified* (no proof obligation available), (iv) *unproven dependency* (proof is closed, but an applied specification is not verified yet) and (v) *closed proof*.

Here, method `update` is proven, but not all used contracts are yet verified. Method `canUpdate` is not verified, which means that also class `Account`, its package and the full project is not verified. The default constructor of `Account` is highlighted as unspecified. Unspecified methods are dangerous, as they do

[2] KeY Resources is available at www.key-project.org/eclipse/KeYResources.

[3] The initial implementation of KeY Resources was implemented as part of the Bachelor's thesis by Käsdorf [83].

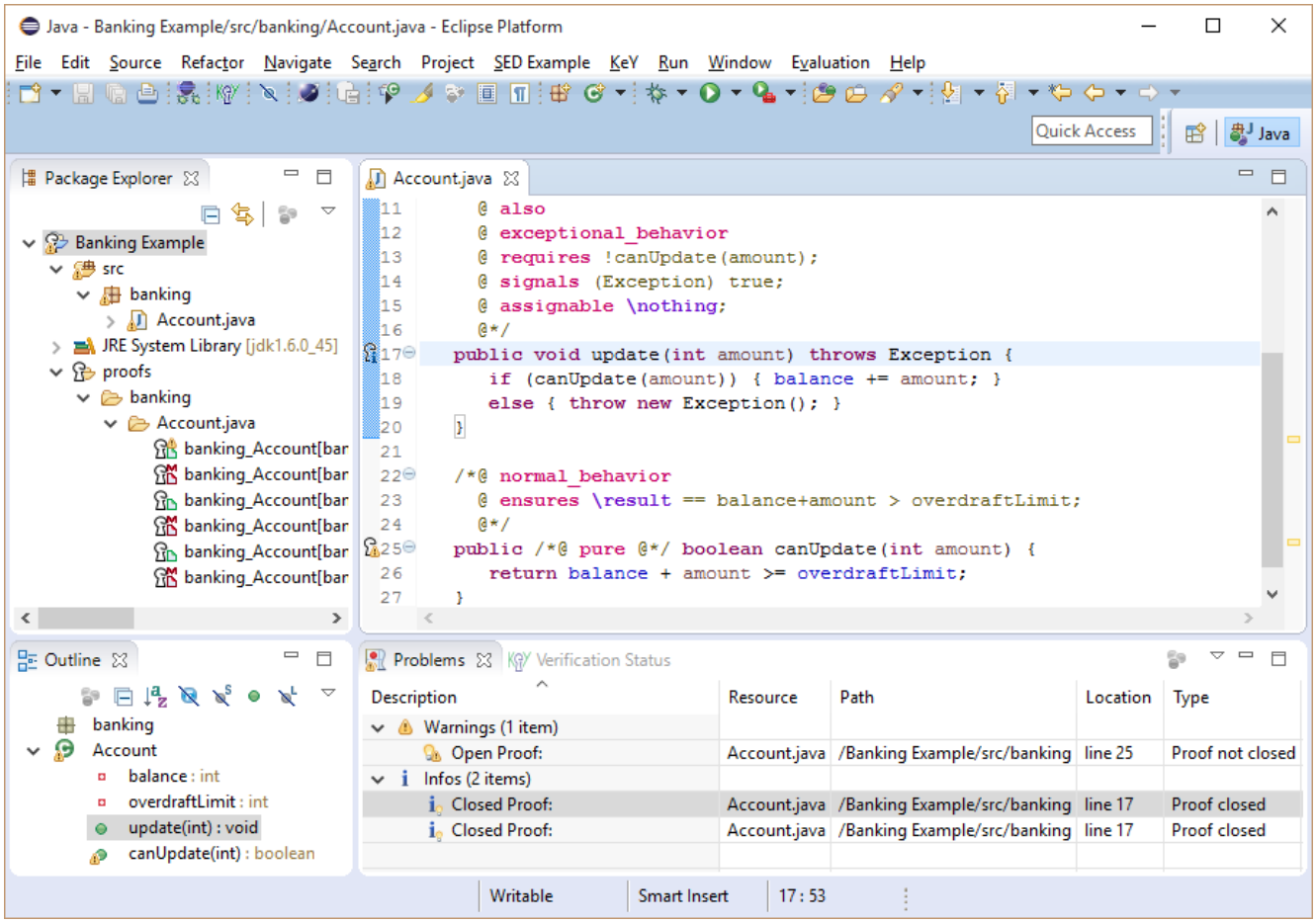


Figure 7.2.: Screenshot of KeY's Eclipse Integration

not spawn any proof obligation. Consequently, no proofs are performed which would ensure that the unspecified code does not call a specified method in a state violating its precondition.[4]

The icons of all proof obligations in Figure 7.3 are annotated with a warning marker. Warning markers appear when the proof of a proof obligation uses taclet options introducing rules that are unsound or incomplete with respect to the Java language specification [60]. Taclet options with such properties are used for instance (i) in teaching, because students coming in contact with verification for the first time should not be overchallenged by overflow semantics, or (ii) in research, when a simpler integer semantics avoids distractions from the actual research focus. Whether the drawback influences the validity of a proof for the current project needs to be decided by the user and that's why the tool informs about it. In this example, class initialization and overflows are ignored. At least class initialization is negligible as the code does not access any other class. Overflows can occur in the banking application and it is advisable to change the taclet options to consider them. The last two entries in the tooltip are just information.

All the information on tab **Proofs and Specifications** is also available as an HTML report on tab **Report**. In addition, the report contains a section about assumptions made, see Figure 7.4. Assumptions need special care as proofs assume them to hold. In this example, several proofs inline for instance API methods (listed under *1.*) which assumes a closed world. This means that the proofs have to be redone when the code is reused in a different project where the listed methods are overridden. Also general assumptions of KeY users might not be aware of are mentioned (*2.* to *7.*).

In the following, Section 7.4.1 discusses important details of the implementation. How test cases and counterexamples can be generated as part of the build process is presented in Section 7.4.2.

[4] In future work default specifications might be used to spawn proof obligations.

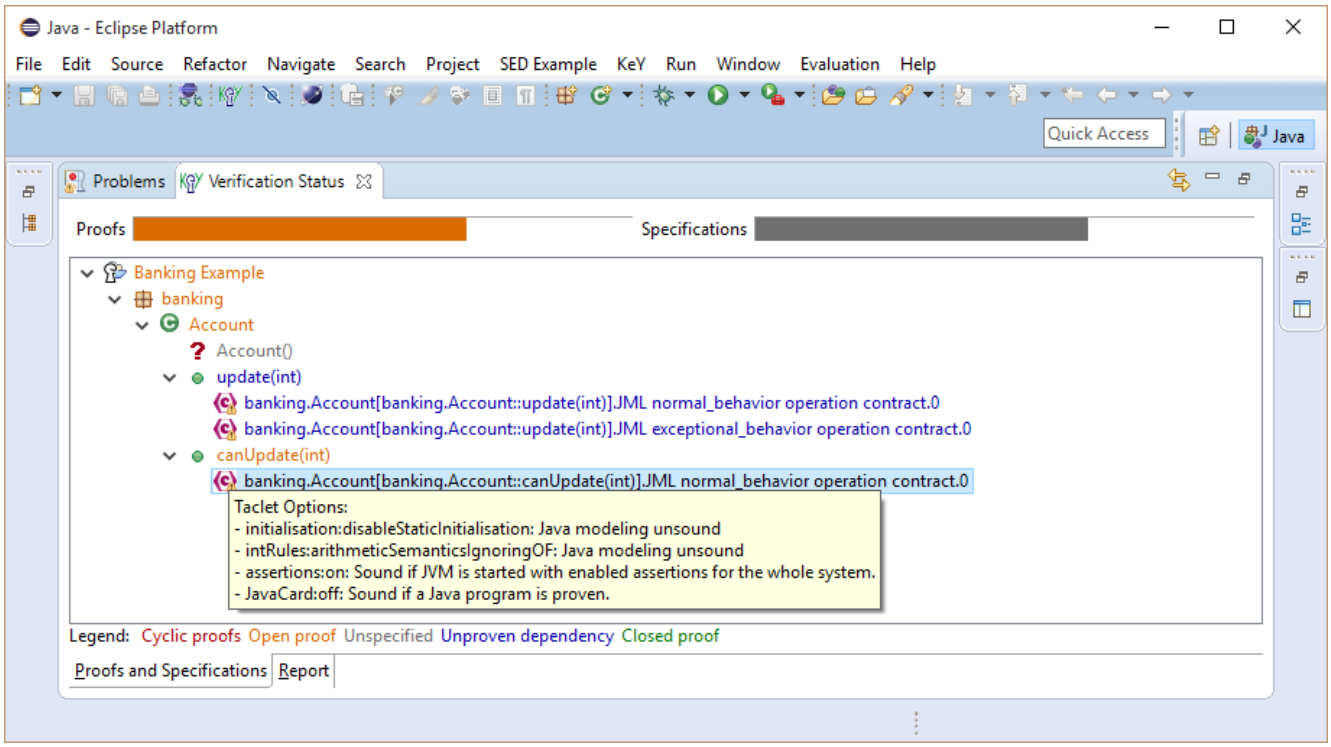


Figure 7.3.: Screenshot of Tab **Proofs and Specifications** of View **Verification Status**

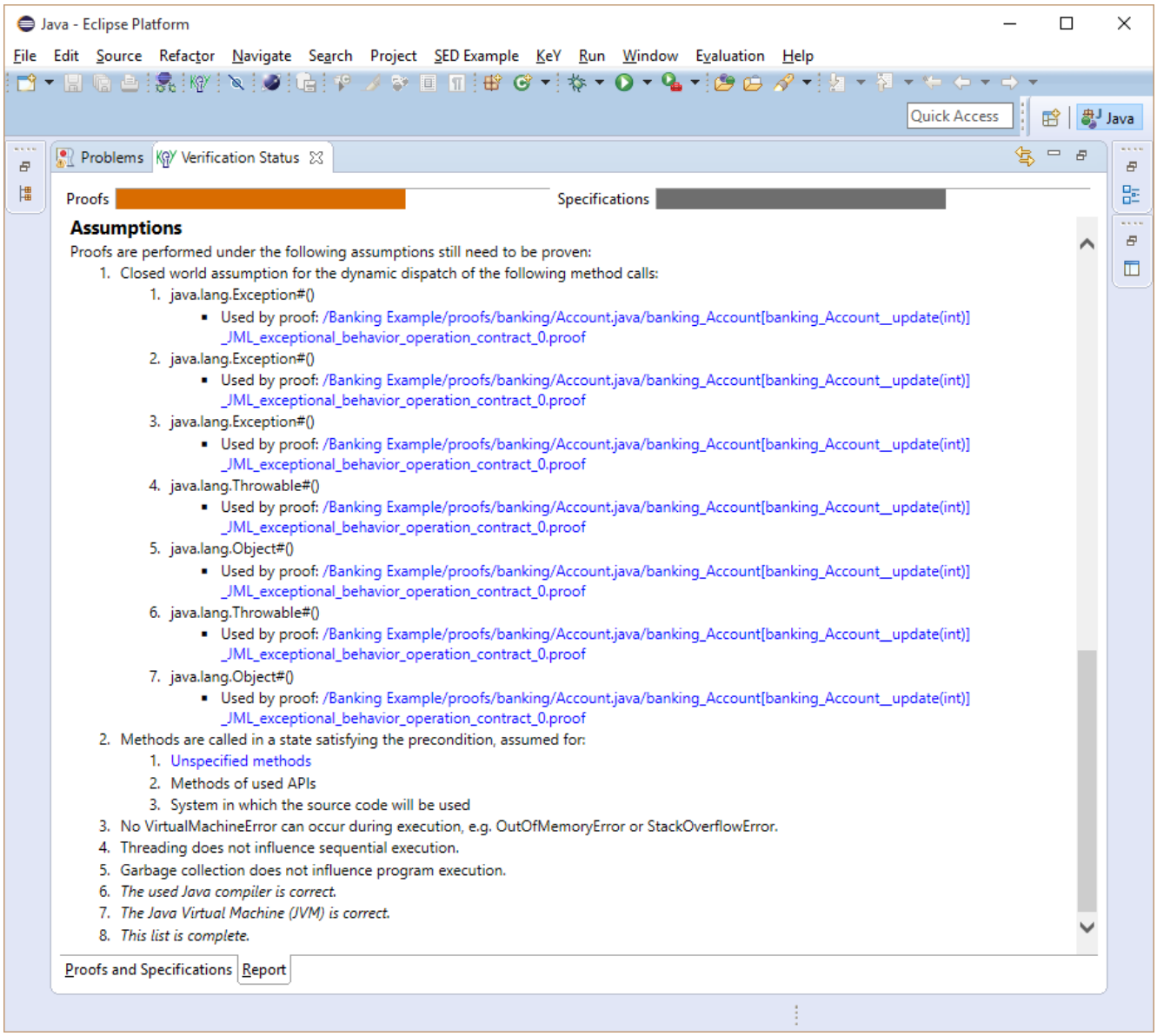


Figure 7.4.: Screenshot of Tab **Report** of View **Verification Status**

7.4.1 Implementation Details

Eclipse satisfies all requirements of the concept on the IDE side. Projects, folders and files are represented as a *resource* within Eclipse. A resource can be annotated with *resource markers* which are used to present proof results to the user. *Integrated project builders* are triggered when the project content has changed. More details about the mentioned workspace concepts can be found in [52].

While the build process is running, the user can continue editing but not change any resource. If she tries for instance to save a file while the build process is running, Eclipse automatically blocks the user interface until the build is done. As performing proofs usually takes some time, it is not advisable to do it as part of the build process for the mentioned reason.

Instead, the update process is implemented as a *job*, a thread with progress indicator in Eclipse. The job first analyzes the change (if available) and marks all affected markers as outdated. Then, the job performs all proofs with an outdated marker. An additional builder is only used to stop an already running job and to trigger a new one with the current change. As also the user can cancel the job or close Eclipse at any time, the job can also be triggered interactively and is automatically triggered when Eclipse is started.

Proof dependencies are determined by analyzing the applied Java DL rules. The dependencies can be directly extracted from each rule application according to Table 7.2. Hence, the computation complexity is linear to the proof size. Please observe that the dependencies have to be computed only once and are then stored in a meta file.

Proof Dependency	Number of Dependencies per Applied Rule	Criteria
Method Invocation	1	A rule performing a method call is applied. In Java DL, the name of such rules starts with “methodCall”.
Method Inlining	1	A method body statement is executed.
Use Contract	1	UseOperationContract rule is applied.
Use Invariant	1	The applied rule is generated from an invariant. Please observe that KeY generates a taclet for each invariant.
Field Access	*	One field access proof dependency for each location read or written by the executed statement/applied specification.

Table 7.2.: Proof Dependency Criteria

Optimizations suggested in Section 7.3.2 are also realized. Proofs are performed in parallel in a user-defined number of threads. If a stored proof is available, proof replay is attempted. Eclipse provides change information by default only on the file level. As a consequence, the selection optimization in simple form considers proof dependencies as tainted or dangling whenever the file containing the element referenced by the target descriptor has changed (or a file containing a subtype or supertype has changed).

Optionally, selection on changed elements within a file is available. The prerequisite to identify what has changed within a file is to know the old state. Here, the old state is the state when the proof was performed. This state might be different from the state before the change in case that the previous proof process was not completely executed. For each proof reference, the pretty printed content of the target descriptor is stored in the meta file as well. When a file change is detected by Eclipse, the selection optimization checks first for each proof reference if the target file is part of the change. If this is the case, then the new pretty printed version of the target descriptor is compared to the one stored in the meta file.

The use of pretty printing allows one also to ignore irrelevant changes in white space or comments. The proof reference is tainted, if the pretty printed content is different and dangling, if no longer available.

An evaluation of the implementation and the discussed optimizations is presented in Section 9.3.

7.4.2 Managing more than Proofs

The integrated proof management (Section 7.3) ensures that proofs are always up to date. Additional analysis based on proof results can be integrated in the proof process (Section 7.3.2) as well to ensure that also their results are always up to date.

KeY offers a counterexample and test case generation [46, 55, 56] facility. Both operate on an existing proof and are also available as part of the Eclipse integration. Whereas generated test cases are originally stored in a user defined folder, the Eclipse integration maintains test cases in an additional Java project. Whenever a proof is performed by the proof process, a test case named after the proof obligation is also generated. Additionally, a test suite listing all test cases is created as last step of the proof process. This ensures a full test suite of the specified code which is always up to date.

Found counterexamples are stored as part of the meta file. That a counterexample is available is shown by the tooltip of the warning marker which indicates an open proof. An additional quick fix allows one to inspect the counterexample.

8 Completing the Eclipse Integration

The Symbolic Execution Debugger (Chapter 6) and the proof management offered by KeY Resources (Chapter 7) are only two aspects of KeY's integration into Eclipse. To achieve an optimal user experience, additional features are needed which are presented in this chapter. First, editing facilities for JML including for instance syntax highlighting, auto completion and refactorings are offered by JML Editing (Section 8.1). Second, stubs need to be provided to use arbitrary API methods in proofs. Writing stubs by hand is laborious and error-prone and can be avoided by automatically generating them with Stubby (Section 8.2). Finally, the path to stubs and other KeY specific settings have to be defined. This and interactive verification in the original user interface of KeY is offered by KeY 4 Eclipse (Section 8.3). An alternative user interface for interactive verification which is deeply integrated into Eclipse is offered by the KeYIDE (Section 8.4).

The architecture of KeY's Eclipse integration is shown in Figure 8.1. SED, Stubby and JML Editing are completely independent from KeY. The source code of KeY including the symbolic execution engine (Chapter 4) is offered to Eclipse plug-ins by KeY 4 Eclipse. KeY 4 Eclipse provides also basic functionality to configure KeY specific settings. If both, KeY 4 Eclipse and Stubby, are available, KeY 4 Eclipse extends the stub generation for specific needs of KeY.

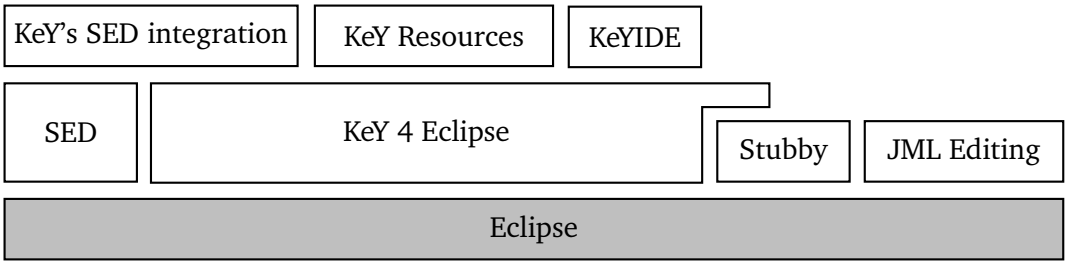


Figure 8.1.: Architecture of the Eclipse Integration

KeY's SED integration is realized on top of SED and KeY 4 Eclipse. Also KeY Resources and the KeYIDE are realized on top of KeY 4 Eclipse.

8.1 JML Editing

The Java Modeling Language (JML) [92, 93] was proposed in 1999 and is developed since then as community effort. Nowadays, many different tools working with JML are available[1]. A prominent example is the extended static checker ESC/Java [50], continued as ESC/Java2 [38] and finally replaced by OpenJML [35]. An Eclipse integration was available for the first time as part of the JML2 [27] tools set.

Although there exists an Eclipse integration of JML tools, until now no editing facility for JML is available. One reason might be that the Java Development Tools (JDT), which provide the functionality to develop Java applications in Eclipse, do not allow to extend the Java language. JDT supports for instance to extend the auto completion, the formatting or refactorings. The default mechanisms of Eclipse allow one to extend menus and to annotate source code. What is not directly supported is to extend the Java parser and the source code highlighting.

The following Section 8.1.1 explains how the syntax highlighting of JDT can be extended at runtime. The used JML parser is then presented in Section 8.1.2. The currently available features of JML Editing are explained in Section 8.1.3.

[1] www.eecs.ucf.edu/~leavens/JML/download.shtml

8.1.1 Extending Syntax Highlighting of JDT

Source code highlighting is a major feature a developer expects to be available. Eclipse can be extended with a separate editor for JML files. The implementation can be done from scratch or by reusing and extending the JDT code. The disadvantages are that the implementation is complicated and difficult to maintain. Also the user has to decide for each Java file whether it should be opened in the Java editor or the JML editor.

To avoid the disadvantages above, source code highlighting of JML specifications is realized by a modification of the Java editor which takes place at runtime. Text editors like the Java editor use an instance of `ISourceViewer` to show and edit text. How text is highlighted is specified by a `SourceViewerConfiguration`. The JML editing facility offers a plug-in which listens for newly opened Java editors. When a new Java editor is opened, the plug-in replaces the used `SourceViewerConfiguration` by a new one which is extendable. Requests are first delegated to the original `SourceViewerConfiguration`. The result can then be modified by the available extensions before it is passed to the caller. The same trick is also applied to manipulate the content shown in the outline of a Java editor.

8.1.2 JML Parser

The JML reference manual [93] describes the JML language, but for various reasons, JML tools usually operate on their own JML dialect. A JML dialect might be a subset of supported features from the reference manual. But a JML dialect can also offer additional features or modify the behavior of existing features in the reference manual.

The KeY dialect of JML supports for instance in addition to the modifier **`pure`** also **`strictly_pure`** which guarantees no side effects and that no new objects are created. KeY's JML dialect also modifies existing features, for instance by allowing to use the **`accessible`** clause to specify which locations are accessed by a model field. This is not only a new use of an existing feature, also a new parameter (the described model field) is introduced which is not present in the JML grammar of the reference manual.

As a consequence, JML specifications are written having a specific tool in mind. This should also be reflected by the used JML editor. JML Editing treats the problem with help of *JML profiles*. Each profile represents a JML dialect and provides for instance the parser to use. A parser is responsible to decide whether a given text is a JML comment and in case it is also to parse it. Code reuse between profiles is desired and supported by a combination of parsers. Each profile offers a list of supported keywords and in addition, for each keyword a parser responsible to parse the keywords content. A JML comment is parsed (i) by detecting a keyword, (ii) by determining the associated parser of the active JML profile for this keyword, and (iii) by parsing the keyword's content with help of the determined parser.

JML Editing operates on top of JDT. This means that JDT is still used to parse a source file. The JDT parser and the created AST are not extendable. JML Editing is part of the user interface and parses comments on demand.

Currently, two profiles are offered by JML Editing. One which captures almost the full KeY dialect of JML and one representing the reference manual. The profile for the reference manual supports currently only the JML features also supported by KeY. Users can derive new profiles from existing ones by disabling and adding keywords. The content of a new keyword needs to be parsable with an existing keyword parser. If this is not the case, JML Editing offers an extension point to integrate new profiles and thus new parsers.

8.1.3 Features of JML Editing

The goal of JML Editing is to make all features of JDT be aware of JML. The currently available features are listed in Table 8.1.

- Syntax highlighting for JML specifications with configurable colors
- Auto completion for JML keywords
- Auto indentation of JML comments while writing
- Code formatter preserves JML comments
- Hovers for JML keywords
- Error marker for syntax violations
- Rename and move refactoring support for JML

Table 8.1.: Features of JML Editing

Figure 8.2 shows a screenshot of Eclipse where JML Editing is enabled. It can be seen that the JML comments of `Account` are nicely highlighted. The opened auto completion offers JML keywords available in the current context. The error marker at line 8 indicates that the JML comment can't be parsed. The reason is given as tooltip, which is here that "assign" is not a supported keyword.

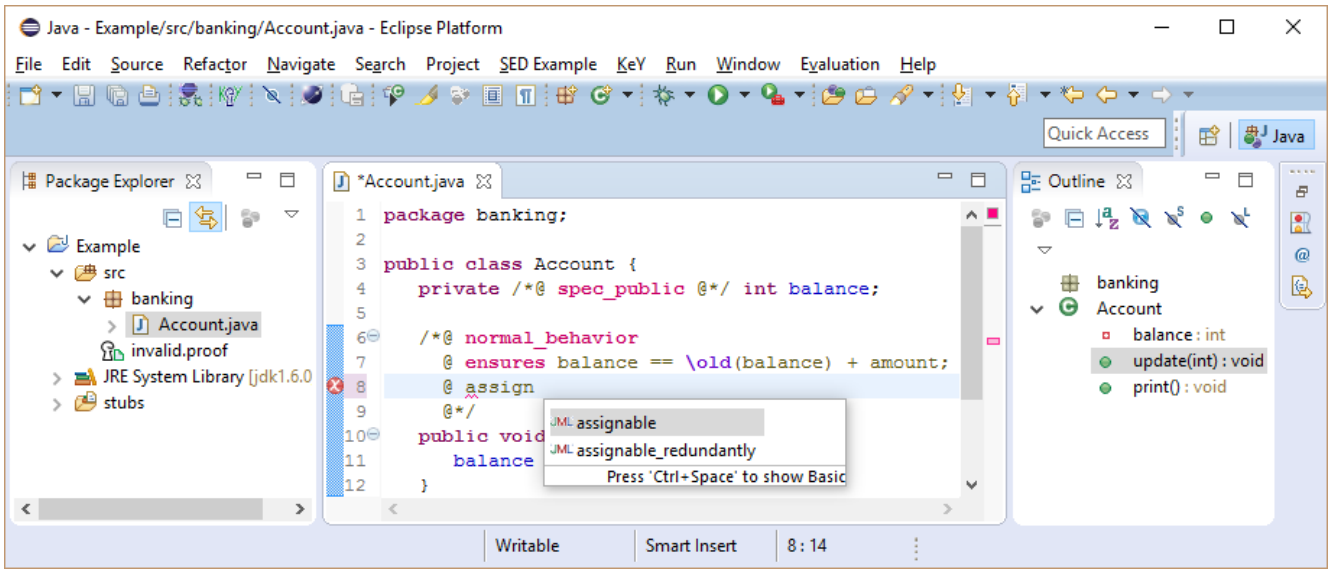


Figure 8.2.: Editing of JML Specifications

8.2 Stubby

KeY operates on the source code level. This means that also the source code of API methods called by the code to be verified needs to be provided. If this is not possible, stubs can be offered instead. A stub is a normal Java file annotated with JML specifications, but without method implementations. The class paths used by KeY are shown in Table 8.2.

Name	Proof Obligations	Method Bodies	Quantity	Description
\javaSource	Yes	Yes	1	The mandatory path to the source code to verify.
\classpath	No	No	*	Optional paths to additional stubs.
\bootclasspath	No	Yes	0..1	Optionally, the path to stubs of basic Java types KeY expects to be available. If not defined, default stubs are used.

Table 8.2.: Class Paths of KeY

Writing stubs is a tedious and error-prone task which is easy to automate. Stubby generates stub files for all API members used by the current source code. Stub generation is triggered by context menu item **Generate Stubs** of a Java project as shown in Figure 8.3.

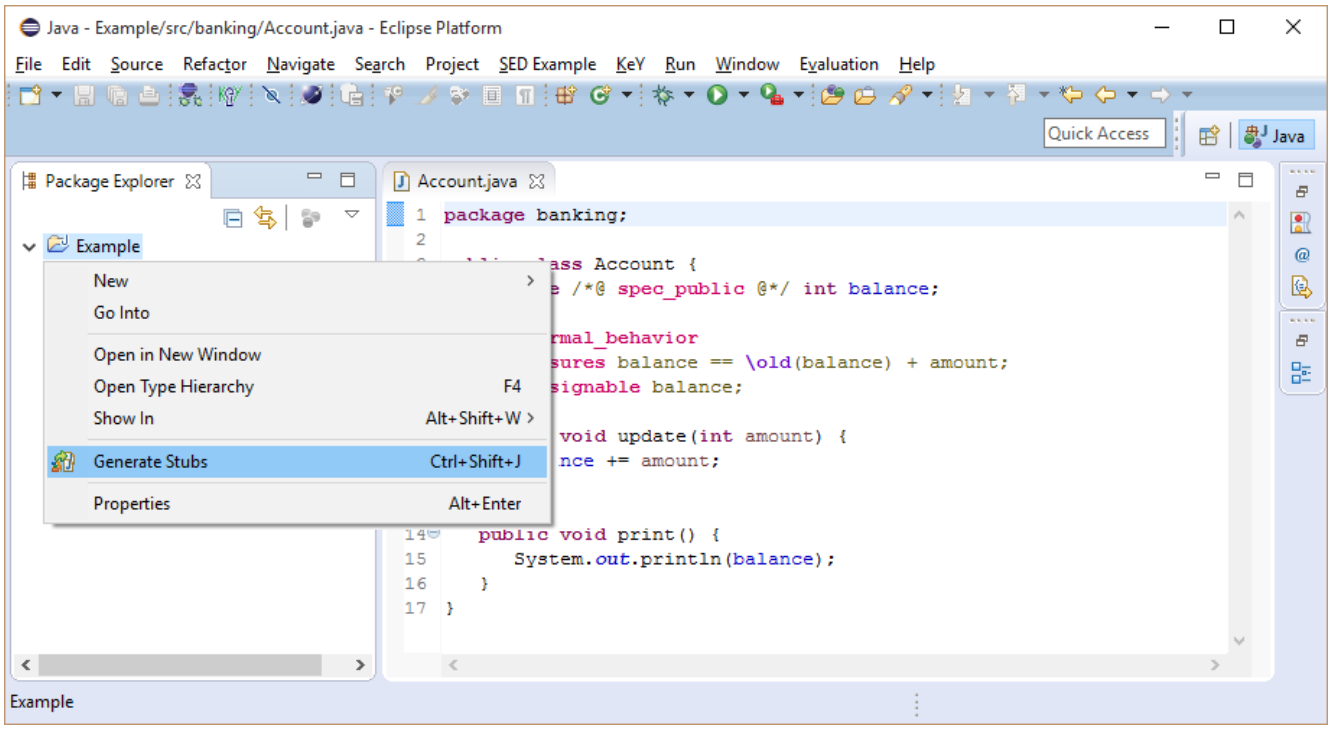


Figure 8.3.: Generation of Stubs for a Java Project

The opened settings dialog (see Figure 8.4) allows one to configure how KeY will use the generated stubs[2]. If used as class path, only stubs for types not part of the boot class path will be generated. If used as boot class path, in addition to the found dependencies, stubs will be generated for all basic API members KeY expects to be available. Compared to the stubs of KeY's default boot class, the generated stubs reflect the type hierarchy of the target Java version. Otherwise, if not used by KeY, also generics will be generated.

The generated stubs used as class paths are shown in the **Package Explorer** view of Figure 8.5. By default, methods are annotated with a weak contract as shown in Listing 8.1. The default contract can be modified by the user as shown in Figure 8.5. The annotation `@generated NOT` ensures that changes are preserved when stub generation is performed the next time.

```
1 /*@ behavior
2   @ requires true;
3   @ ensures true;
4   @ assignable \everything;
5   @*/
```

Listing 8.1: Default Contract of Generated Method Stubs

Stubby uses JET (Java Emitter Templates)[3] to generate stub files. The feature to maintain changes in generated files is one of the strengths of JET.

[2] Only available if both Stubby and KeY 4 Eclipse are installed.
[3] www.eclipse.org/modeling/m2t/?project=jet

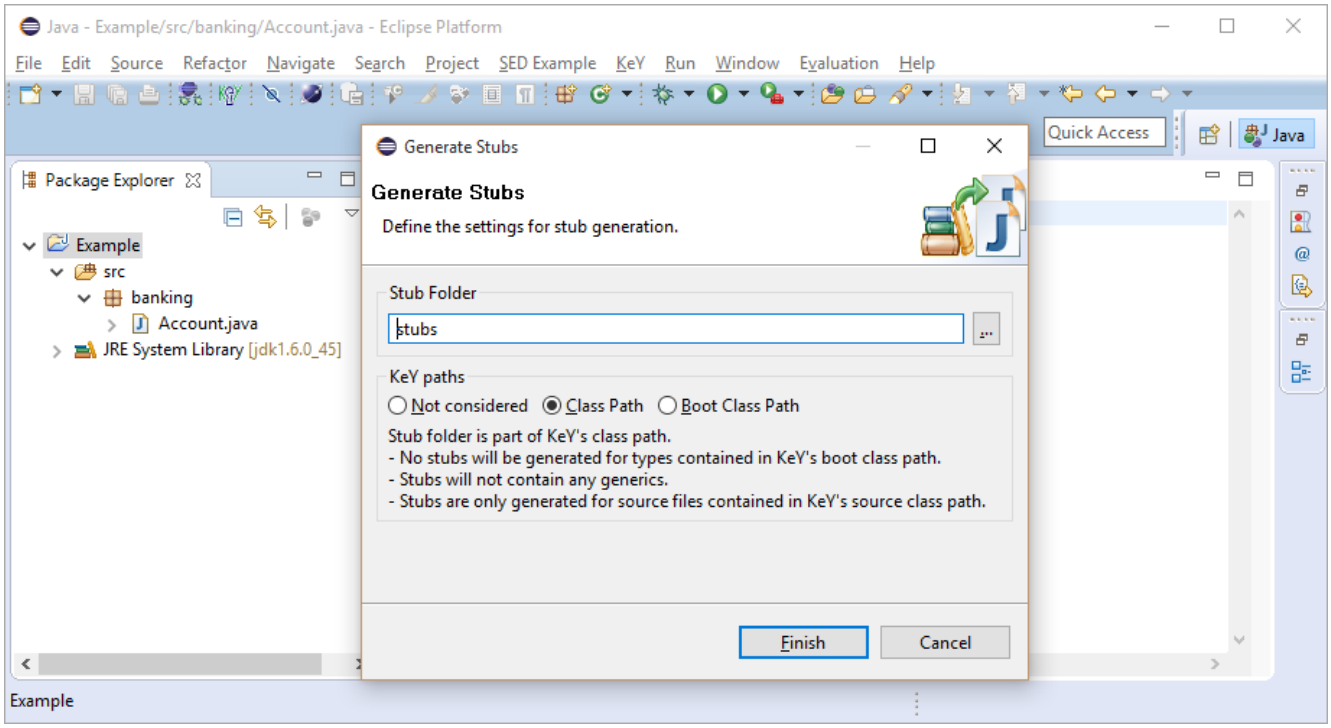


Figure 8.4.: Customization of the Stub Generation

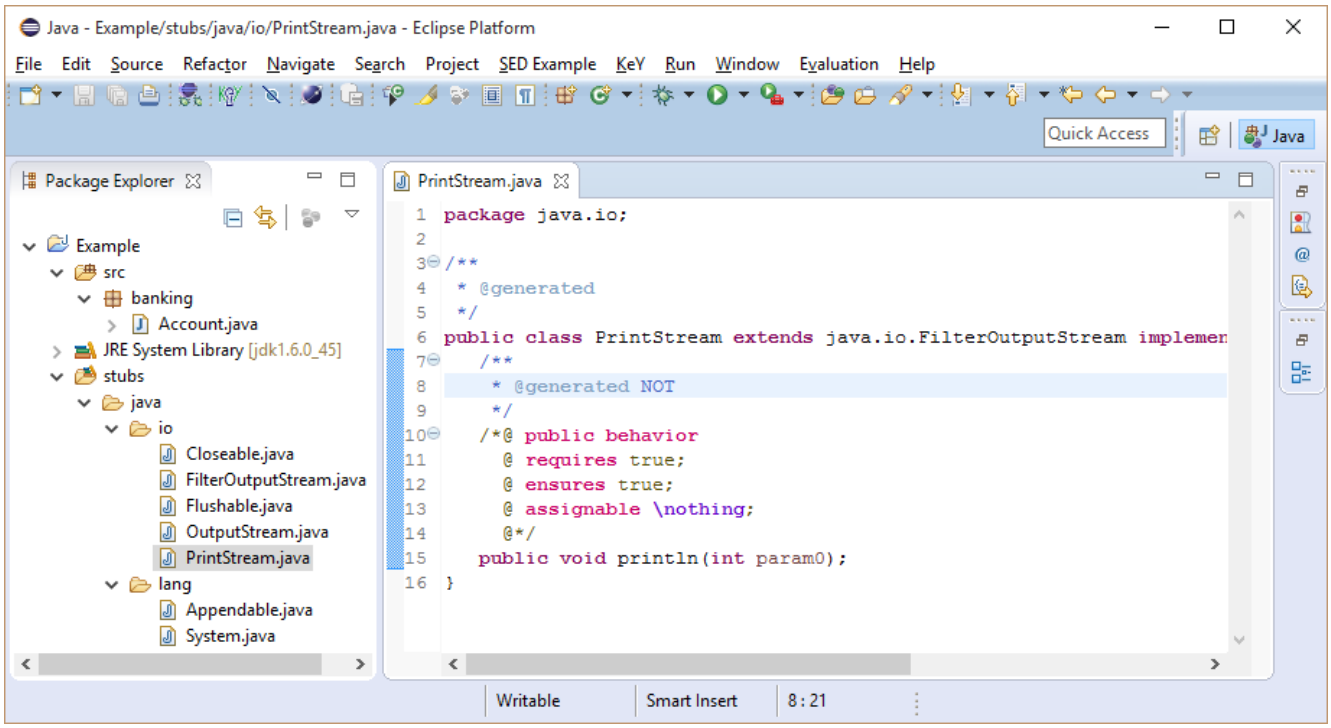


Figure 8.5.: Editing of Generated Stubs

8.3 KeY 4 Eclipse

KeY 4 Eclipse offers the basic infrastructure for verification with KeY. This includes to provide the source code of KeY to Eclipse plug-ins and to offer basic KeY specific user interface controls. KeY 4 Eclipse also extends the user interface of Eclipse. For instance by a preference page to change taclet options and by a Java project properties page to configure KeY's class paths (see Table 8.2).

Another feature of KeY 4 Eclipse is an infrastructure to instantiate proofs in different user interfaces, called *applications*. An application is a user interface for interactive verification with KeY. A proof can be instantiated for instance by context menu item **Start Proof** of a Java method, as shown in Figure 8.6. Other possibilities are to load a Java project or a *.key/*.proof file, or to open an application without a specific context via main menu item **KeY, Open Application**.

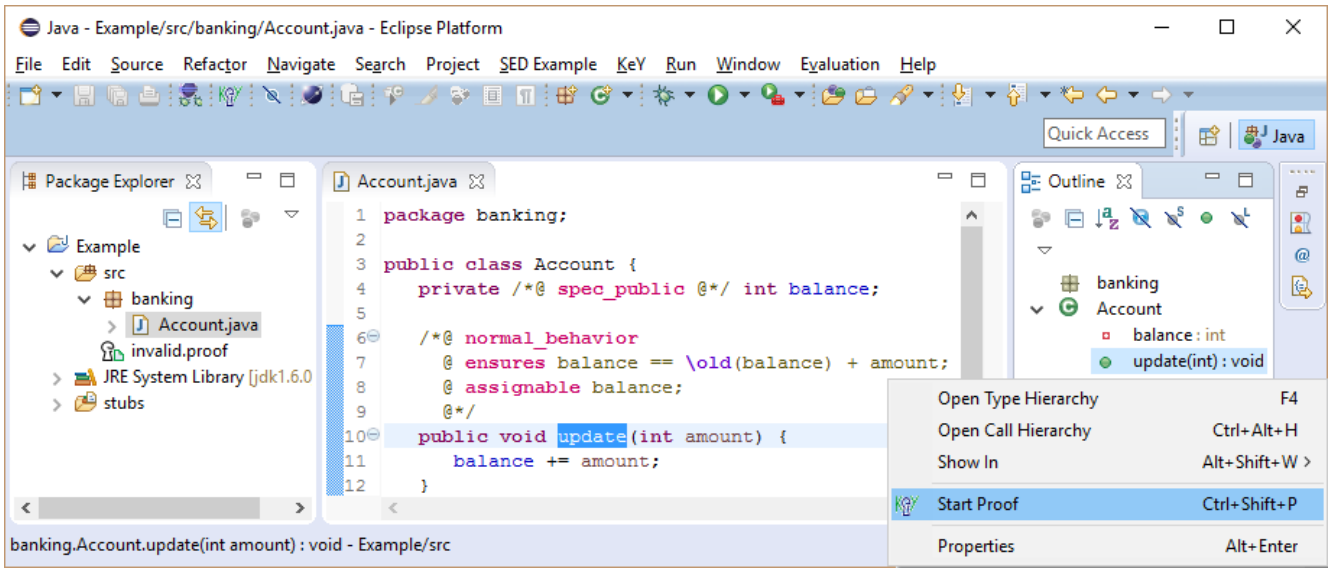


Figure 8.6.: Starting a Proof of a Java Method

If multiple applications are available, the user is asked to select one. In Figure 8.7 the user has the choice between the original user interface of KeY and the KeYIDE (see Section 8.4).

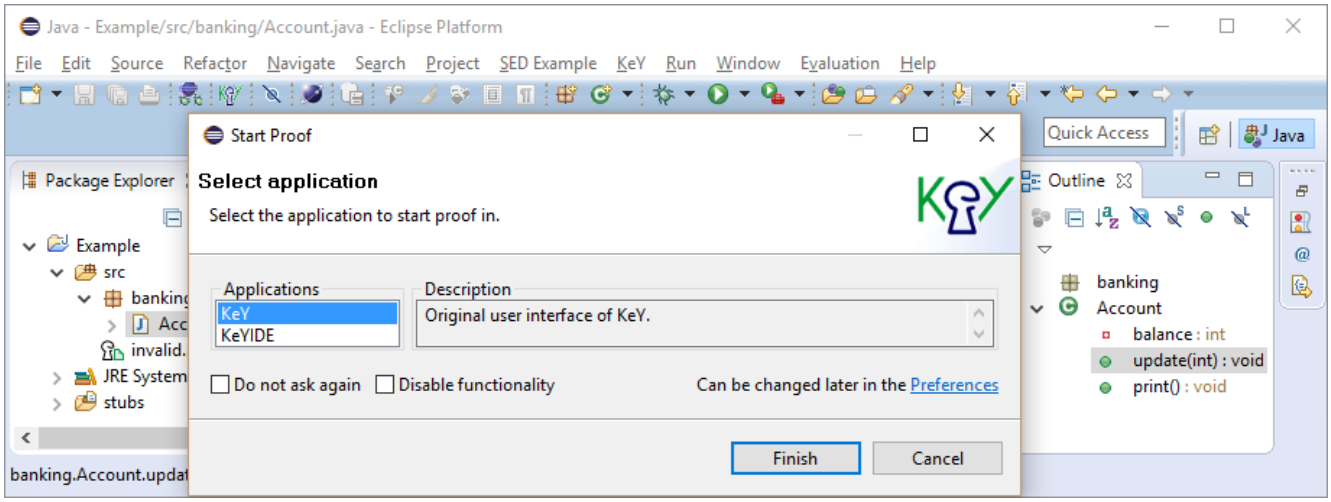


Figure 8.7.: Selection of a Verification Application

The next step depends on the application. In Figure 8.8, the original user interface of KeY is opened which offers to select a proof obligation in the **Proof Management** dialog.

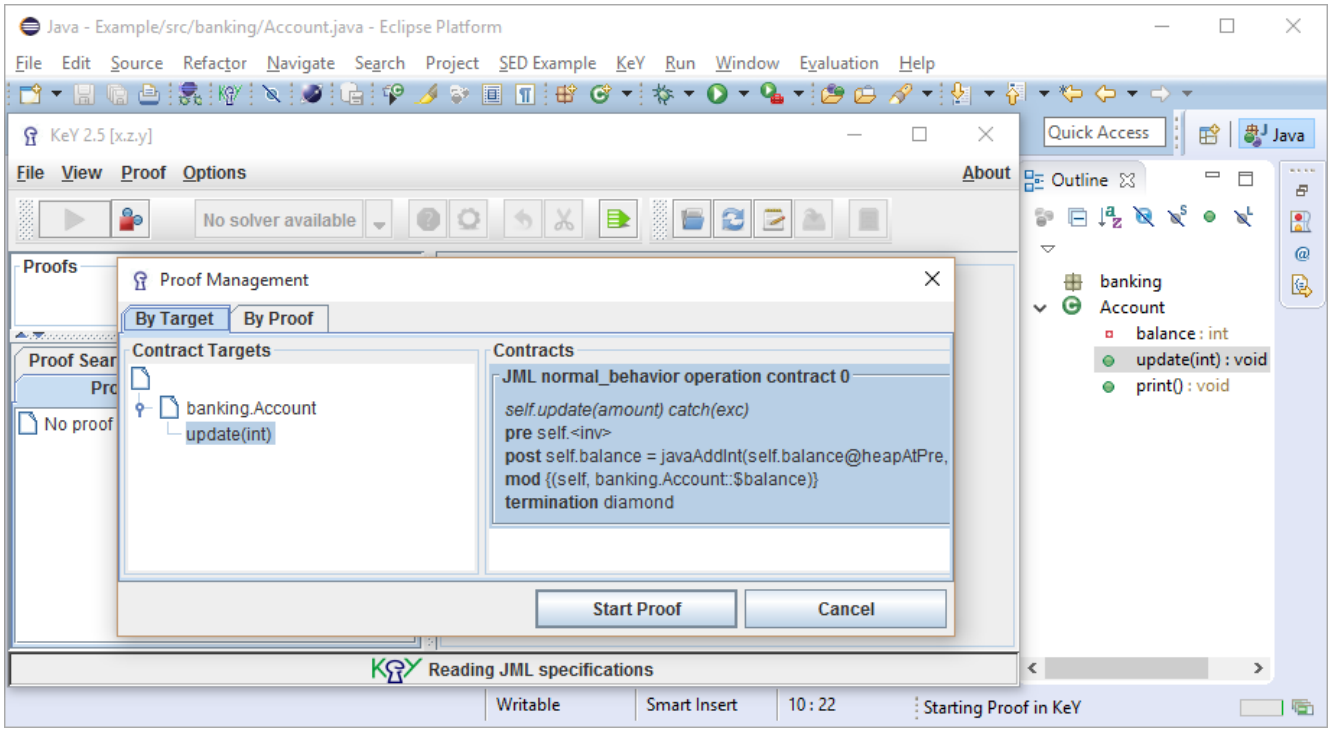


Figure 8.8.: Selection of a Proof Obligation in KeY

8.4 KeYIDE

KeY 4 Eclipse offers the infrastructure for verification with KeY and allows one to open the original user interface of KeY, an external window with a different design and behavior compared to the user interface of Eclipse. This prevents the impression of a seamless integration of KeY into Eclipse.

The goal of the KeYIDE is to implement an alternative user interface for verification with KeY deeply integrated into Eclipse. As Figure 8.9 shows, provides the **KeY** perspective everything needed for verification. The currently selected sequent is shown in the active editor. Interactive rule application is performed as usual by the context menu. The proof tree is shown in view **Outline**. View **Strategy Settings** allows one to change the behavior of the proof search strategy.

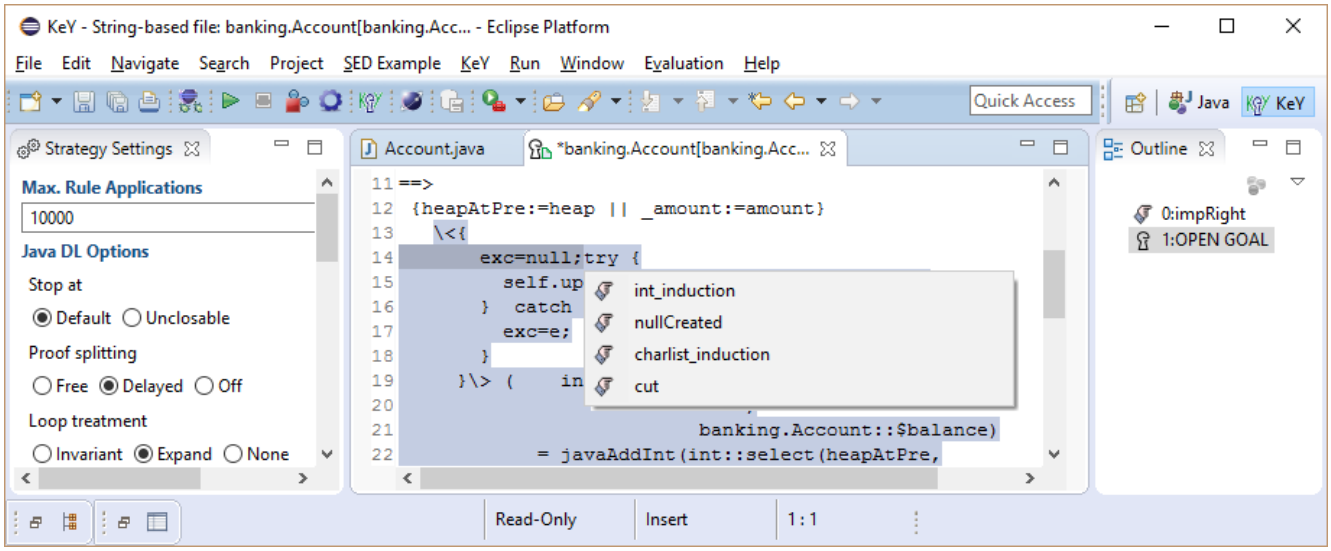


Figure 8.9.: Interactive Verification Directly within Eclipse

9 Evaluation

In this chapter, two experiments evaluating the SED in the context of proof attempt inspection (Section 9.1) and code reviews (Section 9.2) are presented. In addition, the optimizations implemented by KeY Resources to reduce the overall proof time are evaluated in Section 9.3.

9.1 Understanding Proof Attempts Evaluation

The user interface of KeY shows the complete proof tree to the user. Proof tree nodes are labeled with the applied rule and allow the user to inspect the sequent on which the rule was applied. Leaves of the proof tree on which no closing rule is applied are open goals.

The strategy of KeY is in many situations powerful enough to close a proof for a valid formula fully automatically. If the strategy stops the user has to understand the current proof situation to determine the next step. Possible next steps are (i) to change code or specifications in case of a defect or (ii) to apply some rules interactively to help the prover before restarting the strategy.

Understanding the current proof situation is not trivial, because KeY's strategy follows general heuristics which are different to the steps how a user would perform a specific proof. A usability study to analyze the gap between the prover's and the user's model of the proof in context of the KeY system is presented in [23, 20]. An insight of the usability study is that almost all participants tried to abstract the proof tree by using the *Hide Intermediate Proofsteps* feature. This feature hides all intermediate proof nodes with exactly one child. The resulting proof tree consists only of branch conditions and leaf nodes. Branch conditions resulting from symbolic execution describe the taken execution paths whereas other branch conditions describe additional constraints under which the proof obligation is evaluated. Another insight is that users sometimes prune the proof tree at an understood state, like the end of symbolic execution, and then apply rules in a controlled way.

A symbolic execution tree generated by KeY's Symbolic Execution Engine (Chapter 4) abstracts from a proof in a similar way than the *Hide Intermediate Proofsteps* feature. But in addition to the branch conditions, a symbolic execution tree provides all states occurring in the source code. For those states where symbolic execution ends, the truth statuses of the proof obligation to show are traced (Section 5.1).

Whereas KeY shows a proof attempt from the logical perspective with all details, the SED allows one to inspect a proof attempt from the developer's perspective where many proof steps are hidden. This raises the question which user interface is more effective and efficient to understand a proof attempt and thus to bridge the gap between the prover's and the user's model of the proof. To answer the question an experiment is performed which is described in this section.

The planning of the experiment is presented in Section 9.1.1. How the experiment was executed is then presented in Section 9.1.2. Finally, Section 9.1.3 analyzes the collected data before results are discussed in Section 9.1.4.

9.1.1 Experiment Planning

This section presents the planning of the experiment. First, the scope of the experiment is defined. Second, the involved variables are identified and used to define hypotheses. Third, the selected design type is discussed. Fourth, the experiment instrumentation including the chosen proof attempts is presented. Finally, threats to validity are discussed.

Scope

The purpose of the experiment is to evaluate a proof attempt inspection using KeY and the Symbolic Execution Debugger (SED). The experiment is run from the research perspective to find out if there is a significant difference in effectiveness and efficiency between both tools in general. During the evaluation proof attempts and related questions are shown to the participants to measure their performance. Each proof attempt verifies a property of a small example inspired by an interesting problem or a case study. The experiment was announced publicly on the KeY website and KeY mailing list. Participants are thus all KeY users, in particular the KeY developers which are most active in the KeY community. The scope of the experiment is summarized as follows:

Analyze *proof inspection in KeY and SED*
for the purpose of *evaluation*
with respect to *effectiveness and efficiency*
from the point of view of the *researcher*
in the context of *all KeY users*.

Variables Selection

The variables involved in the experiment are shown in Table 9.1. The independent variables determine the cases for which the dependent variables are sampled. A value of an independent variable that changed during the experiment is called treatment. Here, the independent variables are M with treatments KeY and SED, and P with the four proof attempts to inspect. During the experiment, a participant is asked to inspect a proof attempt of P with a tool of M. The controlled variables are used to classify participants according to their experience with Java, JML, KeY and SED in an ordinal scale. The separation between less and more than two years is made to separate beginners from experienced users assuming that this is roughly the time needed to understand Java, JML and KeY well enough for this evaluation. As the SED is rather new, it is assumed that participants do not have a lot of experience with it. For this reason, the separation between beginners and experienced users is set to one year of SED experience.

Efficiency is measured by the time spend to answer the questions using a treatment. Effectiveness is measured in the number of correctly answered questions and the confidence in the given answers. Questions are single and multiple choice questions to ensure an automatic analysis. For each question, a number of correct and wrong answers are shown to the participant and she is asked to select some of them. An answer of a multiple choice question is considered to be correct, if all and only correct answers are selected.

To give credits also to partially correct answers, a correctness score is used. The correctness score $QS_{tm} = \sum_{q \in tm} qs(q)$ is the sum of the scores over all questions of the treatment tm. The question score qs of a single question q is defined as follows:

$$qs(q) = \begin{cases} \frac{\#correctSelectedAnsw(q) - \#wrongSelectedAnsw(q)}{\#correctSelectedAnswers} & \text{if } \#correctSelectedAnsw(q) > \#wrongSelectedAnsw(q) \\ \frac{\#correctSelectedAnsw(q) - \#wrongSelectedAnsw(q)}{\#wrongSelectedAnswers} & \text{if } \#correctSelectedAnsw(q) \leq \#wrongSelectedAnsw(q) \end{cases}$$

Intuitively, the question score of question q is the difference between the number of the correct answers of q selected by the participant and the number of the wrong answers of q selected by the participant. But each question has a different number of wrong and correct answers. To achieve comparability between questions, the difference between correct and wrong answers is divided by the total number of correct or wrong answers depending on the correctness.

The confidence in the correctness of the given answer is asked for each question. Available confidence levels are *sure (My answer is correct!)*, *educated guess (As far as I understood the content, my answer should be correct.)* and *unsure (I tried my best, but I don't believe that my answer is correct.)*. For each question q

Type	Name	Values	Description
Independent Variable	M	{KeY, SED }	The compared tools.
	P	{`Calendar`, `Account`, `ArrayUtil`, `MyInteger`}	The inspected proof attempts and related questions. Each proof attempt verifies a method contract provided by one of the listed classes.
Controlled Variable	E_{Java}	$\{none, < 2\ years, \geq 2\ years\}$	The experience with Java.
	E_{JML}	$\{none, < 2\ years, \geq 2\ years\}$	The experience with JML.
	E_{KeY}	$\{none, < 2\ years, \geq 2\ years\}$	The experience with KeY.
	E_{SED}	$\{none, < 1\ years, \geq 1\ years\}$	The experience with SED.
Dependent Variable	Q_{tm}	Integer	The number of correctly answered questions per treatment tm of M.
	QS_{tm}	Real	The achieved correctness score per treatment tm of M.
	C_{tm}	Integer	The achieved confidence score per treatment tm of M based on Q.
	CS_{tm}	Real	The achieved confidence score per treatment tm of M based on QS.
	T_{tm}	Integer	The time needed to answer questions of a treatment tm of M in seconds.

Table 9.1.: Variables

a confidence rating $c(q)$ is computed according to Table 9.2. If the participant is sure that the answer is correct and it is correct, she gets the maximal points. Otherwise, if the answer is wrong and she is sure that it is correct, she gets the worst score. If the answer is based on an educated guess, which is weaker compared to sureness, she gets less or loses less points accordingly. If the participant is unsure and thinks her answer is wrong, and it is wrong, then she gets a score point because her intuition about the correctness is right. Otherwise, if she thinks her answer is wrong but it is right, she loses a score point for the same reason. The confidence score $C_{tm} = \sum_{q \in tm} c(q)$ is finally the sum of the confidence rating over all questions answered using the treatment tm.

	Correct Answer of q	Wrong Answer of q
Sure	2	-2
Educated Guess	1	-1
Unsure	-1	1

Table 9.2.: Confidence Rating $c(q)$ of a Single Question q

The confidence score $CS_{tm} = \sum_{q \in tm} cs(q) \cdot qs(q)$ takes also partially correct answers into account. For each question of treatment tm is the confidence score (see Table 9.3) multiplied with the achieved question score. The results are then accumulated to compute the confidence score.

	$qs(q) > 0$	$qs(q) \leq 0$
Sure	2	-2
Educated Guess	1	-1
Unsure	-1	1

Table 9.3.: Confidence Rating $cs(q)$ of a Single Question q

Hypothesis Formulation

Proof inspection using SED is considered to be more effective and more efficient as an inspection using KeY. To empirically test whether this claim expressed as alternative hypotheses is valid, the null hypotheses need to be rejected, see Table 9.4.

Kind	Name	Hypothesis	
Null hypothesis	H_{0_Q}	$\mu_{Q_{SED}} = \mu_{Q_{KeY}}$	with $\mu_{Q_{Treatment}} = \frac{Q_{Treatment}}{\#questionsOfTreatment} \in \{x \in \mathbb{Q} \mid 0 \leq x \leq 1\}$
	$H_{0_{QS}}$	$\mu_{QS_{SED}} = \mu_{QS_{KeY}}$	with $\mu_{QS_{Treatment}} = \frac{QS_{Treatment}}{\#questionsOfTreatment} \in \{x \in \mathbb{Q} \mid 0 \leq x \leq 1\}$
	H_{0_C}	$\mu_{C_{SED}} = \mu_{C_{KeY}}$	with $\mu_{C_{Treatment}} = \frac{C_{Treatment}}{\#questionsOfTreatment} \in \{x \in \mathbb{Q} \mid -2 \leq x \leq 2\}$
	$H_{0_{CS}}$	$\mu_{CS_{SED}} = \mu_{CS_{KeY}}$	with $\mu_{CS_{Treatment}} = \frac{CS_{Treatment}}{\#questionsOfTreatment} \in \{x \in \mathbb{Q} \mid -2 \leq x \leq 2\}$
	H_{0_T}	$\mu_{T_{SED}} = \mu_{T_{KeY}}$	with $\mu_{T_{Treatment}} = \frac{T_{Treatment}}{timeOfAllTreatments} \in \{x \in \mathbb{Q} \mid 0 \leq x \leq 1\}$
Alternative hypothesis	H_{1_Q}	$\mu_{Q_{SED}} > \mu_{Q_{KeY}}$	
	$H_{1_{QS}}$	$\mu_{QS_{SED}} > \mu_{QS_{KeY}}$	
	H_{1_C}	$\mu_{C_{SED}} > \mu_{C_{KeY}}$	
	$H_{1_{CS}}$	$\mu_{CS_{SED}} > \mu_{CS_{KeY}}$	
	H_{1_T}	$\mu_{T_{SED}} < \mu_{T_{KeY}}$	

Table 9.4.: Hypotheses

Choice of Design Type

An important design decision of the experiment is that a participant should benefit from her participation. To achieve this, each participant uses both tools resulting in a paired comparison design. In case the participant is unfamiliar with KeY or the SED, this allows her to try out the tools and to obtain her own judgment about when to use which tool. The final design type is a paired comparison design as shown in Table 9.5.

	Proof Attempt 1	Proof Attempt 2	Proof Attempt 3	Proof Attempt 4
SED	Subject_n	Subject_n	Subject_{n+1}	Subject_{n+1}
KeY	Subject_{n+1}	Subject_{n+1}	Subject_n	Subject_n

Table 9.5.: Paired Comparison Design

The general design principles randomization, blocking and balancing are applied as well. The order of proof attempts is random. The first two proof attempts are always inspected by the same tool and the next two proof attempts are then inspected by the other tool. The decision about the first tool is random as well. This avoids multiple switches between tools which could confuse the participant. Also a

participant not familiar with a tool will gain experience towards the later proof attempts. The server used to collect evaluation results guarantees that all possible permutations of proof orders will be evaluated equally often. The server also ensures that each permutation is evaluated with both tools first.

The performance of the participants may depend on their experience with KeY and SED. As the SED is relatively new and thus most likely unknown to most participants, the experience with KeY is used for blocking. Considering only completed evaluations, balancing is automatically achieved by the chosen design, because each participant uses both tools and inspects all four proof attempts. Thus, the number of participants is the same for each treatment.

Instrumentation

The evaluation requires no knowledge about JML, KeY or the SED, only Java basics are required. Consequently, the evaluation has to be self explaining which is ensured by additional instructions.

During the evaluation, a participant inspects proof attempts with KeY and the SED. As both tools are available within Eclipse, the evaluation itself is implemented as an Eclipse wizard. The wizard is opened in an additional window so that Eclipse itself remains fully functional (see Figure 9.1).

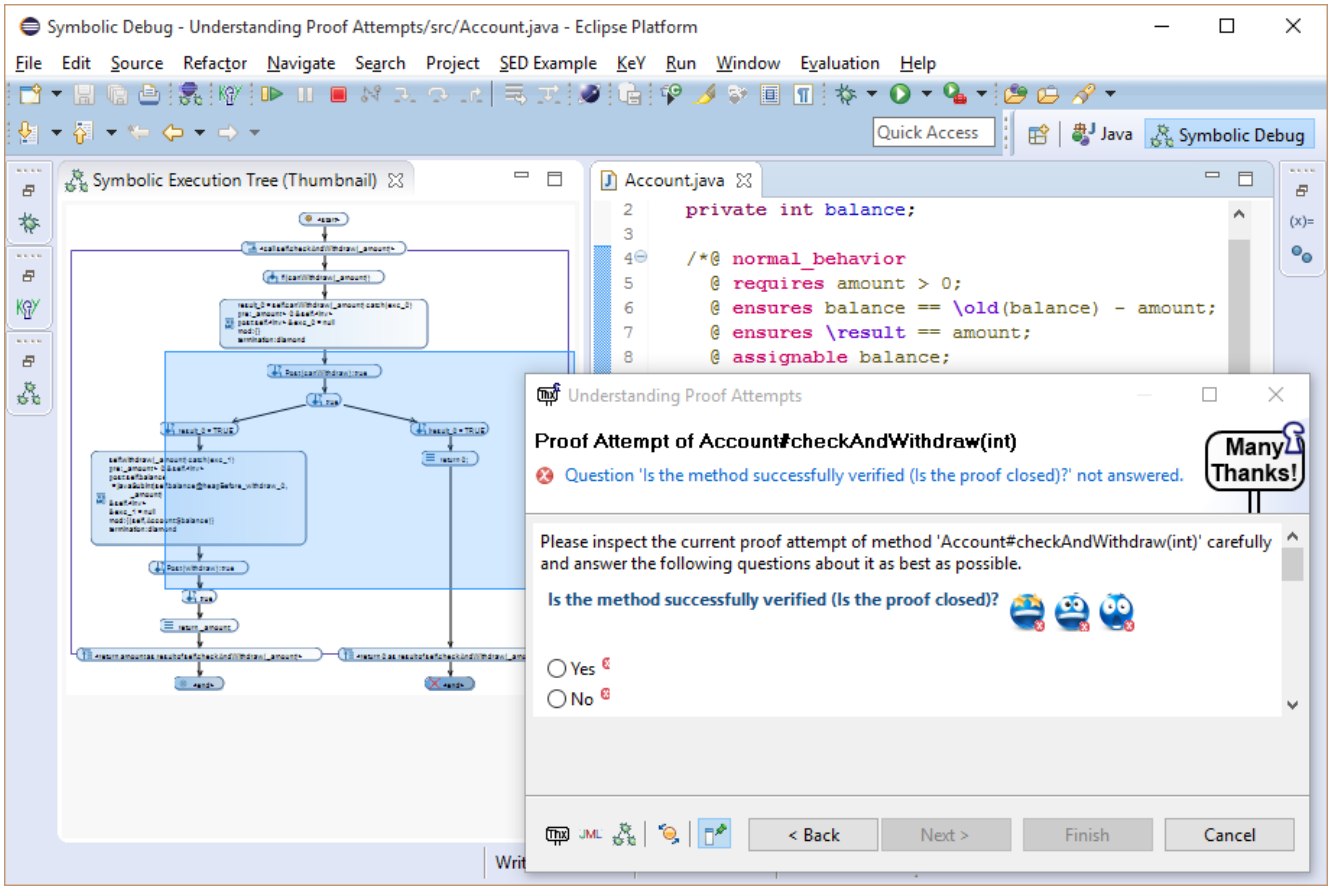


Figure 9.1.: Screenshot of the Understanding Proof Attempts Evaluation Wizard

The evaluation wizard is separated into two phases in which information is collected and sent to the server. The first phase collects the background knowledge of the participant and receives the order of proof attempts and the tool assignment, i.e. which proof attempt has to be inspected with which tool. The evaluation itself is then performed in the second phase. If the participant cancels the evaluation in the second phase, she is asked to send intermediate results to the server. When she opens the evaluation wizard the next time, she is asked to recover the previous state to continue the already started evaluation. The following enumeration shows the steps in detail:

1. Initialization Phase
   a) *Terms of Use*: A text explaining the terms of use is shown to the participant. The participant needs to accept them before she can continue.
   b) *Background Knowledge*: The background knowledge of Java, JML, KeY and SED is collected.
   c) *Sending Data*: The participant accepts sending the background knowledge to the server. The evaluation wizard receives as answer the order of proof attempts and if the SED should be used first for the evaluation phase.
2. Evaluation Phase
   a) *Evaluation Instructions*: A video shows how to answer questions and how to use the wizard.
   b) *JML*: A textual documentation introducing the used features of JML.
   c) *SED or KeY Instructions*: A video explaining needed features and best practices to review a proof attempt using SED or KeY.
   d) *Proof Attempt 1*: The first proof attempt and related questions.
   e) *Proof Attempt 2*: The second proof attempt and related questions.
   f) *The complementary SED or KeY Instructions*: The remaining instruction video.
   g) *Proof Attempt 3*: The third proof attempt and related questions.
   h) *Proof Attempt 4*: The fourth proof attempt and related questions.
   i) *Feedback about Tools and Evaluation*: The participant is asked to rate the helpfulness of the SED and KeY features mentioned in the instruction videos. The SED features are (i) the visualized symbolic execution tree (Section 6.2), (ii) the highlighted statements reached during symbolic execution (Section 6.2), (iii) the **Variables** view (Section 6.2), (iv) the visualization of memory layouts (Section 6.2) and (v) the truth status tracing (Section 6.7). The KeY features are (i) the shown proof tree, (ii) the tab goals, (iii) the shown sequent, (iv) the hiding of intermediate proof steps and (v) the listing of applied method contracts in the proof management dialog.
   j) *Sending Data*: The participant accepts sending the evaluation results to the server.
   k) *Completed Message*: A thank you for participation image is shown. The participant can now finish the wizard.

The proof attempt `Account` verifies method `checkAndWithdraw` of Listing A.1. It is a simplified version of an example used in a graduate course[1] on software verification with KeY taught at Chalmers. The reason why the proof is still open is that during symbolic execution the method call of `canWithdraw` is treated by applying its method contract which says nothing about the result value. Consequently, both execution paths of `checkAndWithdraw` are feasible, but the one returning `0` as result does not fulfill the method contract.

Proof attempt `Calendar`, see Listing A.2, is a simplified version of the FMCO 2006 Submission: 'Verifying Object-Oriented Programs with KeY: A Tutorial' [3][2]. The tricky part is that the class invariant makes the *then-branch* of the **`if`**-statement infeasible. The class invariant is also not preserved and an `ArrayStoreException` can be thrown.

The next proof attempt `ArrayUtil`, see Listing A.3, is a simplified version of a lab exercise from an undergraduate course[3] taught at TU Darmstadt. The proof remains open for two reasons: First, the loop invariant is not preserved, because 1 instead of `i` is assigned to `minIndex`. Second, one execution path returns the value at the found index instead of the found index.

[1] The current SEFM course is available at: www.cse.chalmers.se/edu/course/TDA293
[2] www.key-project.org/fmco06
[3] The course 'Formale Grundlagen der Informatik 3' (FGdI3) is available at: www.se.tu-darmstadt.de/teaching/courses/previous-periods/formale-grundlagen-der-informatik-iii

The last proof attempt `MyInteger`, see Listing A.4, is the running example of the talk 'JML Editing in Eclipse and KeY-IDE' presented at the workshop 'JML: Advancing Specification Language Methodologies'[4]. The relevant aspect is that the postcondition does not hold in case that **`this`** and `s` are aliased.

The questions and available answers for each proof attempt are always generated following the same schema. On top level the participant is asked if the proof is closed and which statements are executed by the proof. The `Account` proof attempt additionally asks for applied method contracts. In case the participant selects that the proof is still open, she has to select all reasons why the proof is still open. Each JML construct is offered as a possible reason. Additional options are that the proof can be closed interactively or something else (free text). In case the participant selects a JML construct, she is finally asked at which execution paths this happens. Possible execution paths are marked with `//XXX:` comments in the source code. If **`normal_behavior`** is violated due to a thrown exception, instead of selecting an execution path the participant is asked which exception is thrown (free text is possible as backup). All questions offer also the opportunity to give up after ten minutes.

In case a participant selected the *give up* answer, it is treated in the check if a question is correctly answered and in the question score *qs* computation as if the answer does not exist at all. This means that it is neither counted as wrong nor as a correct answer. This ensures the same result in case the *give up* answer is selected additionally to a partial or fully correct answer. The answer of a participant who only selected the *give up* answer in order to continue the evaluation is still treated as wrong and achieves no question score points.

Validity Evaluation

"*Conclusion validity* concerns the statistical analysis of results and the composition of subjects." [135, page 185] The hypotheses of this experiment are tested with well known statistical techniques. Threats to conclusion validity are the low number of samples and the validity of the quality of answers. Subjects may fake answers to compromise the experiment. However, several participants are known people and in addition, some were monitored during the evaluation. The motivation of subjects to compromise the experiment is considered to be low because of the mentioned reasons.

"*Internal validity* concerns matters that may affect the independent variable with respect to causality, without the researchers knowledge." [135, page 185] Maturation is a threat to internal validity in this experiment. Inspecting a proof is a time intensive task and the estimated participation time is 60 minutes. Participants may get tired or bored during the experiment. As each tool is applied to two proof attempts, participants may learn how to use it which is desired. The learning between both tools and the order of proof attempts is considered as not critical as randomization is applied. Other threats are considered to be uncritical.

The subjects may have experience with KeY, but most likely not with SED as it is relatively new. This is not critical as the instrumentation introduces the relevant functionality of both tools. There might be a threat that the subjects are not willing to address with SED. However, SED is designed to help verification with KeY. The bias about the experience with the tools is against the hypothesis. There is a risk that subjects lack motivation and thus answer the questions not seriously. However, participation is voluntary and can be done at any time.

"*Construct validity* concerns generalisation of the experiment result to concept or theory behind the experiment." [135, page 185] A threat to construct validity is that the chosen proof attempts might be not representative in general. To mitigate this, the proof attempts are taken from case studies and teaching most likely representative for the use of KeY. The proof attempts also cover important JML features like method contracts, invariants or loop specifications. Other threats to construct validity are considered to be uncritical. Only the motivation of the experiment, to compare the inspection of proof attempts using KeY and the SED, is given to the participants. Therefore, subjects might guess the expected outcome of

[4] www.lorentzcenter.nl/lc/web/2015/677/info.php3?wsid=677&venue=Snellius

the experiment because the SED is the only new tool. However, the exact hypotheses and the related measurements are unknown. In addition, subjects do not have any advantage or disadvantage about the outcome of this experiment.

"*External validity* concerns generalisation of the experiment result to other environments than the one in which the study is conducted." [135, page 185] A threat to external validity is that the source code related to proof attempts is kept to a minimum, in many cases only one method. This is required to reduce the time participants need to understand the verified source code and its specifications. Real Java code is much more complex. Verification on the other hand is modular and only a little part of the source code is considered by a proof. Subjects are selected randomly and their experience varies from none to experts. Consequently, the selection of subjects is not a threat to external validity.

To conclude, there are threats to the validity of the experiment. Hence, conclusions drawn from the results of the experiment are valid within the limitations of the threats.

9.1.2 Execution

The experiment started in June 2015 with the staff of the Software Engineering group at TU Darmstadt. This includes students, PhD students and postdocs. Each of these participants were monitored during the evaluation to improve the instructions and answers. Questions and answers remained stable after the first participant. The results of the first participant are excluded for this reason. Initially, all instructions were texts, but the participants lacked in motivation to read them. All participants mentioned that they would prefer to watch videos instead, which was realized after a view runs. The content of the video is the same as the initial textual descriptions, the results of the previous participants are thus still valid.

The evaluation was then announced in the course 'Formal Specification and Verification of Object-Oriented Software'[5] at TU Darmstadt with about 40 students. Participation was not enforced by the course to not compromise the behavior of the students. The result was disappointing. Only one student volunteered and then canceled the evaluation.

The evaluation became public in July 2015 and was announced on the KeY website, KeY mailinglist and during the 14th KeY Symposium. The evaluation is deployed as a preconfigured Eclipse product. Installation instructions and download links are available on the KeY website[6]. The main steps are to download the Eclipse product, to unpack it, to start it, to perform the evaluation and then to delete it. An installation is not required, so the subject's system remains untouched.

Until mid of November 2015 (and unchanged until end of 2015), 32 participants started the evaluation, but only 21 completed it. Twelve of the participants were monitored during the evaluation.

The background knowledge of the participants is shown in Figure 9.2. All participants had experience with Java and only four participants did not know JML. The knowledge about KeY is fairly distributed between the three classes. Surprisingly, six participants had some experience with the SED.

The relation between the KeY and the SED experience is shown in Table 9.6. It shows that for each class of KeY experience (None, $<$ 2 year, and $\geq$ 2 year), at least some participants have also experience with the SED.

		SED		
		None	$<$ 1 year	$\geq$ 1 year
KeY	None	4	2	0
	$<$ 2 years	7	1	0
	$\geq$ 2 years	4	1	2

Table 9.6.: KeY vs SED Experience of Participants

[5] www.se.tu-darmstadt.de/teaching/courses/lecture-formal-specification-and-verification-of-object-oriented-software

[6] key-project.org/eclipse/SED/UnderstandingProofAttempts.html

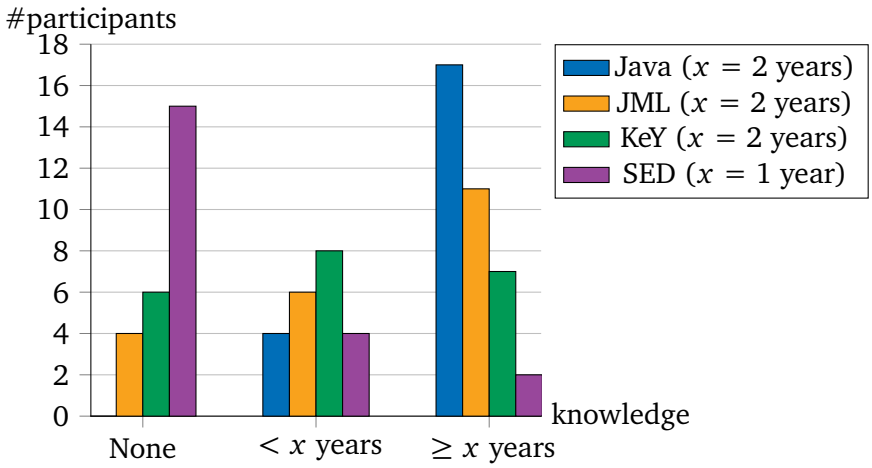
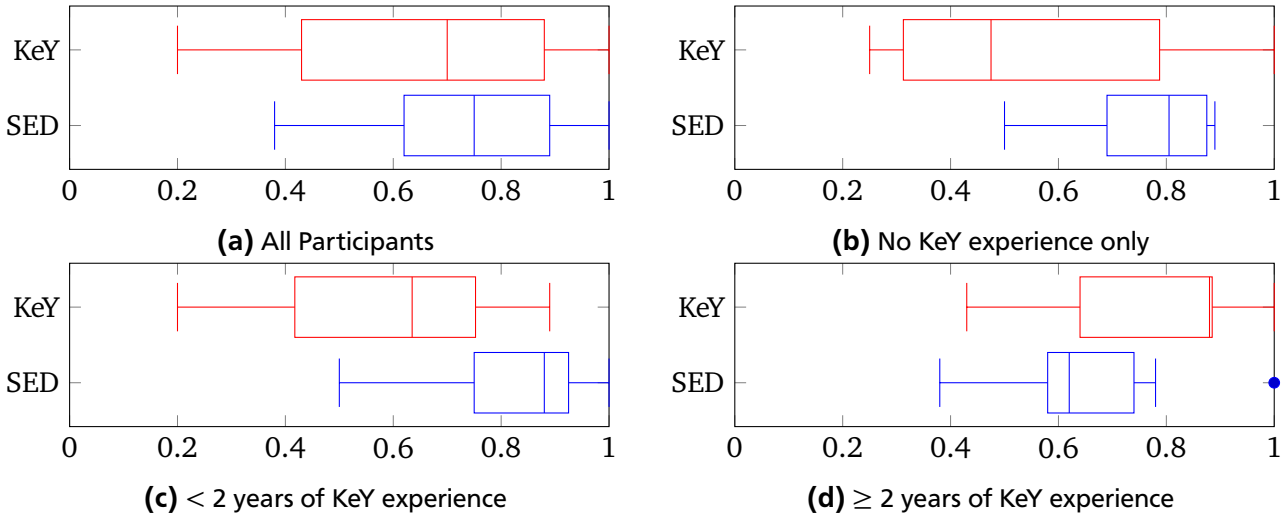


Figure 9.2.: Knowledge of Participants

9.1.3 Analysis

Each of the boxplots in Figures 9.3 to 9.7 show the measured data of a dependent variable. The middle of each box in a boxplot indicates the 50% percentile (the median) of the data. The left border represents the lower quartile lq which is the 25% percentile of the data. The right border represents the upper quartile uq which is the 75% percentile of the data. The left and right whisker indicate the theoretical bounds of the data if it is normally distributed. Data points outside of the whiskers are outliers. The left whisker is defined as $lq - 1.5\,(uq - lq)$ and the right whisker as $uq + 1.5\,(uq - lq)$. Additionally, whiskers are truncated to the nearest existing value within the bounds to avoid meaningless values. The constant 1.5 is chosen according to Frigge et al. [53].

The boxplots in Figure 9.3 show the measured number of correctly answered questions. The maximal value 1 is achieved if all questions are correctly answered. The opposite 0 accordingly if not a single question is correctly answered. The boxplots in Figure 9.3a are based on the results of all participants, whereas only a class of KeY experience is considered in Figures 9.3b to 9.3d. Except for the participants with ≥ 2 years of KeY experience, the total correctness of the answers is better using SED. Only the group with ≥ 2 years of KeY experience achieved better results using KeY. However, this class has an outlier who answered everything correct using the SED.



(a) All Participants

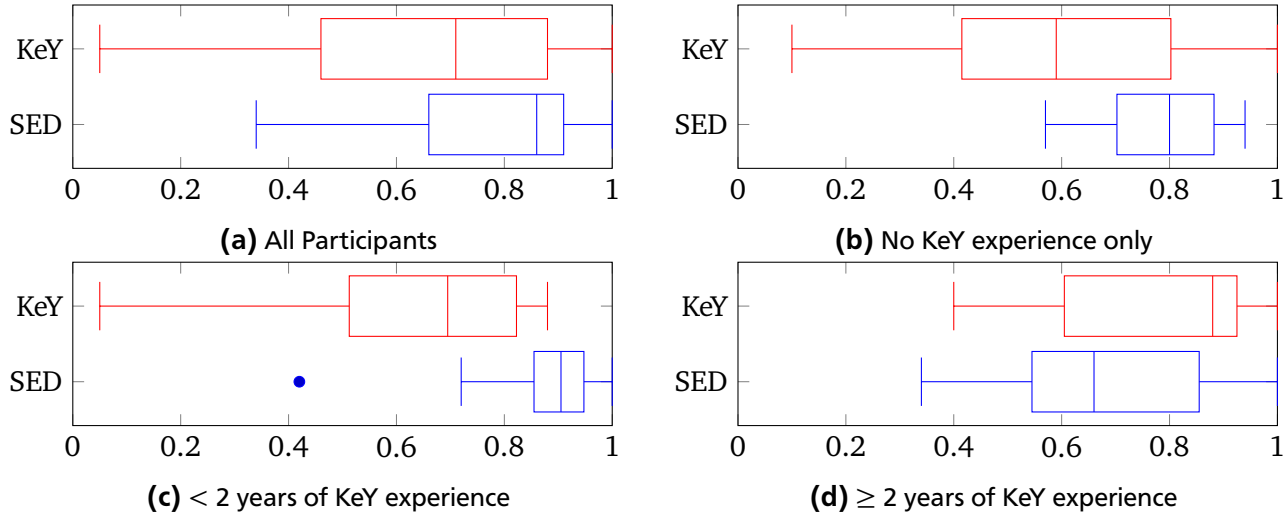
(b) No KeY experience only

(c) $<$ 2 years of KeY experience

(d) $\geq$ 2 years of KeY experience

Figure 9.3.: Correct Answers

Analogous to the number of correctly answered questions, Figure 9.4 shows the measured correctness score taking also partially correct answers into account. The correctness score is better using SED except for the class of KeY users with ≥ 2 years of experience. In this class the correctness using SED is better compared to the number of correctly answered questions, but still behind the achieved one using KeY.

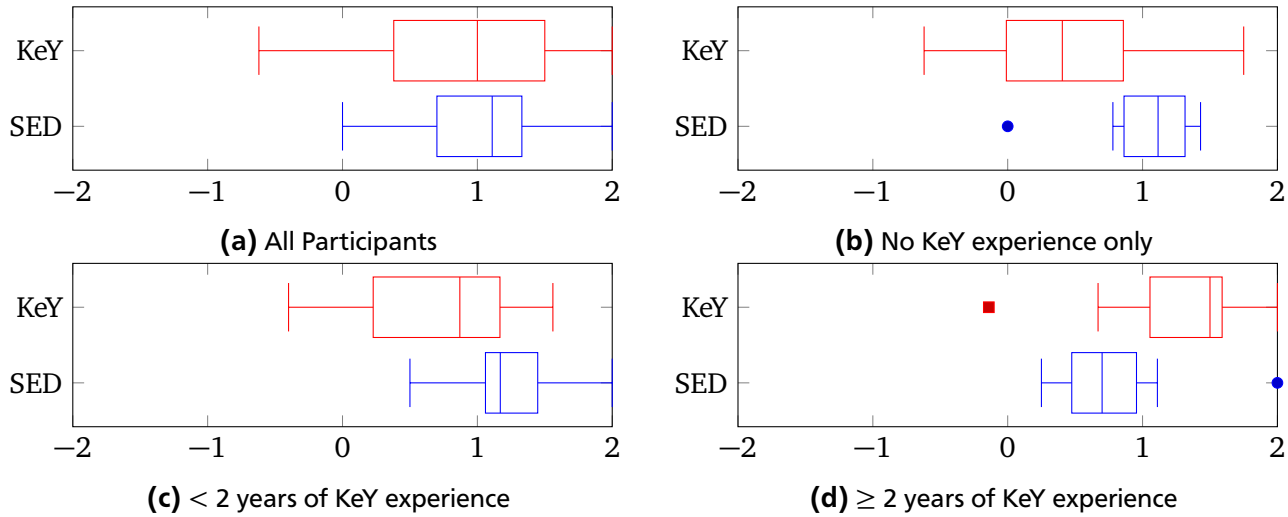


(a) All Participants **(b)** No KeY experience only

(c) $<$ 2 years of KeY experience **(d)** $\geq$ 2 years of KeY experience

Figure 9.4.: Correctness Score

The achieved confidence behaves similarly as the correctness. As Figure 9.5 and Figure 9.6 show, the achieved confidence is better with SED except for the class of KeY users with ≥ 2 years of experience. The difference in that class is less if the confidence takes also partially correct answers into account. A value of 2 is achieved, if a candidate is sure that all answers are right and if they are indeed right. -2 is the worst score achieved if the confidence in the given answer was never right.



(a) All Participants **(b)** No KeY experience only

(c) $<$ 2 years of KeY experience **(d)** $\geq$ 2 years of KeY experience

Figure 9.5.: Confidence Score

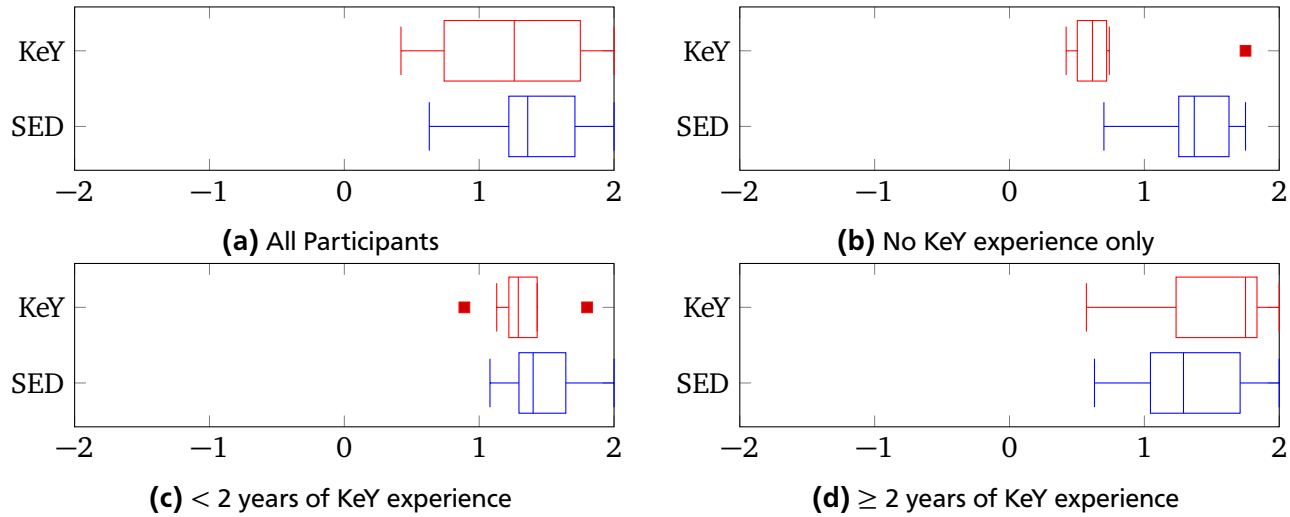


Figure 9.6.: Confidence Score of Partially Correct Answers

The measured time[7] is shown in Figure 9.7. A value of 1 means that a participant spent 100% of the time using one tool. The opposite is 0 when a participant spent 0% of the time using a tool. The participants with KeY experience spent less time when using KeY than when using the SED. For participants without KeY experience it is the other way round.

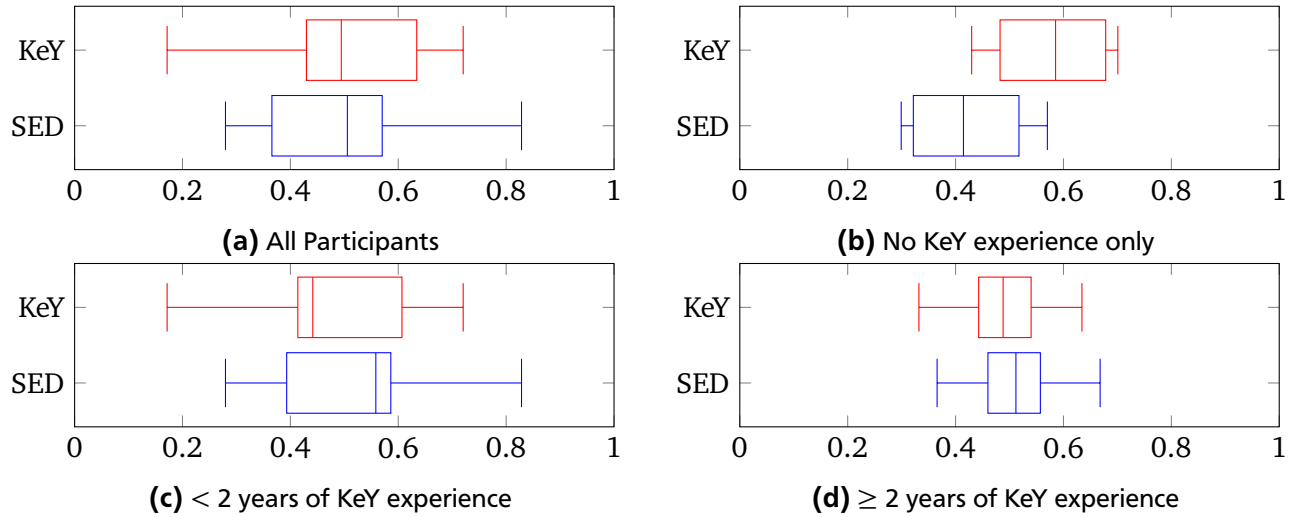


Figure 9.7.: Time

The null hypotheses of Table 9.4 can be rejected according to Wohlin et al. [135] using a one sided *Paired t-Test*, *Wilcoxon Signed Rank Test* or a *Sign Test*. The *Paired t-Test* requires that the data is normally distributed which is tested according to Wohlin et al. [135] with a *Chi-2 Goodness of Fit Test*. The *Sign Test* was performed by my own implementation following Wohlin et al. [135], for all others tests the *Apache Commons Mathematics Library* was used.

The results of the tests are shown in Tables 9.7 to 9.9. The tests are only performed using the results of all participants and not with respect to the separate classes, because the number of participants in each class of KeY experience on each own is not enough to reject a hypothesis. The significance level is set to 0.05 meaning that there is a 5% chance at which a hypothesis is wrongly rejected.

All tests reject the correctness related hypotheses H_{0_Q} and $H_{0_{QS}}$ at the given significance level. The hypotheses $H_{0_{CS}}$ for the confidence taking partial correct answers into account can be rejected at signif-

[7] The times measured for four of the participants were invalid and, therefore, excluded.

icance level of 0.1. Thus there should be a good chance to reject hypotheses $H_{0_{CS}}$ and H_{0_C} when more people participated in the experiment. Unfortunately, the tendency of the data shown in Figure 9.7 hints at only a low chance to reject hypothesis H_{0_T}.

Hypothesis	t-value	p-value	rejected at α =0.05
H_{0_Q}	1,9084	0,0354	true
$H_{0_{QS}}$	2,0406	0,0274	true
H_{0_C}	1,1703	0,1278	false
$H_{0_{CS}}$	1,646	0,0577	false
H_{0_T}	0,1731	0,4324	false

Table 9.7.: Results from the One Sided Paired t-Test

Hypothesis	W-value	p-value	rejected at α =0.05
H_{0_Q}	163,5	0,0479	true
$H_{0_{QS}}$	172	0,0251	true
H_{0_C}	149	0,1286	false
$H_{0_{CS}}$	160	0,064	false
H_{0_T}	79	0,4633	false

Table 9.8.: Results from the One Sided Wilcoxon Signed Rank Test

Hypothesis	p-value	rejected at α =0.05
H_{0_Q}	0,0392	true
$H_{0_{QS}}$	0,0392	true
H_{0_C}	0,1917	false
$H_{0_{CS}}$	0,0946	false
H_{0_T}	0,4018	false

Table 9.9.: Results from the One Sided Sign Test

The paired t-test requires that the data is normally distributed which is tested with a Chi-2 goodness of fit test. The results are shown in Table 9.10. The test rejects the normal distribution of the correctness score at a significance level of 0.05. But the Wilcoxon signed rank test and the sign test do not require normal distribution. Consequently, hypotheses H_{0_Q} and $H_{0_{QS}}$ are considered as rejected.

Data	chi-2-value	p-value	rejected at α =0.05
$\mu_{Q_{SED}} \cup \mu_{Q_{KeY}}$	6,3868	0,4954	false
$\mu_{QS_{SED}} \cup \mu_{QS_{KeY}}$	20,2221	0,0051	true
$\mu_{C_{SED}} \cup \mu_{C_{KeY}}$	3,8675	0,7949	false
$\mu_{CS_{SED}} \cup \mu_{CS_{KeY}}$	8,889	0,2607	false
$\mu_{T_{SED}} \cup \mu_{T_{KeY}}$	1,3239	0,9325	false

Table 9.10.: Results from the Chi-2 Goodness of Fit Test for Normal Distribution

9.1.4 Interpretation

The null hypotheses about the correctness are rejected, thus the achieved correctness using SED is significantly better compared to KeY. Although the hypotheses about the confidence are not rejected, there is a tendency that it is higher using SED compared to KeY.

A remaining question is, if the SED helps to answer all kinds of questions or if it is only helpful for some kinds of questions or proof situations. Table 9.11 shows for all proof attempts the asked questions and the expected correct answers. A selected correct answer does not mean that the question of the participant is correct, only that the participant selected it in addition to possibly wrong answers. For each class of KeY experience and the used tool, the percentage how often an answer is selected is given. If a correct answer was more often selected using one of the tools, the value is colored in blue.

The question "is verified" is almost always correctly answered. Only some participants with ≥ 2 years of KeY experience overlooked nodes marked as not verified in the symbolic execution tree in the `ArrayUtil` example. Consequently, they also did not answer why a proof is not open which explains the bad performance of this class compared to the others. This is an indicator that usability of the SED can be improved by highlighting nodes which are marked as not verified by an icon crossed out in red more noticeable.

The correct reason why a proof remains open is often better identified using the SED. In contrast, the reason why a loop invariant in the `ArrayUtil` is not preserved is better identified by using KeY. A possible reason might be, that participants overlooked the loop body termination node in the symbolic execution tree which is marked as not verified. Usability can be improved by highlighting not verified nodes more clearly in the SED. Another reason might be that experienced KeY users also have a better understanding of loop invariants in general.

The question when something does not hold asks for the execution path resulting in a not verified state according to the source code. It is answered more often correct using SED. It seems that the visualized symbolic execution tree helps to understand when a proof could not be closed. Also the thrown exception is clearly labeled in the symbolic execution tree which might be the reason for the better SED results of the "what is thrown" question.

Interestingly, the applied contracts are also often identified better using SED, although KeY has a special feature to list them. This feature was also mentioned in the introduction videos.

The question about executed statements is often answered better using SED. The SED offers additionally to the symbolic execution tree a special feature which highlights reached code members directly in the source code. Monitored participants often overlooked this feature and answered this question by looking at the symbolic execution tree. Interestingly, the dead code in the `Calendar` example is often answered wrongly by KeY users with ≥ 2 years of experience.

The time spent using SED is often higher compared to KeY. A possible reason might be that most of the participants never used the SED before. In the observed evaluations, the participants often need some time to learn how to use the SED. Also some participants spend additional time to discover all features of the SED. This could not be observed when KeY was used. Most likely because the participants used KeY already before. Participants who were not observed may spend time during the evaluation to do something else which could also have a negative affect on the measurement.

Proof Attempt	Question	Answer	KeY experience: All (21) KeY (%)	All (21) SED (%)	None (6) KeY (%)	None (6) SED (%)	< 2 y. (8) KeY (%)	< 2 y. (8) SED (%)	≥ 2 y. (7) KeY (%)	≥ 2 y. (7) SED (%)
Calendar	Is Verified?	No	100	100	100	100	100	100	100	100
	Why not verified?	Inv. not preserved	33	50	50	25	25	50	33	75
		Exc. is thrown	89	100	100	100	100	100	67	100
	When not preserved?	After Else	33	33	50	25	25	50	33	25
	What is thrown?	`ArrayStoreExc.`	89	100	100	100	100	100	67	100
	What is executed?	Line 14	89	100	100	100	75	100	100	100
		Line 32	67	100	100	100	50	100	67	100
		Line 33	67	100	100	100	50	100	67	100
Account	Is Verified?	No	90	100	100	100	75	100	100	100
	Why not verified?	Post. does not hold	40	100	0	100	25	100	100	100
	When does not hold?	Termination 2	40	91	0	100	25	75	100	100
	What is executed?	Line 11	100	100	100	100	100	100	100	100
		Line 12	70	100	67	100	50	100	100	100
		Line 13	80	100	67	100	75	100	100	100
		Line 16	50	100	33	100	25	100	100	100
	What is applied?	MC `withdraw`	70	82	33	33	75	100	100	100
		MC `canWithdraw`	90	100	67	100	100	100	100	100
ArrayUtil	Is Verified?	No	100	91	100	100	100	100	100	75
	Why not verified?	Post. does not hold	40	64	0	67	50	50	67	75
		L. inv. not preserv.	80	73	100	67	75	100	67	50
	When does not hold?	Termination 2	40	64	0	33	50	75	67	75
	When not preserved?	Loop Body T. 1	60	45	67	0	50	75	67	50
	What is executed?	Line 8	90	100	100	100	75	100	100	100
		Line 9	90	91	100	67	75	100	100	100
		Line 10	90	91	100	67	75	100	100	100
		Line 13	70	100	67	100	50	100	100	100
		Line 14	70	91	67	100	50	100	100	75
		Line 17	90	100	100	100	75	100	100	100
		Line 25: init.	90	100	100	100	75	100	100	100
		Line 25: termin.	90	100	100	100	75	100	100	100
		Line 25: increment	90	100	100	100	75	100	100	100
		Line 26	80	100	67	100	75	100	100	100
		Line 27	80	100	67	100	75	100	100	100
		Line 34	50	91	33	100	25	100	100	75
		Line 39	60	91	33	100	50	100	100	75
MyInteger	Is Verified?	No	92	100	100	100	75	100	100	100
	Why not verified?	Post. does not hold	69	88	75	100	75	100	60	50
	What is executed?	Line 9	92	100	75	100	100	100	100	100
		Winning Tool Count	**3**	**32**	**6**	**16**	**0**	**30**	**8**	**7**

Table 9.11.: Comparison of the Given Expected Answers

From Table 9.11 can be concluded that the SED helps in all proof attempts to understand why the proof is still open. Especially participants without or < 2 years of KeY experience achieved better results using the SED. Participants with ≥ 2 years of KeY experience achieved comparable results with KeY and SED. Taking more samples into account, there is a good chance that also that class of users may achieve better results using SED for the mentioned reasons. But what is the opinion of the participants? Which features do they find helpful and which tool do they prefer? These questions are answered with the feedback questionnaire shown to participants after they analyzed the four proof attempts.

The feedback about the KeY features mentioned in the introduction video is shown in Table 9.12. All participants consider the proof tree and the functionality to hide intermediate proof steps as helpful. The goals tab which allows one to directly jump to a goal is often considered as helpful and sometimes it was never used. The list of applied contracts is only helpful in one of the four proof attempts. Taking this into account explains why about half of the participants never used this feature. The feedback about the sequent view is interesting. It is the only way to inspect an open goal and to understand which part of the proof obligation is not yet proven. Most of the participants without KeY experience consider it as not helpful or only little helpful. With some KeY experience it is considered as little helpful with ≥ 2 years of KeY experience as very helpful. This shows that it needs years of KeY experience to understand the system which is required to draw the right conclusions from the sequent of an open goal.

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Proof Tree View	57	19	23	0	0
Goals Tab	19	28	23	4	23
Sequent View	28	19	38	9	4
Hide Intermediate Steps	42	9	23	4	19
List Applied Contracts	14	9	19	4	52

(a) All Participants

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Proof Tree View	50	16	33	0	0
Goals Tab	0	66	0	16	16
Sequent View	16	16	33	33	0
Hide Intermediate Steps	33	16	33	0	16
List Applied Contracts	33	0	0	0	66

(b) No KeY experience only

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Proof Tree View	50	25	25	0	0
Goals Tab	0	0	50	0	50
Sequent View	12	25	62	0	0
Hide Intermediate Steps	37	0	25	12	25
List Applied Contracts	0	12	37	12	37

(c) < 2 years of KeY experience

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Proof Tree View	71	14	14	0	0
Goals Tab	57	28	14	0	0
Sequent View	57	14	14	0	14
Hide Intermediate Steps	57	14	14	0	14
List Applied Contracts	14	14	14	0	57

(d) ≥ 2 years of KeY experience

Table 9.12.: Feedback about KeY Features

Table 9.13 shows the participant's feedback about the SED features. The symbolic execution tree, the highlighting of reached source code, and the truth status tracing is considered as very helpful by most

of the participants. The shown variables are mostly considered as little helpful and sometimes as not helpful. This view is important when the source code performs many operations on the heap. This was not the case in the four given proof attempts. Aliasing was only possible in one of the proof attempts. This explains why memory layouts were often not used and considered as not helpful.

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Symbolic Exec. Tree	57	38	4	0	0
Reached Source Code	52	28	4	0	14
Variables View	4	9	38	14	33
Memory Layouts	0	19	4	14	61
Truth Status Tracing	42	42	14	0	0

(a) All Participants

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Symbolic Exec. Tree	33	66	0	0	0
Reached Source Code	66	0	16	0	16
Variables View	0	16	33	16	33
Memory Layouts	0	16	16	16	50
Truth Status Tracing	66	33	0	0	0

(b) No KeY experience only

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Symbolic Exec. Tree	75	25	0	0	0
Reached Source Code	50	50	0	0	0
Variables View	0	0	50	12	37
Memory Layouts	0	12	0	12	75
Truth Status Tracing	25	50	25	0	0

(c) $<$ 2 years of KeY experience

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Symbolic Exec. Tree	57	28	14	0	0
Reached Source Code	42	28	0	0	28
Variables View	14	14	28	14	28
Memory Layouts	0	28	0	14	57
Truth Status Tracing	42	42	14	0	0

(d) $\geq$ 2 years of KeY experience

Table 9.13.: Feedback about SED Features

The feedback questionnaire offers also the optional opportunity to give feedback as free text. The results are shown in Table 9.14. Feedback with ID 5 and ID 6 is about the evaluation itself. Indeed, the use of three windows (Eclipse, Evaluation Wizard, and KeY) is painful. The monitored evaluations were done for this reason with two high resolution screens. One showing the evaluation wizard, and the other one showing Eclipse (SED and source code) and KeY. The Feedback with ID 2 just says the participant has no experience with KeY and SED.[8]

Constructive ideas for improvements of the SED are given in feedback with ID 1, ID 3, ID 4, ID 7, and ID 8. First, KeY and SED operate on first-order terms as the internal representation of JML expressions instead on the JML expressions itself. The mapping from JML expressions to terms is a barrier for new users which could for instance be avoided in the SED which tries to be as close to the source code as possible. Second, the truth status tracing highlights formulas in red, green and yellow. People with a red-green blindness cannot distinguish these colors and consequently cannot interpret the results. Usability can be easily improved by allowing to change the used colors. Third, to extend the SED for interactive verification is ongoing work. This will allow one to use the SED as an equivalent replacement for KeY.

[8] To find out to which tool the participant refers to, the participant's background was considered.

Fourth, branch condition and termination nodes are not related to the source code and consequently selection is not synchronized. Many monitored participants struggled with that. In most cases the situation can be improved by highlighting the source code element related to the parent node. But in interactively performed proofs a branch condition is not necessarily related to the previously executed statement. Fifth, view **Variables** shows the symbolic values of each location. A feature to find a concrete model fulfilling the path condition could be helpful. Also generating a test case as witness that a path is feasible might help to increase the understanding of the symbolic execution tree.

ID	Feedback
1	"Very well made interactive evaluation! Introduction to SED was too brief to understand many things. I found naming of some things counter intuitive. I think that presentation of logical expressions in a different language than the spec language opens a wide gap between the specification and the proof."
2	"lack of practice on the tool."
3	"Please change the colors."
4	"SED looks quite impressive, and also appears to be more stable than I expected, good work. Debugging proof attempts only using KeY is predictably tedious, but KeY is also a more general tool than SED. I suspect that it is possible to further simplify and streamline the SED user interface; only show the symbolic execution tree and the source code, and everything else only on demand or depending on the context. When selecting a node in the symbolic execution tree, it is sometimes not easy to identify the corresponding statement in code. Some nodes don't seem to correspond to statements; in this case, still something should be shown in the code to not confuse the user."
5	"The evaluation is a bit painful to complete, lots of windows, the videos have a bit of too quick pace."
6	"Umfragefenster störte beim Umgang mit Eclipse und KeY. Quellcode für die KeY-Sachen sollte sichtbar sein." *Approximate Translation: Evaluation wizard interferes with handling of Eclipse and KeY. Source code for KeY proofs should be visible.*
7	"Make SED a default interface for KeY."
8	"shown variables fand ich in dem beispiel nicht hilfreich, da ich nicht gesehen habe bei welchem wert von der variable summand es schief ging. dachte das kann ich da sehen." *Approximate Translation: I considered the feature 'shown variables' as not helpful in the example, because I could not see at which value of variable summand it went wrong. I assumed I could see this.*

Table 9.14.: Feedback of Participants

The participants are also asked which tool they prefer to inspect proof attempts. The results are shown in Table 9.15. As both tools have their strengths it is not surprising that many participants like to choose the tool based on the actual proof. Otherwise, participants without or < 2 years of KeY experience prefer the SED. Participants with ≥ 2 years of KeY experience prefer KeY on the other hand. This confirms the already made observation that the SED helps to reduce the barrier to learn the KeY system. To satisfy the needs of the KeY experts, features like the inspection of goal sequents are missing.

		KeY experience		
	All (%)	None (%)	< 2 y. (%)	≥ 2 y. (%)
KeY	14	0	0	42
KeY and SED, both are equally good	0	0	0	0
KeY and SED, depending on the proof	42	33	50	42
KeY and SED, both are equally bad and should be improved	0	0	0	0
SED	42	66	50	14

Table 9.15.: Participants Tool Preference

9.2 Reviewing Code Evaluation

Writing and reading source code is the daily business of software developers. Whenever the behavior of the program is not well understood or the program does not behave as expected, a debugger is an important tool. After input values driving execution to the point of interest are found, the developer controls execution and comprehends each performed step until the program behavior is fully understood. Suitable input values are often provided by failed test cases or by bug reports. Otherwise, it can be challenging to determine the exact conditions under which the inspected code exhibits the faulty behavior. Complicating is also the fact, that only one specific execution path is inspected in a debugging session. To inspect a different execution path, debugging needs to start from scratch with different input values.

Additional to debugging and testing, the source code can be studied in a static code review. Possible goals of a review might be to find defects or to improve design and code quality. A review can be performed by a team or by a single person. An example for a team review is an inspection [47, 48] in which persons with different roles study the source code under aspects according to their role. Also a single developer can review source code as part of her personal software process (PSP) [71] to ensure that she is satisfied with the achieved quality. Checklists are often used to guide a review and to define the criteria under which the source code is reviewed.

Reviews can be performed for all kinds of documents without the need for any additional tool support. However, the Symbolic Execution Debugger (Chapter 6) allows one to directly execute the studied source code. Execution can be controlled like in a traditional debugger, but all feasible execution paths are explored at once. The experiment described in this section evaluates, whether the use of the SED in a review helps to increase program understanding and thus to find defects.

The planning of the experiment is presented in Section 9.2.1. How the experiment was executed is then presented in Section 9.2.2. Finally, Section 9.2.3 analyzes the collected data before it is interpreted in Section 9.2.4.

9.2.1 Experiment Planning

This section presents the experiment planning. First, the scope of the experiment is defined. Second, the involved variables are identified and used to define hypotheses. Third, the selected design type is discussed. Fourth, the experiment instrumentation including the chosen source code examples is presented. Finally, threats to validity are discussed.

Scope

The purpose of the experiment is to evaluate the effectiveness and efficiency of a Java source code review with and without the SED. Traditional debugging is explicitly allowed in a direct code review (DCR) without the SED. The experiment is run from the research perspective to find out if the SED significantly improves the review quality. During the evaluation Java source code and related questions are shown to the participants to measure their performance. Each code example realizes a complete functionality and is inspired by literature or interesting problems. The expected behavior is always described by comments and sometimes additionally by JML specifications. The asked questions are a kind of checklist under which participants review the source code. The experiment was performed with engineers at Bosch Engineering GmbH and announced publicly on the KeY website and the JML and KeY mailing lists. The scope of the experiment is summarized as follows:

Analyze *code review with and without the SED*
for the purpose of *evaluation*
with respect to *effectiveness and efficiency*
from the point of view of the *researcher*
in the context of *Java developers in industry and research*.

Variables Selection

The variables involved in the experiment are shown in Table 9.16. The independent variables determine the cases for which the dependent variables are sampled. A value of an independent variable changed during the experiment is called treatment. Here, the independent variables are M with treatments SED and DCR, and S with the six code examples to review as treatments. During the experiment, a participant is asked to review a code example of S with a method of M. The controlled variables are used to classify users according to their experience with Java, JML, symbolic execution and SED in an ordinal scale. The separation between less and more than two years is made to separate beginners from experienced users assuming that this is roughly the time needed to understand Java, JML and symbolic execution well enough for this evaluation. As the SED is rather new, it is assumed that participants do not have a lot of experience with it. For this reason, the separation between beginners and experienced users is set to one year of SED experience. Efficiency and Effectiveness are measured as described in Section 9.1.1.

Type	Name	Values	Description
Independent Variable	M	{SED, DCR}	The compared methods.
	S	{`BankUtil`, `IntegerUtil`, `MathUtil`, `ValueSearch`, `ObservableArray`, `Stack`}	The reviewed source code example and related questions.
Controlled Variable	E_{Java}	$\{none, < 2\ years, \geq 2\ years\}$	The experience with Java.
	E_{JML}	$\{none, < 2\ years, \geq 2\ years\}$	The experience with JML.
	E_{SE}	$\{none, < 2\ years, \geq 2\ years\}$	The experience with symbolic execution.
	E_{SED}	$\{none, < 1\ years, \geq 1\ years\}$	The experience with SED.
Dependent Variable	Q_{tm}	Integer	The number of correctly answered questions per treatment tm of M.
	QS_{tm}	Real	The correctness score per treatment tm of M.
	C_{tm}	Integer	The achieved confidence score per treatment tm of M based on Q.
	CS_{tm}	Real	The achieved confidence score per treatment tm of M based on QS.
	T_{tm}	Integer	The time needed to answer questions of a treatment tm of M in seconds.

Table 9.16.: Variables

Hypothesis Formulation

A review for which the SED is used is considered to be more effective and more efficient as a review without the SED. To empirically test whether this claim expressed as alternative hypotheses is valid, the null hypotheses need to be rejected, see Table 9.17.

Kind	Name	Hypothesis	
Null hypothesis	H_{0_Q}	$\mu_{Q_{SED}} = \mu_{Q_{DCR}}$	with $\mu_{Q_{Treatment}} = \frac{Q_{Treatment}}{\#questionsOfTreatment} \in \{x \in \mathbb{Q} \mid 0 \leq x \leq 1\}$
	$H_{0_{QS}}$	$\mu_{QS_{SED}} = \mu_{QS_{DCR}}$	with $\mu_{QS_{Treatment}} = \frac{QS_{Treatment}}{\#questionsOfTreatment} \in \{x \in \mathbb{Q} \mid 0 \leq x \leq 1\}$
	H_{0_C}	$\mu_{C_{SED}} = \mu_{C_{DCR}}$	with $\mu_{C_{Treatment}} = \frac{C_{Treatment}}{\#questionsOfTreatment} \in \{x \in \mathbb{Q} \mid -2 \leq x \leq 2\}$
	$H_{0_{CS}}$	$\mu_{CS_{SED}} = \mu_{CS_{DCR}}$	with $\mu_{CS_{Treatment}} = \frac{CS_{Treatment}}{\#questionsOfTreatment} \in \{x \in \mathbb{Q} \mid -2 \leq x \leq 2\}$
	H_{0_T}	$\mu_{T_{SED}} = \mu_{T_{DCR}}$	with $\mu_{T_{Treatment}} = \frac{T_{Treatment}}{timeOfAllTreatments} \in \{x \in \mathbb{Q} \mid 0 \leq x \leq 1\}$
Alternative hypothesis	H_{1_Q}	$\mu_{Q_{SED}} > \mu_{Q_{DCR}}$	
	$H_{1_{QS}}$	$\mu_{QS_{SED}} > \mu_{QS_{DCR}}$	
	H_{1_C}	$\mu_{C_{SED}} > \mu_{C_{DCR}}$	
	$H_{1_{CS}}$	$\mu_{CS_{SED}} > \mu_{CS_{DCR}}$	
	H_{1_T}	$\mu_{T_{SED}} < \mu_{T_{DCR}}$	

Table 9.17.: Hypotheses

Choice of Design Type

An important design decision of the experiment is that a participant should benefit from her participation. To achieve this, each participant reviews source code with and without the SED resulting in a paired comparison design. In case the participant is unfamiliar with the SED, this allows her to tryout the SED and to decide if the SED is helpful for her in a code review. The final design type is a paired comparison design as shown in Table 9.18.

	Example 1	Example 2	Example 3	Example 4	Example 5	Example 6
SED	Subject_n	Subject_n	Subject_n	Subject_{n+1}	Subject_{n+1}	Subject_{n+1}
DCR	Subject_{n+1}	Subject_{n+1}	Subject_{n+1}	Subject_n	Subject_n	Subject_n

Table 9.18.: Paired Comparison Design

The general design principles randomization, blocking and balancing are applied as well. The order of code examples is random. The first three examples are always reviewed with or without the SED and the next three examples the other way round. The decision whether the SED is used first is by random as well. This avoids multiple switches between the reviewing method which could confuse the participant. Also a participant not familiar with the SED (or the direct code review) will gain experience towards the later examples. The server used to collect evaluation results guarantees that all possible permutations of code examples will be evaluated equally often. The server also ensures that each permutation is evaluated with and without SED first.

The performance of the participants may depend on their experience with Java which is used for blocking. Considering only completed evaluations, balancing is automatically achieved for the reviewing method by the chosen design, because each participant reviews code examples with and without SED. The number of participants who reviewed a source code example might be not balanced in case participants do not finish the evaluation or decided to review only four of the six code examples.

Instrumentation

The evaluation requires no knowledge about JML, symbolic execution or the SED, only a basic understanding of Java is required. Consequently, the evaluation has to be self explaining which is ensured by additional instructions.

During the evaluation, a participant reviews source code with and without the SED. As the SED is an Eclipse extension, the evaluation itself is implemented as an Eclipse wizard. The wizard is opened in an additional window so that Eclipse itself remains fully functional (see Figure 9.8).

The evaluation wizard is separated into two phases in which information is collected and sent to the server. The first phase collects the background knowledge of the participant and receives the order of code examples as well as whether the SED is used first. The evaluation itself is then performed in the second phase. If the participant cancels the evaluation in the second phase, she is asked to send intermediate results to the server. When she opens the evaluation wizard the next time, she is asked to recover the previous state to continue the already started evaluation. The following enumeration shows the steps in detail:

1. Initialization Phase
   a) *Terms of Use*: A text explaining the terms of use is shown to the participant. The participant needs to accept them before she can continue.
   b) *Background Knowledge*: The background knowledge of Java, JML, symbolic execution and SED is collected.

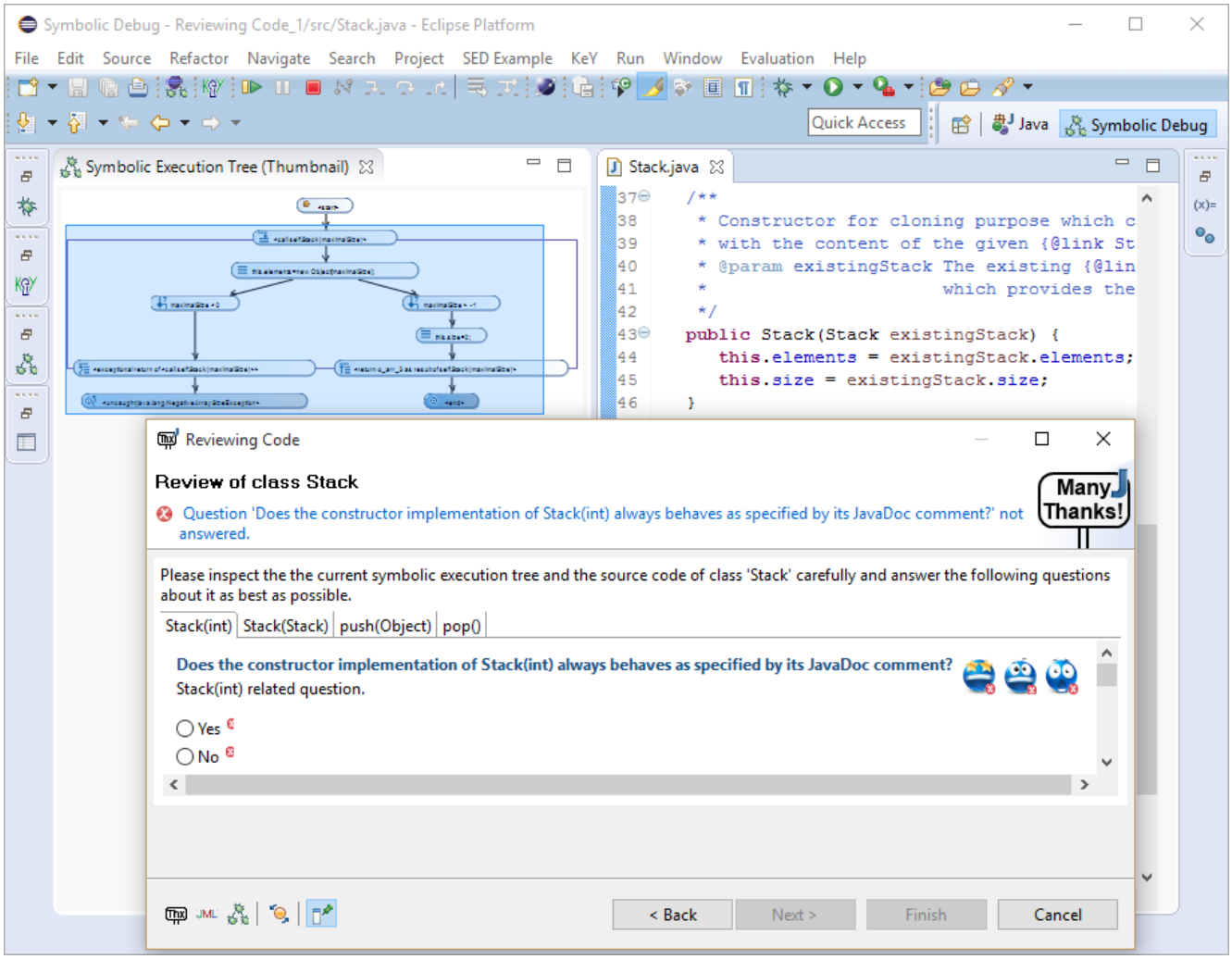


Figure 9.8.: Screenshot of the Reviewing Code Evaluation Wizard

c) *Extent*: The participant can select between reviewing four (45 to 60 minutes) or six (45 to 90 minutes) code examples according to the time she likes to spend for the evaluation.

d) *Sending Data*: The participant accepts sending the background knowledge to the server. The evaluation wizard receives as answer the order of code examples and if the SED should be used first for the evaluation phase.

2. Evaluation Phase

a) *Evaluation Instructions*: A video shows how to answer questions and how to use the evaluation wizard.

b) *JML*: A textual documentation introducing the used features of JML.

c) *SED or DCR Instructions*: A video explaining needed features and best practices to review source code with or without the SED. In case of a direct code review (DCR), the usage of the Eclipse Java Debugger is explained.

d) *Code Example 1*: The first code example and related questions.

e) *Code Example 2*: The second code example and related questions.

f) *Code Example 3*: Optionally, the third code example and related questions if the extent is six.

g) *The complementary SED or DCR Instructions*: The remaining instruction video.

h) *Code example 4*: The fourth code example and related questions.

i) *Code example 5*: The fifth code example and related questions.

j) *Code Example 6*: Optionally, the sixth code example and related questions if the extent is six.

k) *Feedback about SED and Evaluation*: The participant is asked to rate the helpfulness of the SED features (see Section 6.2) mentioned in the instruction video. These are (i) the visualized symbolic execution tree, (ii) the highlighted statements reached during symbolic execution, (iii) the **Variables** view and (iv) the path condition shown in the **Properties** view.

l) *Sending Data*: The participant accepts sending the evaluation results to the server.

m) *Completed Message*: A thank you for participation image is shown. The participant can now finish the wizard.

The `BankUtil` code example (Section A.2.1) implements a stair-step table lookup, inspired by [100, page 427]. The source code does not adhere to its documentation because a wrong value is returned for an age above 35.

The `IntegerUtil` code example (Section A.2.2) is inspired by [137, page 255]. The challenge is to find the mistake that in one case `y` is returned instead of `x`.

A potential overflow needs to be discovered in the `MathUtil` code example (Section A.2.3) which causes an `ArrayIndexOutOfBoundsException`. Such an overflow was part of Java's binary search implementation, see bug item JDK-5045582[9].

The statement **`return false;`** in the `ValueSearch` code example (Section A.2.4) is dead code. The surrounding method `accept` is only called within the loop in method `search`. The loop guard already ensures that the then branch of `accept` is never taken. Such circumstances are hard to detect for instance by test case generation tools based on symbolic execution. By unrolling the loop, there might be a future loop iteration in which the then branch will be taken. Further, there is a defect in method `find`. The parameter `value` is never used. Furthermore, in case that the `array` is **`null`**, a `NullPointerException` is thrown.

The `ObservableArray` code example (Section A.2.5) is inspired by [24, page 265]. The constructor and method `setArrayListeners` behave according to their documentations. But method **`set`** has several problems. First, an `ArrayIndexOutOfBoundsException` is thrown in case the index is outside the array bounds. Second, an `ArrayStoreException` is thrown if the element is not compatible to the component type of the array. Third, not all at call time available observers are informed about the change in case that an observer changes the available observer during the event. Furthermore, if an observer sets `arrayListeners` to **`null`**, a `NullPointerException` is thrown.

Finally, the `Stack` code example (Section A.2.6) is inspired by [24, page 24]. The first constructor which creates a new stack throws a `NegativeArraySizeException` if the maximal size is negative. The second constructor which clones a stack does not behave as documented, because the clone shares the same `elements` array and is thus not independent. Further, the constructor throws a `NullPointerException` if the existing stack is **`null`**. Method `push` behaves as documented, but method `pop` does not remove the top element from the stack which violates the class invariant.

The questions and available answers for each code example are always generated following the same schema. On the top level, the participant is asked for each method and constructor to review whether the implementation behaves as documented and to select the statements which can be executed. In case the source code behaves not as documented, several possible reasons are offered. In case an undocumented exception is thrown, the participant is also asked which one. If a class invariant is available, the participant is asked whether it is preserved or established. If this is not the case, the participant is asked to select the part of the class invariant which is broken. In case the participant thinks that none of the mentioned reasons is correct, she can enter the supposed reason as free text. For methods with a return value, the participant is asked to select valid claims about the returned value.

[9] bugs.java.com/bugdatabase/view_bug.do?bug_id=5045582

For source code examples reviewed with the SED the participant is asked to rate the helpfulness of the symbolic execution tree. For reviews without the SED the participant is asked if she used the traditional Java debugger. If so, she is also asked how helpful debugging was and to submit the written code (if still available).

In a review using the SED, a fully explored symbolic execution tree is shown to the user. Interactive symbolic execution is avoided, to ensure that all participants review the source code under the same aspects. In direct code reviews without the SED, a main method with a call of the code to review is provided. Required values of parameters are missing and have to be provided by the participant. The code should only help participants without or little Java experience to start a debugging session.

Validity Evaluation

"*Conclusion validity* concerns the statistical analysis of results and the composition of subjects." [135, page 185] The hypotheses of this experiment are tested with well known statistical techniques. Threats to conclusion validity are the low number of samples and the validity of the quality of answers. Subjects may fake answers to compromise the experiment. However, several participants are known people and in addition, some were monitored during the evaluation. The motivation of subjects to compromise the experiment is considered to be low because of the mentioned reasons.

"*Internal validity* concerns matters that may affect the independent variable with respect to causality, without the researchers knowledge." [135, page 185] Maturation is a threat to internal validity in this experiment. Reviewing source code is a time intensive task and the estimated participation time is up to 90 minutes. Participants may get tired or bored during the experiment. As each method is applied to two or three code examples, participants may learn how to use it which is desired. The learning between both methods and the order of code examples is considered as not critical as randomization is applied. Other threats are considered to be uncritical.

The subjects have most likely no experience with SED as it is relatively new. This is not critical as the instrumentation introduces the relevant functionality of both methods. There might be a threat that the subjects are not willing to work with SED. However, SED is designed to support code reviews. The bias about the experience with the SED is against the hypothesis. There is a risk that subjects lack motivation and thus answer the questions not seriously. However, participation is voluntary and can be done at any time.

"*Construct validity* concerns generalisation of the experiment result to concept or theory behind the experiment." [135, page 185] A threat to construct validity is that the chosen code examples might be not representative in general. To mitigate this, the code examples are inspired by the literature or interesting problems. Other threats to construct validity are considered to be uncritical. Only the motivation of the experiment, to compare direct source code review with an inspection using the SED, is given to the participants. Therefore, subjects might guess the expected outcome of the experiment because the SED is a new tool. However, the exact hypotheses and the related measurements are unknown. In addition, subjects do not have any advantage or disadvantage about the outcome of this experiment.

"*External validity* concerns generalisation of the experiment result to other environments than the one in which the study is conducted." [135, page 185] A threat to external validity is that the source code is kept to a minimum. This is required to reduce the time participants need to review source code and its documentation. Real Java code is much more complex. Symbolic execution with specifications on the other side is modular and only a small part of the source code is considered. Subjects are selected randomly and their experience varies from none to experts. Consequently, the selection of subjects is not a threat to external validity.

To conclude, there are threats to the validity of the experiment. Hence, conclusions drawn from the results of the experiment are valid within the limitations of the threats.

9.2.2 Execution

The experiment started in September 2015 with the staff of the Software Engineering group at TU Darmstadt. This includes students, PhD students and postdocs. Each of these participants were monitored during the evaluation to improve the instructions and answers. Questions and answers remained stable after the first participant. The results of the first participant are excluded for this reason.

The evaluation was then performed with engineers at Bosch Engineering GmbH. None of the engineers use Java in their daily business and the Java experience was none or less then a year. However, participants were interested in new methods and liked to try out the SED. In total, 11 engineers started the evaluation. One participant canceled the evaluation and three participants did not submit the answers. Additionally, one participant did not follow the instructions and used the SED for all code examples. Consequently, the results of six participants are considered as valid.

The evaluation became then public in October 2015 and was announced on the KeY website and the JML and KeY mailing lists. The evaluation is deployed as a preconfigured Eclipse product. Installation instructions and download links are available on the KeY website[10]. The main steps are to download the Eclipse product, to unpack it, to start it, to perform the evaluation and then to delete it. An installation is not required, so the subject's system remains untouched.

Until end of 2015, 25 participants started the evaluation, but only 18 completed it. Twelve of the participants, who finished the evaluation, were monitored during the evaluation.

The background knowledge of the participants is shown in Figure 9.9. The experience of participants with knowledge in JML and symbolic execution is evenly distributed. Most of the participants had more than two years of Java experience and none with SED.

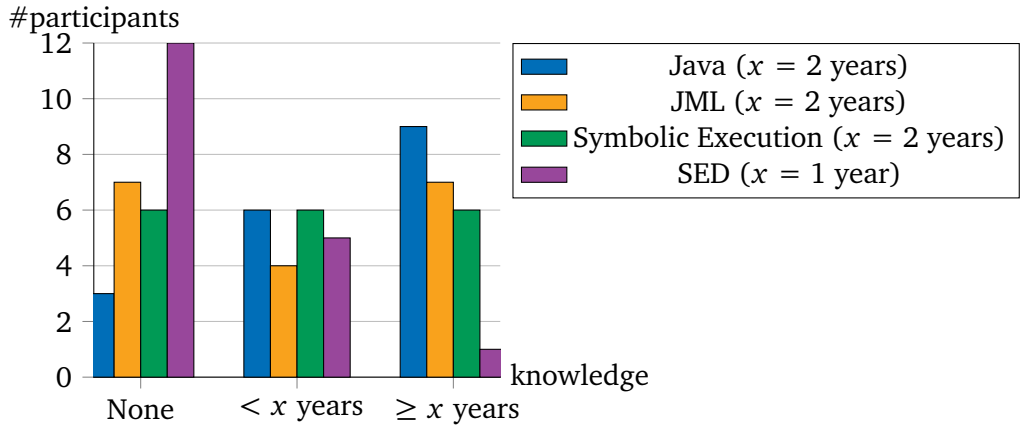


Figure 9.9.: Knowledge of Participants

The relation between the Java and the SED experience is shown in Table 9.19. It shows that all participants with SED experience have at least two years of Java experience.

		SED		
		None	< 1 year	≥ 1 year
Java	None	3	0	0
	< 2 years	6	0	0
	≥ 2 years	3	5	1

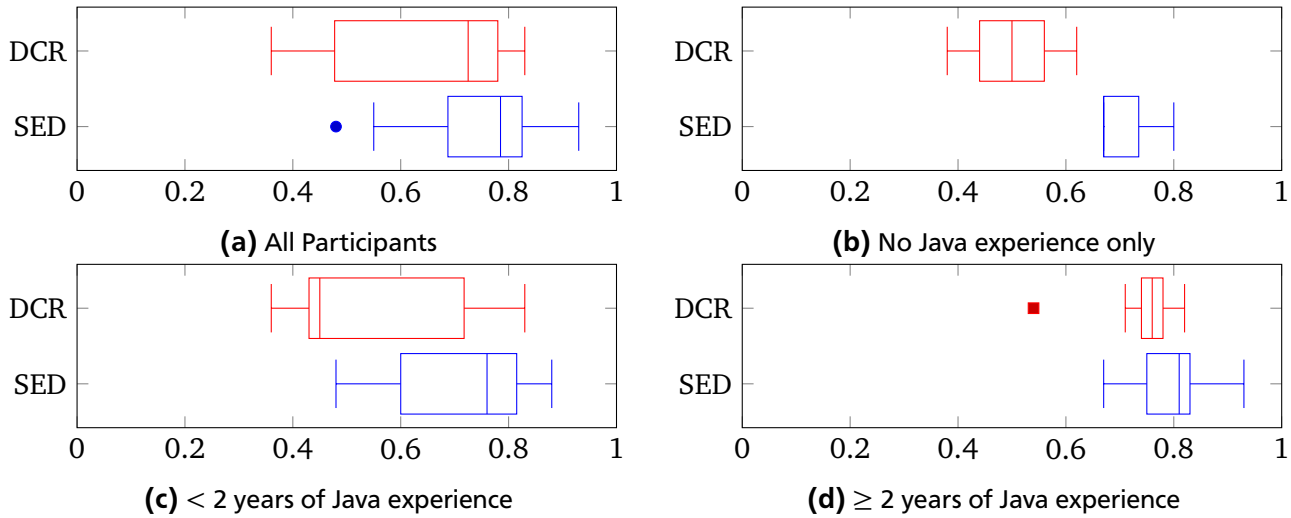
Table 9.19.: Java vs SED Experience of Participants

[10] key-project.org/eclipse/SED/ReviewingCode.html

9.2.3 Analysis

Each of the boxplots in Figures 9.10 to 9.14 show the measured data of a dependent variable. They are constructed in the same manner as in Section 9.1.3.

The boxplots in Figure 9.10 show the measured number of correctly answered questions. The maximal value 1 is achieved if all questions are correctly answered. The opposite 0 accordingly if not a single question is correctly answered. The boxplots in Figure 9.10a are based on the results of all participants, whereas only a class of Java experience is considered in Figures 9.10b to 9.10d. In all boxplots, the achieved correctness is better using SED.

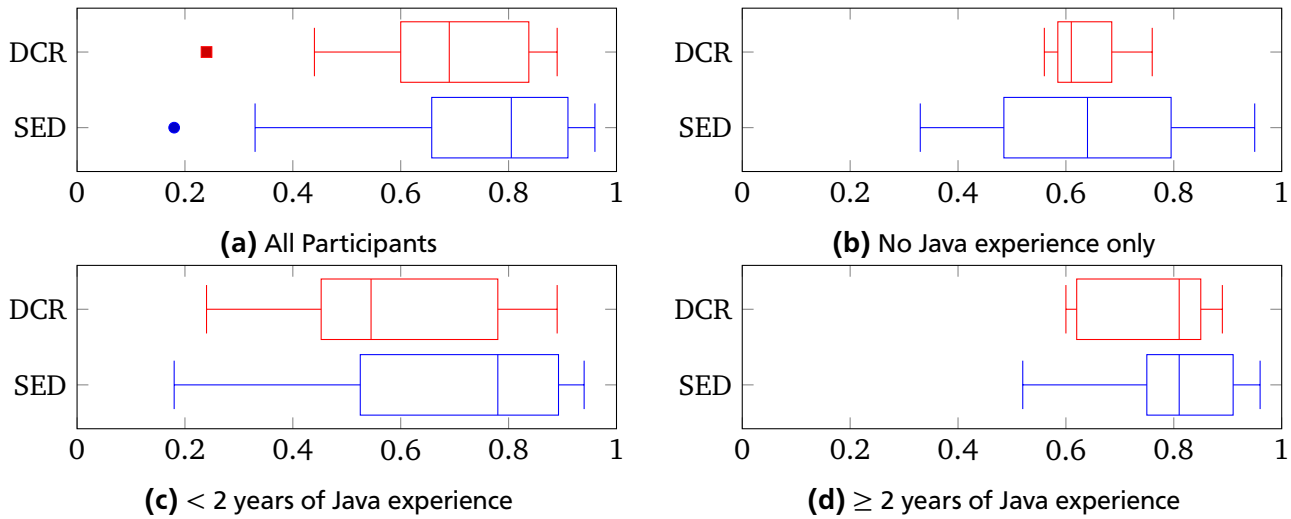


(a) All Participants **(b)** No Java experience only

(c) $<$ 2 years of Java experience **(d)** $\geq$ 2 years of Java experience

Figure 9.10.: Correct Answers

Analogous to the number of correctly answered questions shows Figure 9.11 the measured correctness score taking also partially correct answers into account. Again, the achieved correctness is always better using SED. In the class of participants without Java experience, the achieved correctness using SED varies a lot.



(a) All Participants **(b)** No Java experience only

(c) $<$ 2 years of Java experience **(d)** $\geq$ 2 years of Java experience

Figure 9.11.: Correctness Score

The achieved confidence behaves similar to the correctness. As Figure 9.12 shows, the confidence is better using SED except for the class of participants without Java experience. In this class, the upper

quartile is better in a direct code review without the SED. However, only three participants are in this class which is not enough to draw meaningful conclusions.

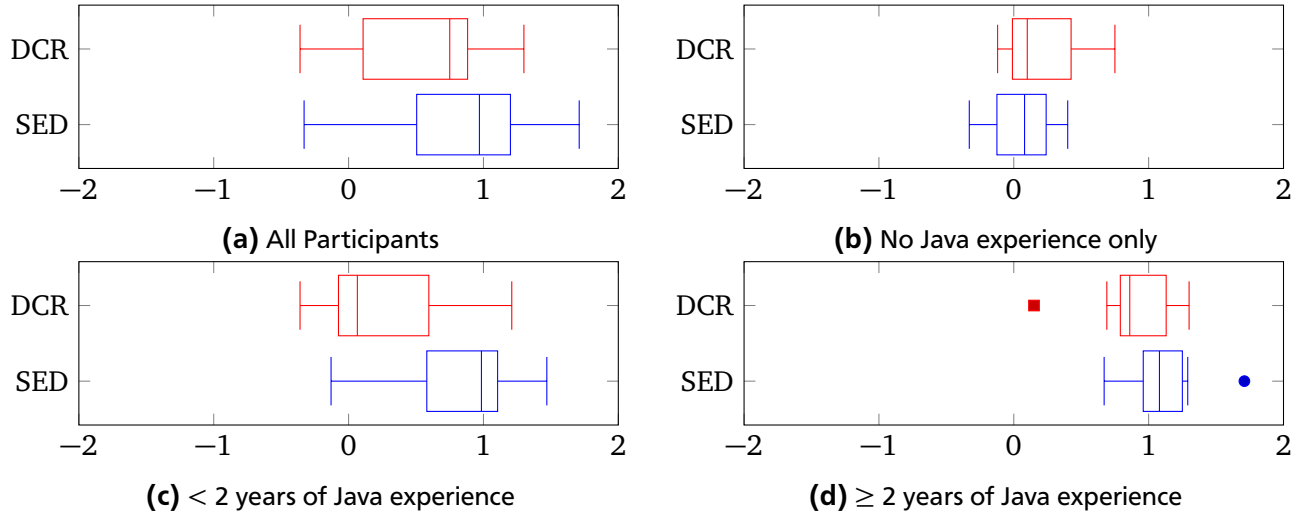


Figure 9.12.: Confidence Score

Figure 9.13 shows the achieved confidence taking also partially correct answers into account. In all classes the achieved confidence is better using SED. However, the confidence achieved with the SED in the class of participants without Java experience varies a lot.

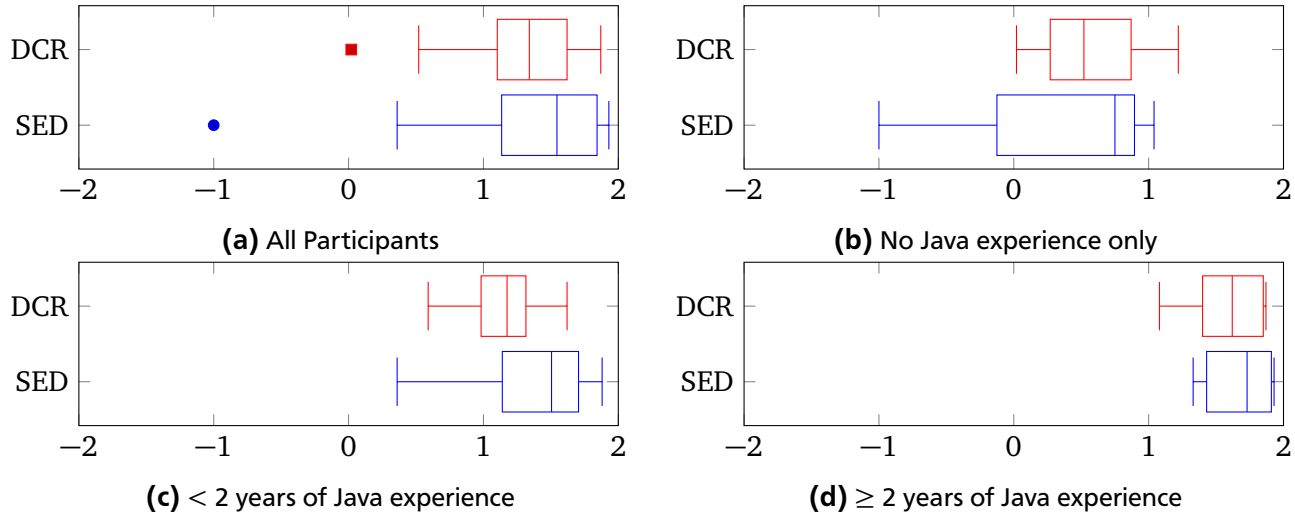


Figure 9.13.: Confidence Score of Partially Correct Answers

The measured time[11] is shown in Figure 9.14. A value of 1 means that a participant spent 100% of the time using one method. The opposite is 0 when a participant spent 0% of the time using a method. In all boxplots, the review time is less using SED.

[11] The times measured for three of the participants were invalid and, therefore, excluded.

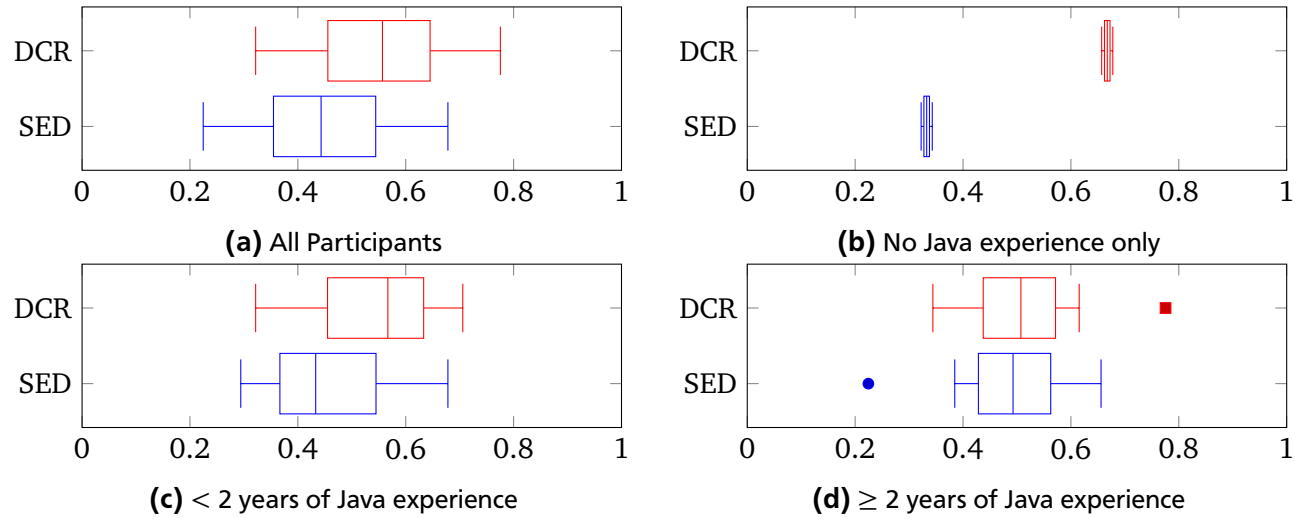


Figure 9.14.: Time

The null hypotheses of Table 9.17 can be rejected according to Wohlin et al. [135] using a one sided *Paired t-Test*, *Wilcoxon Signed Rank Test* or a *Sign Test*. The *Paired t-Test* requires that the data is normally distributed which is tested according to Wohlin et al. [135] with a *Chi-2 Goodness of Fit Test*. The *Sign Test* was performed by my own implementation following to Wohlin et al. [135], for all others tests the *Apache Commons Mathematics Library* was used.

The results of the tests are shown in Tables 9.20 to 9.22. The tests are only performed using the results of all participants and not with respect to the separate classes, because the number of participants in each class of Java experience on each own is not enough to reject a hypothesis. The significance level is set to 0.05 meaning that there is a 5% chance at which a hypothesis is wrongly rejected.

All tests reject the correctness hypotheses H_{0_Q} at the given significance level. Additionally, the paired t-Test and the Wilcoxon signed rank test reject the confidence hypotheses H_{0_C}. Looking at the promising boxplots, there should be a good chance to reject the other hypotheses when more people participate in the experiment.

Hypothesis	t-value	p-value	rejected at α =0.05
H_{0_Q}	3,1589	0,0029	true
$H_{0_{QS}}$	0,8936	0,192	false
H_{0_C}	2,1882	0,0215	true
$H_{0_{CS}}$	0,6326	0,2677	false
H_{0_T}	1,2569	0,1147	false

Table 9.20.: Results from the One Sided Paired t-Test

Hypothesis	W-value	p-value	rejected at α =0.05
H_{0_Q}	157,5	0,0003	true
$H_{0_{QS}}$	101,5	0,2475	false
H_{0_C}	132	0,0216	true
$H_{0_{CS}}$	104	0,2211	false
H_{0_T}	82	0,1147	false

Table 9.21.: Results from the One Sided Wilcoxon Signed Rank Test

Hypothesis	p-value	rejected at α =0.05
H_{0_Q}	0,0021	true
$H_{0_{QS}}$	0,5982	false
H_{0_C}	0,1189	false
$H_{0_{CS}}$	0,2403	false
H_{0_T}	0,1509	false

Table 9.22.: Results from the One Sided Sign Test

The paired t-test requires that the data is normally distributed which is tested with a Chi-2 goodness of fit test. The results are shown in Table 9.23. Except for the correctness score, the test rejects normal distribution at a significance level of 0.05. But the Wilcoxon signed rank test and the sign test do not require normal distribution. Consequently, hypotheses H_{0_Q} and H_{0_C} are considered as rejected.

Data	chi-2-value	p-value	rejected at α =0.05
$\mu_{Q_{SED}} \cup \mu_{Q_{DCR}}$	22,734868	0,000379	true
$\mu_{QS_{SED}} \cup \mu_{QS_{DCR}}$	8,563456	0,127795	false
$\mu_{C_{SED}} \cup \mu_{C_{DCR}}$	14,365368	0,013448	true
$\mu_{CS_{SED}} \cup \mu_{CS_{DCR}}$	34,7653	0,000002	true
$\mu_{T_{SED}} \cup \mu_{T_{DCR}}$	15,0506	0,0101	true

Table 9.23.: Results from the Chi-2 Goodness of Fit Test for Normal Distribution

9.2.4 Interpretation

The null hypotheses about the correctness and the trust are rejected, thus the achieved correctness and the achieved confidence are significantly better using SED. Although the other hypotheses are not rejected, there is a tendency that the review results are better using SED.

A remaining question is, if the SED helps to answer all kinds of questions or if it is only helpful for some kinds of questions or code examples. Table 9.24 shows for all code examples the asked questions and the expected correct answers. A selected correct answer does not mean that the question of the participant is correct, only that the participant selected it in addition to possibly wrong answers. For each class of Java experience and the used method, the percentage how often an answer is selected is given. If an answer was more often correctly answered using one of the methods, the value is colored in blue. The percentage is left empty in case that it was never answered.

Source Code	Question	Answer	Java experience: All (18) DCR (%)	All (18) SED (%)	None (3) DCR (%)	None (3) SED (%)	< 2 y. (6) DCR (%)	< 2 y. (6) SED (%)	≥ 2 y. (9) DCR (%)	≥ 2 y. (9) SED (%)
ObservableArray	As documented?	Yes	100	100		100	100	100	100	100
	Invariant established?	Yes	88	71		100	100	67	83	67
	What is executable?	Line 30	100	100		100	100	100	100	100
		Line 31	100	100		100	100	100	100	100
		Line 33	100	100		100	100	100	100	100
		Line 34	100	100		100	100	100	100	100
	As documented?	No	88	71		0	100	67	83	100
	What is wrong?	Not all informed.	12	0		0	0	0	17	0
		Exc. is thrown	88	71		0	100	67	83	100
	What is thrown?	`NullPointerExc.`	12	71		0	50	67	0	100
		`ArrayIndex OutOfBoundsExc.`	75	71		0	50	67	83	100
		`ArrayStoreExc.`	25	71		0	0	67	33	100
	Invariant preserved?	Yes	88	100		100	50	100	100	100
	What is executable?	Line 51	100	100		100	100	100	100	100
		Line 52	100	100		100	100	100	100	100
		Line 61	100	100		100	100	100	100	100
		Line 66: init.	88	100		100	50	100	100	100
		Line 66: termin.	88	100		100	50	100	100	100
		Line 62 increment	0	0		0	0	0	0	0
		Line 67	88	100		100	50	100	100	100
		Line 68	88	100		100	50	100	100	100
	As documented?	Yes	100	100		100	100	100	100	100
	Invariant preserved?	Yes	88	100		100	50	100	100	100
	What is executable?	Line 79	100	100		100	100	100	100	100
BankUtil	As documented?	No	100	100	100	100		100	100	100
	What is wrong?	Wrong if age ≥ 35	100	100	100	100		100	100	100
	What is returned?	570	100	100	100	100		100	100	100
	What is executable?	Line 18	100	100	100	100		100	100	100
		Line 19	100	100	100	100		100	100	100
		Line 20	100	100	100	100		100	100	100
		Line 21	100	100	100	100		100	100	100
		Line 22	100	100	100	100		100	100	100
		Line 23	100	100	100	100		100	100	100
		Line 24	100	100	100	100		100	100	100
		Line 26	100	100	100	100		100	100	100
		Line 28	86	100	0	100		100	100	100
	What is true?	Not in `ageLimits`	29	29	0	0		33	33	33
		In `insuranceRates`	100	86	100	100		100	100	67
		Not in `insuranceR.`	86	86	0	100		100	100	67
		Positive	100	86	100	100		100	100	67

Source Code	Question	Answer	Java experience: All (18) DCR (%)	All (18) SED (%)	None (3) DCR (%)	None (3) SED (%)	< 2 y. (6) DCR (%)	< 2 y. (6) SED (%)	≥ 2 y. (9) DCR (%)	≥ 2 y. (9) SED (%)
ValueSearch	As documented?	No	88	78	100	0	75	100	100	83
	What is wrong?	S. criteria unused	62	22	100	0	50	0	67	33
		Exc. is thrown	0	67	0	0	0	100	0	67
	What is thrown?	`NullPointerExc.`	0	67	0	0	0	100	0	67
	What is executable?	Line 20	100	89	100	0	100	100	100	100
		Line 30	75	78	100	0	75	50	67	100
		Line 34	62	78	0	0	75	50	67	100
	What is true?	-1 returned	50	78	0	0	75	100	33	83
		0 returned	62	78	100	0	75	50	33	100
		`array.length - 1`	38	67	0	0	75	50	0	83
		Index in array	38	67	0	0	75	0	0	100
Stack	As documented?	No	83	55		0	100	50	75	80
	What is wrong?	Exc. is thrown	67	55		0	50	50	75	80
	What is thrown?	`NegativeArray SizeExc.`	67	55		0	50	50	75	80
	Invariant established?	Yes	67	64		0	50	75	75	80
	What is executable?	Line 33	100	91		50	100	100	100	100
		Line 34	100	82		50	100	75	100	100
	As documented?	No	67	64		0	100	75	50	80
	What is wrong?	Same array	50	18		0	50	0	50	40
		Exc. is thrown	33	64		0	50	75	25	80
	What is thrown?	`NullPointerExc.`	33	64		0	50	75	25	80
	Invariant established?	Yes	83	73		50	50	75	100	80
	What is executable?	Line 44	100	82		50	100	75	100	100
		Line 45	100	82		50	100	75	100	100
	As documented?	Yes	83	82		50	50	75	100	100
	Invariant preserved?	Yes	83	82		50	50	75	100	100
	What is executable?	Line 54	100	91		50	100	100	100	100
		Line 55	100	91		50	100	100	100	100
		Line 58	100	91		50	100	100	100	100
	As documented?	No	33	18		0	50	50	25	0
	What is wrong?	elem. not updated	33	18		0	50	50	25	0
	Invariant preserved?	No	67	36		0	100	50	50	40
	What is wrong?	Elem. not set to null	50	36		0	50	50	50	40
	What is executable?	Line 68	100	82		0	100	100	100	100
		Line 69	100	82		0	100	100	100	100
		Line 72	100	82		0	100	100	100	100
	What is true?	Null retruned	83	73		0	100	100	75	80
		Object returned	100	82		0	100	100	100	100
		Elem. at `size - 1`	83	45		0	50	50	100	60

Source Code	Question	Answer	Java experience: All (18) DCR (%)	All (18) SED (%)	None (3) DCR (%)	None (3) SED (%)	< 2 y. (6) DCR (%)	< 2 y. (6) SED (%)	≥ 2 y. (9) DCR (%)	≥ 2 y. (9) SED (%)
IntegerUtil	As documented?	No	100	50	100	0	100	0	100	80
	What is wrong?	y instead of x	89	50	100	0	100	0	75	80
	What is executable?	Line 13	100	100	100	100	100	100	100	100
		Line 14	100	100	100	100	100	100	100	100
		Line 15	100	100	100	100	100	100	100	100
		Line 18	100	100	100	100	100	100	100	100
		Line 19	100	100	100	100	100	100	100	100
		Line 24	100	100	100	100	100	100	100	100
		Line 25	100	100	100	100	100	100	100	100
		Line 28	100	100	100	100	100	100	100	100
		Line 29	100	100	100	100	100	100	100	100
		Line 33	89	100	50	100	100	100	100	100
	What is true?	Negative	89	75	100	0	100	100	75	80
		0 returned	89	88	100	100	100	100	75	80
		Positive	89	88	100	100	100	100	75	80
		x	100	75	100	100	100	100	100	60
		y	100	75	100	100	100	100	100	60
		z	100	75	100	100	100	100	100	60
MathUtil	As documented?	No	60	83	100		50	100	50	80
	What is wrong?	Exc. is thrown	30	83	50		25	100	25	80
	What is thrown?	ArrayIndex OutOfBoundsExc.	20	83	0		25	100	25	80
	What is executable?	Line 22	90	100	50		100	100	100	100
		Line 23	80	100	50		75	100	100	100
		Line 25	100	100	100		100	100	100	100
		Line 26	90	100	100		75	100	100	100
		Line 28	90	100	100		75	100	100	100
		Line 29	90	100	100		75	100	100	100
		Line 32	100	100	100		100	100	100	100
		Line 33	100	100	100		100	100	100	100
		Line 34	70	100	50		50	100	100	100
		Line 37	90	100	100		75	100	100	100
	What is true?	Negative	50	83	0		50	0	75	100
		0 returned	80	100	50		100	100	75	100
		Positive	80	100	100		75	100	75	100
		In array	70	100	50		75	100	75	100
		Not in array	60	83	0		75	100	75	80
		Winning Tool Count	**44**	**35**	**8**	**3**	**19**	**31**	**18**	**33**

Table 9.24.: Comparison of the Given Expected Answers

Table 9.24 shows that for both classes of participants with Java experience, the total number of correct answers is higher when using the SED than without. In the class without Java experience correct answers are more often selected in a direct code review. Taking into account that only three participants are in that class and that many questions are never answered with both methods, no meaningful conclusions can be drawn. Looking at the winning tool counts of all participants shows that for more questions the correct answer is given in a direct code review. This might be an indicator, that the SED is helpful for specific kinds of questions.

The questions "As documented?" and "Invariant established?" are more often answered correctly in a direct code review, especially in the `Stack` and `ValueSearch` example. In traditional debugging sessions, the explored execution path shows only symptoms of a defect. This is also true for symbolic execution paths as visualized by the SED. To locate the defect in the program logic causing the observed symptom (at runtime) the source code needs to be reviewed. This relation between debugging and reviews is also discussed by Humphrey [71]. However, the SED shows source code and symbolic execution tree at the same time.

The question "What is wrong?" is only answered by participants if they identified before that something is wrong. As participants using the SED often failed to identify a problem, it is not surprising that the correct answers are also more often selected in a direct code review. Interestingly, participants identified more often that an exception is thrown and the correct type of the thrown exception ("What is thrown?") using the SED. Thrown exceptions are explicitly visualized in the SED by special symbolic execution tree nodes.

In addition to the symbolic execution tree, the SED highlights statements reached during symbolic execution. This seems to be helpful as the executable statements ("What is executable?") are more often correctly identified using the SED.

Participants are also asked to select correct statements about the returned value ("What is true?"). Possibly returned symbolic values are visualized by the SED as part of the method return node. In the `MathUtil` and `ValueSearch` example the correct answers are more often selected using SED, whereas in the `BankUtil`, `Stack` and `IntegerUtil` example the direct code review achieved better results.

From Table 9.24 it can be concluded that the SED helps to answer questions about things which are directly part of the symbolic execution tree. According to the given answers, the SED seems not to increase the understanding of the program logic.

The feedback about the SED features mentioned in the introduction video is shown in Table 9.25. Nearly all participants considered the visualized symbolic execution tree and the highlighting of statements reached during symbolic execution as (very) helpful. The views **Properties** and **Variables** were often not used by the participants. The participants using these views considered them mostly as helpful.

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Symbolic Exec. Tree	37	43	18	0	0
Properties view	6	31	12	6	43
Reached Source Code	56	18	6	0	18
Variables View	18	25	25	0	31

(a) All Participants

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Symbolic Exec. Tree	0	0	100	0	0
Properties view	0	0	0	0	100
Reached Source Code	100	0	0	0	0
Variables View	0	0	0	0	100

(b) No Java experience only

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Symbolic Exec. Tree	33	50	16	0	0
Properties view	0	50	0	0	50
Reached Source Code	50	0	0	0	50
Variables View	33	16	33	0	16

(c) $<$ 2 years of Java experience

	Very Helpful (%)	Helpful (%)	Little Helpful (%)	Not Helpful (%)	Never Used (%)
Symbolic Exec. Tree	44	44	11	0	0
Properties view	11	22	22	11	33
Reached Source Code	55	33	11	0	0
Variables View	11	33	22	0	33

(d) $\geq$ 2 years of Java experience

Table 9.25.: Feedback about SED Features

The participants are asked for each reviewed method if the symbolic execution tree or alternatively the concrete execution/debugging was helpful. The results are shown in Section A.2. The symbolic execution tree was mostly considered to be helpful, often as very helpful. Often participants did not write source code. If source code was written, it was often considered as less or not helpful.

The feedback questionnaire offers also the optional opportunity to give feedback as free text. The results are shown in Table 9.26.

ID	Feedback
1	"- The videos should be replaced and instead, slideshows should be used. For me, the videos were often too fast and I had to stop them every 10 seconds or so. - I needed some time to get familiar with Eclipse (only use Atom in my daily work). My impression was that in the 3. and 4. example, I found my way back into Eclipse and could better work with the code. Maybe the level of familiarity with Eclipse should also be asked for at the beginning of the evaluation."
2	"Very interesting kind of debugging. Looking forward to see this "visual" kind of debugging in our daily work. . . "
3	"UX for visualization of state and frame is obviously the key challenge. A bijective coupling of node or path selection to code highlights is a valuable feature which is missing."
4	"Very interesting tool and impressive evaluation! I didn't understand the question about which lines can be executed. Maybe that should be made somewhat clearer; or maybe it's just me."
5	"Nice tool to evaluate and debug source code. But time was to less to use/explore all functions and benefits. "
6	"After using symbolic debugger, you tend to think symbolically even when not using SED."
7	"SED especially useful in seeing undocumented exceptions."

Table 9.26.: Feedback of Participants

Feedback with ID 1 contains comments and suggestions for improvements of the evaluation itself. The Feedback with ID 3 gives hints how to improve the user experience of the visualization. Indeed, it is planned to visualize additional information like the path condition or selected variables as part of each symbolic execution tree node. Also nodes of the current path and the corresponding elements in the source code could be highlighted in a different color. This should help a user to focus on the currently inspected path. The remaining feedback praise the SED without constructive feedback for improvements.

The participants are also asked if they prefer a source code review with or without the SED. The results are shown in Table 9.27. SED is designed to support a review and not to replace it. As this evaluation shows, the SED helps to find out information about feasible execution paths. But studying a symbolic execution tree alone is not sufficient to understand the program logic. This makes it not surprising that 2/3 of the participants would consider the SED depending on the given source code.

		Java experience		
	All (%)	None (%)	< 2 y. (%)	≥ 2 y. (%)
directly	0	0	0	0
directly and using SED, both are equally good	31	0	33	33
directly and using SED, depending on the source code	62	100	50	66
directly and using SED, both are equally bad and should be improved	0	0	0	0
using SED	6	0	16	0

Table 9.27.: Participants Tool Preference

9.3 Evaluation of the Proof Management Optimizations realized by KeY Resources

This section compares the impact of the optimizations proposed in Section 7.3.2 for the reduction of the overall proof time based on KeY Resources (Section 7.4). To this end, a development process on a small Java project is simulated where the source code and its specifications are modified several times.

The starting point is the *PayCard* case study of the *KeY Quicktour*[12]. Initially, the PayCard case study consists of 4 classes, 18 methods and 22 contracts but has no inheritance relations. 25 modifications (see Table 9.28) are performed which add new elements, modify or remove existing ones. By performing these modifications the project grows intermittently to 10 classes, 27 methods, 61 contracts and 6 inheritance relations. Hence, the independent variable are the performed changes. The dependent variable is the overall proof time. The overall proof time is defined as the total time required to perform the verification process without the parsing time for the source code to achieve comparable results not obfuscated by technical issues.

Modification	Description	Classes	Proofs
1	Populate the project with initial classes	4	22
2	Delete a proof file	4	22
3	Delete a meta file	4	22
4	Modify a proof file	4	22
5–8	Modify an initial class	4	22
9	Add two new subclasses	6	34
10	Add two new subclasses	8	49
11	Add two new subclasses	10	61
12	Modify an initial class	10	61
13–18	Modify an added class	10	61
19–21	Modify an initial class	10	61
22	Delete two added classes	8	46
23	Delete four added classes	4	22
24	Remove a method from initial class	4	21
25	Remove a constructor from initial class	4	20

Table 9.28.: Performed Modifications

The system used for the evaluation is powered by an Intel Core i7-2600K CPU, 8 GB RAM and Windows 7 64 Bit. For the Java Virtual Machine the initial heap size is 128 MB and the maximum memory allocation is set to 1024 MB.

In the following, the optimizations selection (Section 9.3.1), proof replay (Section 9.3.2) and parallelization (Section 9.3.3) are evaluated. The combination of all optimizations and the results are presented in Section 9.3.4. Finally, threats to validity are discussed in Section 9.3.5.

9.3.1 Impact of Optimization Selection

Figure 9.15 shows that the simple selection optimization based on changed files reduces the number of proofs in some cases significantly, whereas in others most proofs need to be redone. The reason is that some modifications invalidate almost all proofs whereas others have only a local effect. The initial population (modification 1) requires all proofs to be redone as no previous proofs exist. Starting with modification 2 the optimization was always successful in filtering out some proofs. This is also reflected by the overall proof times shown in Figure 9.16.

[12] www.key-project.org/download/quicktour/quicktour-2.0.zip

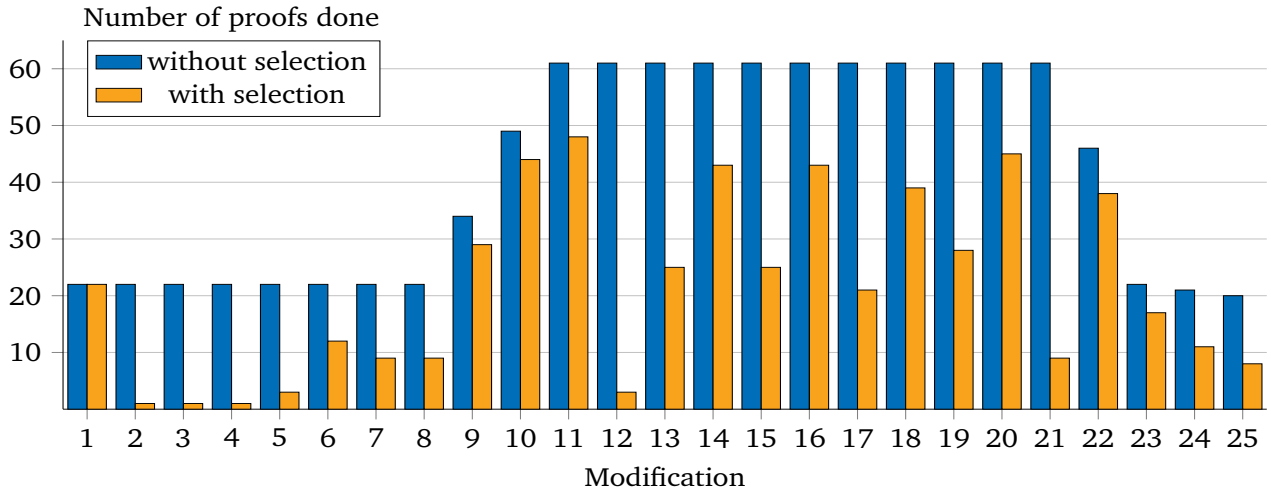


Figure 9.15.: Proof Count of Optimization Selection

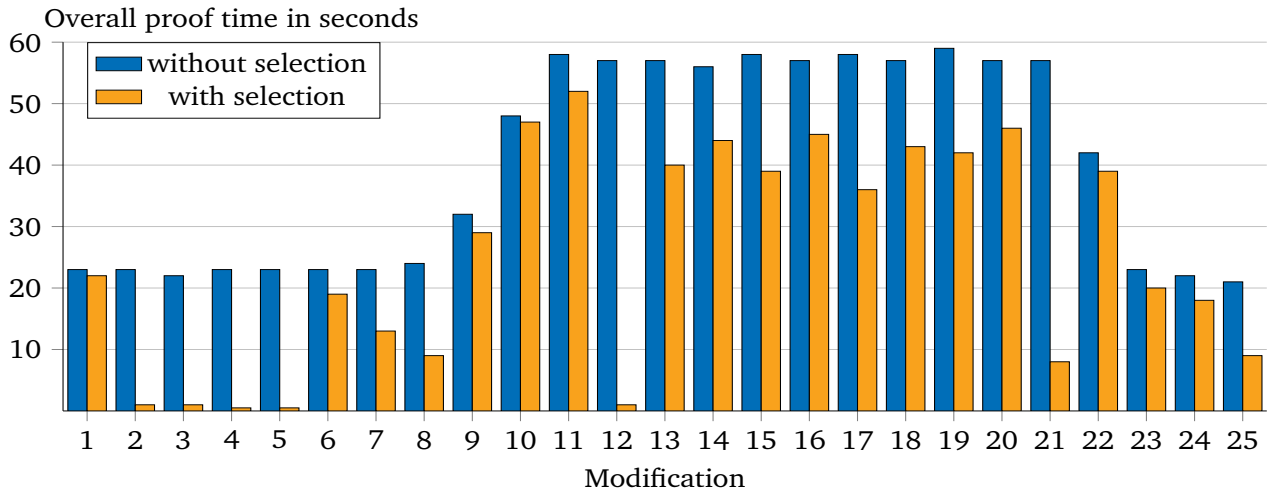


Figure 9.16.: Proof Times of Optimization Selection

As mentioned in Section 7.4.1, the decision whether a proof has to be performed is based in simple cases on changed files and not on the changed content within a file. Even better results are achieved by taking more information about the modification into account.

9.3.2 Impact of Proof Replay

Figure 9.17 compares the overall proof times per modification with and without replay. Parallelization and filtering are not performed. The proof time is almost identical across different modifications as long as the number of performed proofs is the same. The overall proof time with replay is always less than the proof time without proof replay. However, proof replay does not save as much time as one would expect. The reason is most likely that KeY does not switch off all of the proof search infrastructure during a proof replay. Without this limitation a larger improvement is expected.

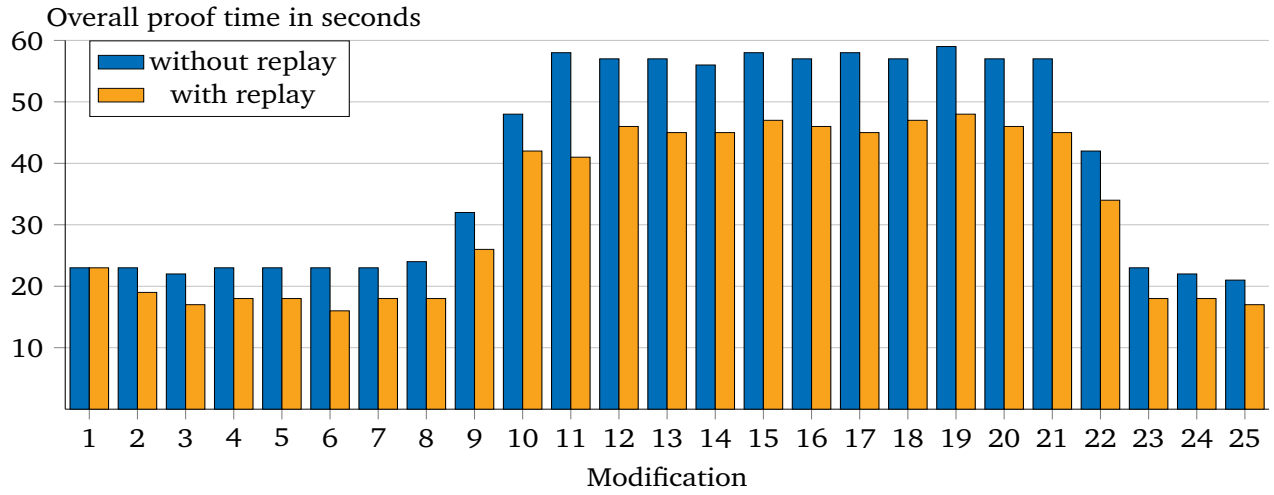


Figure 9.17.: Proof Times of Optimization Proof Replay

9.3.3 Impact of Optimization Parallelization

Figure 9.18 shows the results comparing the use of 1, 2, 4 and 8 threads without replaying and selecting proofs. With a growing number of threads the proof time is reduced until a certain threshold is reached. Beyond that threshold additional threads increase the overall proof time because of an increased synchronization overhead. This overhead is implementation- and hardware-specific and cannot be generalized.

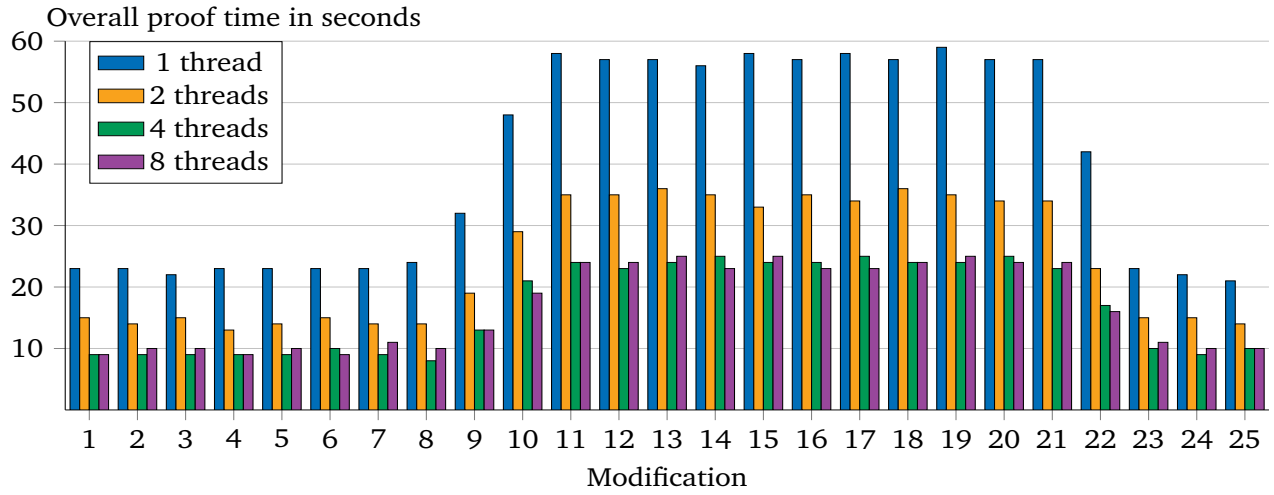


Figure 9.18.: Proof Times of Optimization Parallelization

9.3.4 Combined Optimizations

As described above, each optimization on its own reduces the overall proof time. Now all optimizations are combined such that the number of performed proofs is reduced, replay is used and proofs are performed in parallel. The result is shown in Figure 9.19.

Interestingly, the overall proof time of modifications 2 to 5 and modification 12 is better without multithreading. In these cases almost all proofs are filtered out by the selection optimization. A possible explanation is the synchronization overhead caused by the parallelization optimization.

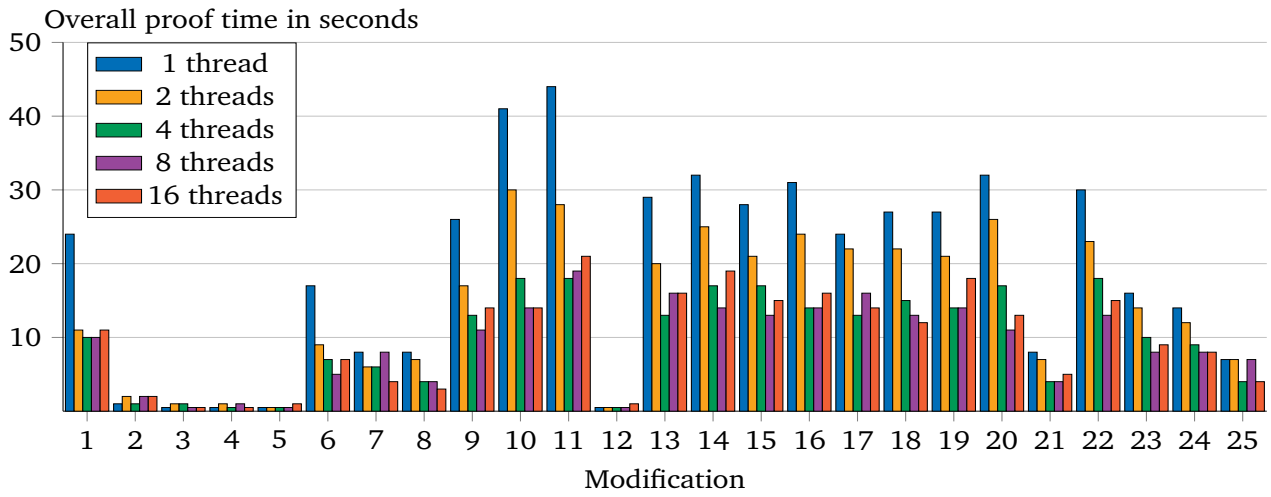


Figure 9.19.: Proof Times of Combined Optimizations

Table 9.29 summarizes the results. The first column specifies which optimizations are used. The next three columns show first the sum (Σ) of all proof times per modification, the resulting average value (∅) and the percentage (%) compared to the worst scenario without optimizations. The last three columns list the total number of proofs done for all modifications, the resulting average value and again a percentage comparison against the worst case.

	Proof time			# of proofs		
Description	Σ in s	∅ in s	%	Σ	∅	%
1 Thread no replay and no selection	1003	40.1	100	1039	41.6	100
1 Thread with no replay and selection	624	25.0	62	534	21.4	51
4 Threads no replay and selection	417	16.7	42	1039	41.6	100
8 Threads no replay and selection	421	16.8	42	1039	41.6	100
1 Thread with replay	803	32.1	80	1039	41.6	100
1 Thread with replay and selection	476	19.0	47	534	21.4	51
4 Threads with replay and selection	245	9.8	24	534	21.4	51
8 Threads with replay and selection	**227**	**9.1**	**23**	**534**	**21.4**	**51**
16 Threads with replay and selection	243	9.7	24	534	21.4	51

Table 9.29.: Analysis of Proof Times and Number of Performed Proofs

Each optimization on its own reduces the overall proof time. The best value is achieved by combining all optimizations and using 8 threads, which reduces the average proof time by 77% from 40.1 seconds to 9.1 seconds.

With the advanced selection optimization, which identifies changed elements within a file, the overall proof time is reduced to 84.9 seconds and the average proof time by 91% to 3.4 seconds.

The achieved improvement is significant but not yet sufficient. A developer may save a file every few seconds triggering the proof process each time. For this reason it is required that the user can continue working while the update process is running. To achieve this, the project builder is only used to trigger the update process executed in a separate job (see Section 7.4.1).

9.3.5 Threats to Validity

The total proof time depends strongly on the number and the complexity of the proofs. The chosen example is well known in the KeY community and covers different verification scenarios. Changes are designed carefully to simulate realistic modifications and to cover all scenarios which can influence the implemented selection optimization.

However, the achieved improvement depends in the end on the source code and its specifications as well as on the used verification tool.

10 Related Work

This chapter discusses work related to this thesis. First, Section 10.1 discusses topics related to symbolic execution using specifications. Second, work related to the symbolic execution engine (Section 10.2), truth status tracing (Section 10.3) and slicing is discussed (Section 10.4). Third, Section 10.5 discusses work related to the SED. Fourth, Section 10.6 discusses work related to the proof management. Fifth, work related to the evaluation is discussed in Section 10.7. Finally, work related to JML Editing is discussed in Section 10.8 and Section 10.9 discusses work related to the KeYIDE.

10.1 Related to Symbolic Execution using Specifications

Compositional symbolic execution [128] employs *method summaries* that are obtained by symbolically executing each method separately and maintaining information about the explored paths in the form of path conditions (which constrain the input values) and a symbolic representation of the final state of each method. The concept of method summaries is compatible with symbolic execution using contracts and could easily be integrated into it.

Using method contracts instead of method summaries has the advantage to achieve more robustness in the presence of changes to implementations of called methods. The key point is that it completely separates the contract of a method from its implementation. This means that symbolic execution does not rely on the availability of an actual implementation. Hence the approach can be used in early prototyping or evaluation phases as well as in situations where a third party library implementation is not available. In the context of test generation, this allows one to generate acceptance tests for an externally developed library. Another consequence of the separation of contracts and method implementations is that a correct symbolic execution tree using method contracts remains correct for any implementation that satisfies the used contracts. Specifically, this holds for overridden methods as long as the overriding method satisfies Liskov's behavioral subtyping principle [99]. Further, the additional abstraction provided by a declarative contract nurtures program understanding as it focuses on *what* is computed instead of *how* something is computed.

Godefroid [57] uses symbolic execution for dynamic test generation and proposes to use loop summaries to deal with unbounded loops. The computation of loop summaries is restricted to certain types of loops to be able to compute the input-output dependencies. The presented approach where loop specifications are applied during symbolic execution is agnostic with respect to the actual loop invariant generation method and not limited to a specific loop inference approach. Even though the approach is presented in a static symbolic execution setting, it is expected to be applicable also to dynamic symbolic execution strategies as presented in [40, 78].

Behavior trees [45] are an abstract visual notation to specify the behavior of software systems. They are derived from a detailed requirements analysis rather than from source code and they do not represent symbolic states.

10.2 Related to Symbolic Execution Engine

Several tools based on symbolic execution exist. Often, the symbolic execution engine is part of the tool and not available as a stand-alone application or API. Some examples are TRACER [77], KIV [110] or VeriFast [74] in the context of verification, Bogor [113] in the context of model checking and Pex [123], SAGE [59] or DART [58] in the context of test case generation. Other tools like JPF–SE [10], XRT [62], Silicon [81], KLEE [30], Otter [112], EXE [29] or Kudzu [114] are specifically developed to perform symbolic execution, possibly beside some additional tasks. Whereas these tools collect path conditions,

check constraints or generate concrete input values, KeY's Symbolic Execution Engine offers the full explored symbolic execution tree without a specific use case in mind.

Another difference is that especially symbolic execution engines used for test case generation are optimized to deal with a large code base whereas KeY's Symbolic Execution Engine can apply auxiliary specifications (method contracts and loop specifications) to ensure finite symbolic execution trees.

In the context of code generation, PE-KeY [79] uses similarly to KeY's Symbolic Execution Engine an uninterpreted predicate to separate feasible (open goals) from infeasible (closed proof branches) execution paths.

The prototypic symbolic state debugger by Hähnle et al. [63] also generates a symbolic execution tree from a proof tree. However, the approaches are completely different. Hähnle et al. [63] instrument the source code and use additional calculus rules in the proof to deal with the instrumentation. This comes with several limitations: (i) the original source code is not analyzed, (ii) a special strategy is needed to ensure that the additional rules are applied, (iii) specifications are not supported, (iv) symbolic execution tree generation from arbitrary verification proofs is not possible. KeY's Symbolic Execution Engine avoids these limitations by the use of metadata which makes it possible to support the Java DL calculus as it is. The design of KeY's Symbolic Execution Engine allows one with relative ease to support arbitrary proofs from verification attempts as discussed in Section 4.2. The computation of memory layouts is based on the same idea, namely to compute equivalence classes by systematic cuts. Nevertheless, the instantiation of the cut formulas as well as the initial proof obligation are different.

10.3 Related to Truth Status Tracing

Whyline [88] is a debugging tool which allows the user to ask questions in the form of *why something did happen* or *why something did not happen*. The tool shows as answer the relevant states of the execution history. The facility to let the user ask questions can also be applied to the results of the truth status tracing presented in this thesis. Possible questions are (i) why does a formula evaluate to true or false and (ii) why is a formula not evaluated. For question (i) the answer consists of all involved proof tree nodes. The mapping as result of Algorithm 5.2 can easily be extended to store the proof tree node as well. The involved proof tree nodes are thus all nodes of all mapping entries considered by Algorithm 5.3. For more precise results, all nodes can be included at which a rewrite rule is applied to a formula considered by Algorithm 5.3. The answer of question (ii) are all enclosing formulas and their truth statuses.

Dafny [94], Spec# [15] and Boogie [16] place in case of a failed verification attempt an error marker in the source code as close as possible to the failed proof obligation. Future work includes to maintain position information of specifications in KeY to realize such a precise placement of markers. However, the mentioned systems do not allow to trace why a (complex) formula could not be proven.

10.4 Related to Slicing

Several methods to compute slices exist, consider for instance [124]. They are not applied to proof trees or symbolic execution trees. In this thesis the concept of program slicing is generalized to proof trees and symbolic execution trees. An implementation of thin slicing [118] is presented as well. Taking only one path and a specific memory layout into account, the computed slice is identical to a dynamically computed slice with concrete values following that path. A comparison to static and dynamic slicing in general needs further investigations.

Asavoae et al. [12] present an interprocedural slicing realized in the Maude [33] system. It has in common with the slicing of a proof tree presented in this thesis that it works with terms (and formulas). The difference is that it executes a Maude program to compute the slice whereas the presented proof tree slicing extracts the slice from an existing proof (which was not performed for the purpose of slicing).

Agrawal et al. [2] present a debugging model based on dynamic program slicing and execution backtracking techniques. It is used in a tool called *SPYDER*. The SED allows the user to slice a symbolic

execution tree in a similar way and consequently supports the debug model presented by Agrawal et al. [2].

CodeSurfer [11] offers functionality useful for program inspection, like the highlighting of the source code corresponding to the slice. Similarly, the SED highlights the symbolic execution tree nodes corresponding to the slice and could easily be extended to color the related source code. In addition, CodeSurfer allows the user to compare different slices which would also be helpful in the SED.

Whyline [88] and its version for Java [89] uses the execution history and slicing techniques to allow the user to ask questions in the form of *why something did happen* or *why something did not happen*. The tool shows as answer the relevant states of the execution history. Applying this style of debugging to symbolic execution trees will in addition offer the opportunity to ask under which condition something does happen or does not happen.

10.5 Related to the Symbolic Execution Debugger

EFFIGY [85] was the first system that allowed to interactively execute a program symbolically in the context of debugging. It did not support specifications or visualization.

A number of recent tools implement symbolic execution for program verification [76] or test generation [9, 123], which are complementary to SED. In fact, SED could be employed to control or visualize these tools.

The Boogie Verification Debugger [91] and the symbolic debugger part of the VeriFast IDE [75] can be used to inspect the states on a path which is not verified. Like the SED, these tools present the memory and assumptions of each selected state to the user. With KeY's SED integration, all execution paths explored by a proof can be inspected at once. In addition, the truth status tracing can be used to determine the not yet verified part of the proof obligation.

The Eclipse plug-in of Java PathFinder (JPF) [109] prints the analysis results obtained from symbolic execution as a text report, but does neither provide visualization nor interactive control of symbolic execution. JPF–SE [10] is prototypically supported by SED as an alternative symbolic execution engine.

The symbolic execution engine and its Eclipse integration described by Ibing [72] features non-interactive graphic visualization of the symbolic execution tree. SED allows the user to interact with the visualization as a means to control symbolic execution and to inspect symbolic states.

A prototypic symbolic state debugger that could not make use of method contracts and loop specifications was presented by Hähnle et al. [63]. However, that tool was not very stable and its architecture was tightly integrated into the KeY system. As a consequence, the SED was developed from scratch as a completely new application featuring significant extended and new functionality. It is realized as a reusable Eclipse extension which allows one to integrate different symbolic execution engines, see also Section 10.2.

In case that the assumed precondition specifies exactly one execution path, the resulting symbolic execution tree is the execution history as it would be shown by an omniscient or reverse debugger such as ODB [98] or TOD [107]. Using symbolic execution, the challenge to keep history of a concrete execution is avoided, because (i) execution can start at any point, (ii) states consist only of locations written during symbolic execution and (iii) states are eagerly simplified. In addition, the symbolic debug model can be instantiated to realize an omniscient debugger. To support multi-threading, the implementation needs to model the interleaving of threads. In such a case, the visualization of the symbolic execution tree would show a list structure for each thread. For efficiency, a table control could be used instead where each column represents a thread and each row the program state of the currently active thread. Additional columns might be added to show the values of user defined locations.

There is clearly a relation between symbolic execution trees and code representations used traditionally in static analyses, such as control flow graphs and program dependence graphs [105]. A main difference is that the latter do not contain symbolic state information (except indirectly, in the form of path conditions).

10.6 Related to Proof Management

Mossakowski et al. [103] suggest development graphs for proof management. Development graphs constitute a more general structure than proof dependencies, but are tailored to algebraic specification languages such as CASL. Development graphs capture dependencies among specifications and can be used to compute the effect of a specification change. Proofs seem not to be analyzed which might result in more proofs being redone than necessary. Instantiating development graphs to a design-by-contract setting for a real-world programming language is not straightforward concerning the representation of source code. Development graphs are implemented in the tool Maya [13], which seems not be integrated with a mainstream IDE. Proof dependencies provide a lightweight approach that can be implemented for most combinations of verification system and IDE.

SPEEDY [37] and OpenJML with ESC [36] provide an integration into an IDE. To the best of the author's knowledge they do neither manage proofs nor use change information to restrict the amount of verification effort. In case of interactive or semi-automatic verification tools the user is also interested in the proofs and to maintain them along the source code.

Boogie [16] is an intermediate verification language used for instance by Dafny [94], Spec# [15] or VCC [34] to express proof obligations. The identically named verification engine can be used to verify different versions of a program. To reduce the overall proof time, parallelization [97] is applied as well. For each program entity, verification results are cached and only changed or by a change affected entities are re-verified. Changed program entities are identified with help of an entity checksum computed by the client using Boogie. In addition, Boogie computes for each program entity a dependency checksum based on its entity checksum and the dependency checksums of entities it directly depends on. A dependency checksum is used to identify program entities affected by a change. Checksums are used to define the order in which program entities are verified which fits with the presented prioritization optimization. With caching, verification effort is reduced similarly as the presented selection optimization based on proof dependencies. However, proof dependencies allow one to model different kinds of dependencies in case the affected elements need to be computed differently. A fine-grained caching [96] can be used in addition to reuse verification results for some parts of an implementation. The presented proof management can be realized without changing the used verification tool. The use of a modified proof replay strategy in KeY Resources which explores only execution paths affected by a change will, similarly to the fine-grained caching [96], reduce the number of applied rules and consequently improve the performance.

Rodin is the main IDE used for modeling in Event-B [1]. It features a semi-automatic verification system for Event-B models and stores proofs within the project similarly to the presented proof management concept. The detection of invalidated proofs is based on the name of proof obligations which change when the model is changed. Invalid proofs are neither redone nor removed automatically.

Other static analyses like FindBugs [69] use also common IDE concepts to organize the work and to present results, but do not manage their results within the IDE and have to be run again from scratch when the source code is modified.

10.7 Related to Evaluation

Beckert et al. [23] evaluate the usability of KeY [22] and Isabelle [104] in focus groups. The goal is to improve the usability of the tools and not a direct comparison of them. Also mock-ups of new features are evaluated. One of them is a window which shows the ancestors of a formula on the path up to the initial proof obligation. Such a tracing of formulas is done as part of the truth status tracing (Section 5.1) and should allow one to implement the functionality presented by Beckert et al. [23] as part of the KeY GUI. In addition, some participants asked for a feature to show the path condition related to an open goal. Path condition computation is already part of KeY's Symbolic Execution Engine and also supported by the SED.

Usability investigations were made for several interactive theorem provers, e.g. [82, 8, 7, 6, 73, 31]. To the best of the author's knowledge, none of them evaluated achieved improvements or compared different user interfaces.

10.8 Related to JML Editing

EditBox[1] is an Eclipse extension which highlights the background of code blocks in Java. This functionality is also realized by replacing the source viewer configuration at runtime.

SPEEDY [37] offers features to edit specifications of C programs similar to JML Editing. They are realized without runtime replacements because the C/C++ Development Tooling (CDT)[2] allows one to extend the C syntax.

10.9 Related to KeYIDE

The KeYIDE is an alternative user interface for verification with KeY deeply integrated into Eclipse. Instead of building the user interface from scratch, Proof General/Eclipse [134] could have been used. This is a generic user interface for interactive proofs. The decision against it is based on the fact that Proof General/Eclipse offers functionality to parse and edit documents like a proof script. However, the proving style in Java DL is different because rules are applied on existing sequents. Consequently, user input in form of free text is only required in rare cases when an instantiation is missing.

[1] editbox.sourceforge.net
[2] eclipse.org/cdt

11 Conclusion And Future Work

This chapter concludes the thesis and offers an outlook on future work. It is structured following the title of this thesis in symbolic execution (Section 11.1), debugging (Section 11.2) and verification (Section 11.3).

11.1 Symbolic Execution

Symbolic execution is extended by the use of specifications to ensure that the resulting symbolic execution trees are finite even in presence of loops and recursive methods. The use of method contracts also makes execution more robust against implementation changes and allows symbolic execution even if the concrete implementation is not (yet) available.

Both, KeY's Symbolic Execution Engine and the Symbolic Execution Debugger (SED) support the application of JML specifications during symbolic execution. This means that any contract or loop invariant generation tool which outputs JML can be used to annotate the analyzed programs. The approach to symbolic execution with specifications could be implemented also for other languages than Java/JML, for example, Code Contracts[1].

An application scenario planned to be investigated in the future is to extend the test generation in KeY [46] to symbolic execution trees with specifications. One problem to be solved is that path conditions are no longer necessarily quantifier-free formulas which might impair the computation of test input values.

It would be interesting to formalize the relation between symbolic execution with specifications, control flow graphs, and program dependence graphs. As the underlying construction algorithms are rather different, it is likely that synergies can be gained from their combination.

11.2 Debugging

Recent years witnessed a renewed dynamics in research devoted to debugging. To a considerable degree this is based on breakthroughs in static analysis of software, see Ayewah et al. [14]. The book by Zeller [137] presents a systematic approach to debugging and an overview of currently developed and researched debugging techniques.

The Symbolic Execution Debugger (SED) is the first debugging tool that is (i) based on symbolic execution and first-order automated deduction, (ii) visualizes complex control and data structures, including reference types, (iii) can render unbounded loops and method calls with the help of specifications, and (iv) is seamlessly integrated into a mainstream IDE (Eclipse). Other tools have capabilities (ii) and (iv), but to the best of the author's knowledge, the SED is the first tool to realize (i) and (iii). A prototype of the SED was presented by Hähnle et al. [63], however, it lacked (iii).

The SED is not a single tool, it is an extension of the Eclipse debug platform for interactive symbolic execution which allows one to integrate any symbolic execution engine and to present results of the analysis using it. By default, the SED comes with KeY as symbolic execution engine and supports also verification with KeY. In addition, an experimental integration of JPF–SE [10] exists.

In order to implement KeY's SED integration, a pure Java API for symbolic execution with KeY was developed. It is called KeY's Symbolic Execution Engine and is already used in several projects (see Section 4.12).

An experimental evaluation that was performed shows that the SED is a powerful assistant in code reviews. Using the SED more defects are found with statistical relevance and often in less time. Future

[1] research.microsoft.com/en-us/projects/contracts

work includes the joining of execution paths to achieve a more compact representation closer to the control flow and thus the source code. In addition, symbolic execution tree nodes forming a common bug pattern can be directly highlighted.

For using the SED in real world applications, it is necessary to increase the coverage of Java beyond what is currently supported by the KeY verifier. The most important gaps are floating point types and concurrent programs. Both areas constitute open research problems for formal verification, however, it is not at all unrealistic to implement support of these features in the context of debugging. The reason is that for debugging purposes often an approximation of the program semantics is already useful. For example, floating point types might be approximated by fixed point representations or by confidence intervals, whereas symbolic execution of multi-threaded Java would simply concentrate on the thread from which execution is started.

Another experimental evaluation that was performed compares the inspection of proof attempts using SED and KeY. Participants identified the reason why a proof attempt is still open with statistical relevance more often using the SED. This shows that the user interface of the SED which focuses on the source code and hides nearly all the logical reasoning helps to flatten the learning curve to use a verification system. Especially in a fully automatic setting, the user can now inspect a failed proof attempt and identify the problem instead of guessing which modification of source code or specifications might fix it.

Already now it is possible to use the SED for formal verification (see Section 6.6), but it lacks the possibility to switch into interactive mode when KeY's proof strategy was not powerful enough to close a goal automatically. In future work, KeY's SED integration will be extended for interactive verification resulting in a full-fledged alternative GUI for the KeY verification system. The advantages are obvious: an SED-like interface for the KeY verifier will inherit properties (ii) and (iv) from above. The resulting tool will not only be attractive to software developers unfamiliar with formal methods, but it will also constitute a continuous transition from the world of software developers into the world of formal verification.

On the other hand, exploiting verification results during symbolic execution allows one to classify execution paths automatically as correct or wrong. In future work this will be used to apply algorithmic debugging [117] to symbolic execution trees in a semi-automatic fashion.

11.3 Verification

A lightweight approach to integrate an interactive verification tool into an IDE is presented. Its implementation is called KeY Resources which integrates KeY into Eclipse. The integration achieves that source code, specifications and proofs are always in sync without placing that responsibility on the user. Proof results are presented as early as possible and user interaction is only required when a proof cannot be closed automatically. Several optimizations can be added in a modular manner. With them it was possible to reduce the overall proof time on average by 91% in the presented case study.

All presented optimizations have in common that they can be realized without modifying the verification tool. With future improvements of KeY, it is planned to increase the performance of KeY Resources and at the same time to preserve more interactively performed proof steps. Promising candidates are proof reuse [86], abstract contracts [64] and a strategy focussing on branches affected by a change similar to fine-grained caching [96]. In addition, proof dependencies will be formalized to obtain a formal proof of the correctness of the presented approach.

In state-of-art software development nearly all tasks are done within an IDE to achieve a consistent software development process. Verification should aim to become part of that process and integrate seamlessly into the existing infrastructure. The presented approach is a step toward this goal and can be realized without any changes to the verification tool.

For an optimal integration of KeY into Eclipse supporting the full verification process, additional tools are realized (see Chapter 8). This includes (i) editing facilities for JML, (ii) a generator for stubs needed by KeY if no source code is present for some classes (e.g. used libraries) and (iii) an alternative implemen-

tation of the original user interface of KeY deeply integrated into Eclipse. A highlight of the JML Editing is, that it has no dependencies to KeY and supports different, user editable, JML dialects. In future work, editing facilities for taclets and the language of *key* and *proof* files will be implemented.

Beyond the classical verification use case, KeY's Eclipse integration altogether can be easily adapted to support also other use cases like information flow security [116, 115, 26, 126]. All that needs to be done is to adjust KeY's Symbolic Execution Engine and the detection of proof dependencies to support the additionally introduced calculus rules. Already now, counterexample generation and test case generation [46, 19, 56] is available in Eclipse (KeY Resources and KeYIDE) and adapted for the needs when used and integrated into an IDE. In future work, both will be also part of KeY's Symbolic Execution Engine. Further, it is desirable to integrate the missing parts of the KeY Framework like code transformation [79] and generation of loop invariants [130].

Furthermore, KeY's Eclipse integration can be the basis for all tools using JML. It is planned for instance to use the StaRVOOrS [32] technology to generate optimized JML specifications for runtime assertion checking for not statically verified proof obligations, e.g. with OpenJML [36]. In this context, also other test case generation tools can be integrated. This will result in a tool suite in which as much as possible is automatically verified. If something is not automatically verifiable, the user can put effort in interactively finishing the proof (e.g. with the SED) or increase her trust in the correctness with help of the generated test cases and runtime assertions.

A Appendix

A.1 Source Code of the Understanding Proof Attempts Evaluation

A.1.1 Class Account

```
1  public class Account {
2     private int balance;
3
4     /*@ normal_behavior
5       @ requires amount > 0;
6       @ ensures balance == \old(balance) - amount;
7       @ ensures \result == amount;
8       @ assignable balance;
9       @*/
10    public int checkAndWithdraw(int amount) {
11       if (canWithdraw(amount)) {
12          withdraw(amount);
13          return amount; //XXX: Termination 1
14       }
15       else {
16          return 0; //XXX: Termination 2
17       }
18    }
19
20    /*@ normal_behavior
21      @ requires amount > 0;
22      @ ensures balance == \old(balance) - amount;
23      @ assignable balance;
24      @*/
25    public void withdraw(int amount) {
26       balance -= amount;
27    }
28
29    /*@ normal_behavior
30      @ requires amount > 0;
31      @ ensures true;
32      @ assignable \nothing;
33      @*/
34    public boolean canWithdraw(int amount) {
35       return amount > 0;
36    }
37
38    /*@ normal_behavior
39      @ requires true;
40      @ ensures \result == balance;
41      @ assignable \nothing;
42      @*/
43    public int getBalance() {
44       return balance;
45    }
```

```
}
```

Listing A.1: Class `Account`

A.1.2 Class `Calendar`

```
public class Calendar {
   protected /*@ non_null @*/ Entry[] entries = new Entry[8];

   /*@ invariant entrySize >= 0 && entrySize < entries.length;
     @*/
   protected int entrySize = 0;

   /*@ normal_behavior
     @ ensures entries[\old(entrySize)] == entry;
     @ ensures entrySize == \old(entrySize) + 1;
     @ assignable entries, entries[entrySize], entrySize;
     @*/
   public void addEntry(/*@ non_null @*/ Entry entry) {
      if (entrySize == entries.length) {
         Entry[] newEntries = new Entry[entries.length * 2];
         /*@ loop_invariant i >= 0 && i <= entries.length;
           @ loop_invariant (\forall int j; j >= 0 && j < i;
           @                         newEntries[j] == entries[j]);
           @ decreasing entries.length - i;
           @ assignable newEntries[*], i;
           @*/
         for (int i = 0; i < entries.length; i++) {
            newEntries[i] = entries[i];
            //XXX: Loop Body Termination (of the 'Body Preserves Invariant' branch)
         }
         entries = newEntries;
         //XXX: Continuation After Then
      }
      else {
         //XXX: Continuation After Else
      }
      entries[entrySize] = entry;
      entrySize++;
   }

   public static class Entry {
   }
}
```

Listing A.2: Class `Calendar`

A.1.3 Class `ArrayUtil`

```
public class ArrayUtil {
   /*@ normal_behavior
     @ ensures array == null || array.length == 0 ==> \result == -1;
     @ ensures array != null && array.length >= 1 ==> (\forall int i; i >= 0 && i < array.length;
    array[\result] <= array[i]);
     @ assignable \nothing;
```

```
6       @*/
7     public static int minIndex(/*@ nullable @*/ int[] array) {
8       if (array != null) {
9         if (array.length == 0) {
10          return -1; //XXX: Termination 1
11        }
12        else {
13          if (array.length == 1) {
14            return array[0]; //XXX: Termination 2
15          }
16          else {
17            int minIndex = 0;
18            /*@ loop_invariant i >= 1 && i <= array.length;
19              @ loop_invariant minIndex >= 0 && minIndex < i;
20              @ loop_invariant (\forall int j; j >= 0 && j < i;
21              @                           array[minIndex] <= array[j]);
22              @ decreases array.length - i;
23              @ assignable minIndex, i;
24              @*/
25            for (int i = 1; i < array.length; i++) {
26              if (array[i] < array[minIndex]) {
27                minIndex = 1;
28                //XXX: Loop Body Termination  1 (of the 'Body Preserves Invariant' branch)
29              }
30              else {
31                //XXX: Loop Body Termination 2 (of the 'Body Preserves Invariant' branch)
32              }
33            }
34            return minIndex; //XXX: Termination 3
35          }
36        }
37      }
38      else {
39        return -1; //XXX: Termination 4
40      }
41    }
42  }
```

Listing A.3: Class `ArrayUtil`

A.1.4 Class `MyInteger`

```
1  public class MyInteger {
2    public int value;
3
4    /*@ normal_behavior
5      @ ensures value == \old(value) + summand.value;
6      @ assignable value;
7      @*/
8    public void add(/*@ non_null @*/ MyInteger summand) {
9      value += summand.value;
10   }
11 }
```

Listing A.4: Class `MyInteger`

A.2 Source Code of the Reviewing Code Evaluation

A.2.1 `BankUtil` Code Example

```
/**
 * This class provides utility methods for a banking application.
 */
public class BankUtil {
   /**
    * Computes the insurance rate based on the given age.
    * @param age The requested age.
    * @return The insurance rate according to the age:
    * <ul>
    *    <li>{@code 200} if {@code age < 18}</li>
    *    <li>{@code 250} if {@code age >= 18} and {@code age < 19}</li>
    *    <li>{@code 300} if {@code age >= 19} and {@code age < 21}</li>
    *    <li>{@code 450} if {@code age >= 21} and {@code age < 35}</li>
    *    <li>{@code 575} if {@code age >= 35}</li>
    * <ul>
    */
   public static long computeInsuranceRate(int age) {
      int[] ageLimits = {18, 19, 21, 35, 65};
      long[] insuranceRates = {200, 250, 300, 450, 575};
      int ageLevel = 0;
      long insuranceRate = 570;
      while (ageLevel < ageLimits.length - 1) {
         if (age < ageLimits[ageLevel]) {
            return insuranceRates[ageLevel];
         }
         ageLevel++;
      }
      return insuranceRate;
   }
}
```

Listing A.5: Class `BankUtil`

	All (14)		None (2)		< 2 y. (3)	≥ 2 y. (9)	
	DCR (7)	SED (7)	DCR (1)	SED (1)	DCR (0)	SED (3)	DCR (6)	SED (3)
Yes, Very helpful	29 %	86 %	0 %	0 %	100 %	33 %	100 %	
Yes, Helpful	14 %	14 %	100 %	100 %	0 %	0 %	0 %	
Yes, Little helpful	0 %	0 %	0 %	0 %	0 %	0 %	0 %	
No, Not helpful	14 %	0 %	0 %	0 %	0 %	17 %	0 %	
Not considered	43 %	0 %	0 %	0 %	0 %	50 %	0 %	

Table A.1.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution

A.2.2 `IntegerUtil` Code Example

```
1  /**
2   * This class provides general utility methods dealing with integer numbers.
3   */
4  public class IntegerUtil {
5     /**
6      * Returns the middle value of {@code x}, {@code y} and {@code z}.
7      * @param x The first integer number.
8      * @param y The second integer number.
9      * @param z The third integer number.
10     * @return The middle value of {@code x}, {@code y} and {@code z}.
11     */
12    public static int middle(int x, int y, int z) {
13       if (y < z) {
14          if (x < y) {
15             return y;
16          }
17          else {
18             if (x < z) {
19                return y;
20             }
21          }
22       }
23       else {
24          if (x > y) {
25             return y;
26          }
27          else {
28             if (x > z) {
29                return x;
30             }
31          }
32       }
33       return z;
34    }
35 }
```

Listing A.6: Class `IntegerUtil`

	All (17)		None (3)		< 2 y. (5)		≥ 2 y. (9)	
	DCR (9)	SED (8)	DCR (2)	SED (1)	DCR (3)	SED (2)	DCR (4)	SED (5)
Yes, Very helpful	0 %	38 %	0 %	0 %	0 %	50 %	0 %	40 %
Yes, Helpful	0 %	12 %	0 %	0 %	0 %	0 %	0 %	20 %
Yes, Little helpful	11 %	50 %	50 %	100 %	0 %	50 %	0 %	40 %
No, Not helpful	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %
Not considered	89 %	0 %	50 %	0 %	100 %	0 %	100 %	0 %

Table A.2.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution

A.2.3 `MathUtil` Code Example

```
/**
 * Provides static methods performing mathematically operations.
 */
public class MathUtil {
   /**
    * Computes the median value between the given start and end index of a
    * sorted array without modifying it. The relation between start and end
    * is not important. This means that the median is computed in case that
    * {@code start <= end} holds but also in case that {@code start > end} holds.
    * <p>
    * In case that the number of array elements between start and end is odd (uneven),
    * the median is the value of the array element in the middle of start and end.
    * <p>
    * In case that the number of array elements between start and end is even,
    * the median is the average (integer division) of the two middle elements.
    * @param array The sorted array for which to compute the median.
    * @param start A valid index in the array.
    * @param end A valid index in the array.
    * @return The median value of the array between start and end index.
    * @throws IllegalArgumentException in case of illegal parameters.
    */
   public static int median(int[] array, int start, int end) {
      // Check parameters
      if (array == null) {
         throw new IllegalArgumentException("Array is null.");
      }
      if (start < 0 || start >= array.length) {
         throw new IllegalArgumentException("Start is not within the array bounds.");
      }
      if (end < 0 || end >= array.length) {
         throw new IllegalArgumentException("End is not within the array bounds.");
      }
      // Compute median
      int middle = (start + end) / 2;
      if ((start + end) % 2 == 0) {
         return array[middle];
      }
      else {
         return (array[middle] + array[middle + 1]) / 2;
      }
   }
}
```

Listing A.7: Class `MathUtil`

	All (16)		None (2)		< 2 y. (5)		≥ 2 y. (9)	
	DCR (10)	SED (6)	DCR (2)	SED (0)	DCR (4)	SED (1)	DCR (4)	SED (5)
Yes, Very helpful	10 %	17 %	0 %		0 %	0 %	25 %	20 %
Yes, Helpful	60 %	67 %	50 %		50 %	100 %	75 %	60 %
Yes, Little helpful	10 %	17 %	0 %		25 %	0 %	0 %	20 %
No, Not helpful	0 %	0 %	0 %		0 %	0 %	0 %	0 %
Not considered	20 %	0 %	50 %		25 %	0 %	0 %	0 %

Table A.3.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution

A.2.4 `ValueSearch` Code Example

```
1  /**
2   * Provides a linear search to find a given value.
3   * @see AbstractSearch
4   */
5  public class ValueSearch extends AbstractSearch {
6     /**
7      * The value to search.
8      */
9     private int value;
10
11    /**
12     * Performs a linear search to find the first array index
13     * containing the given value. The array is not modified by the search.
14     * @param array The array in which the search is performed.
15     * @param value The value to search (search criteria).
16     * @return The index of the first found element or
17     *         {@code -1} if no element was found.
18     */
19    public static int find(int[] array, int value) {
20       return new ValueSearch().search(array);
21    }
22
23    /**
24     * Accepts an array index if it's value is equal to {@link #value}.
25     * @param array The array in which the search is performed.
26     * @param index The current array index to check.
27     * @return {@code true} value at {@code array[index]} is equal to {@link #value}, {@code false}
       otherwise.
28     */
29    protected boolean accept(int[] array, int index) {
30       if (index < 0 || index >= array.length) {
31          return false;
32       }
33       else {
34          return array[index] == value;
35       }
36    }
```

```
}
```

Listing A.8: Class `ValueSearch`

```
/**
 * Provides the basic functionality to perform a linear search.
 * @see ValueSearch
 */
public abstract class AbstractSearch {
   /**
    * Performs a linear search without modifying the given array.
    * @param array The array in which the search is performed.
    * @return The index of the first found element or
    *         {@code -1} if no element was found.
    */
   protected int search(int[] array) {
      /*@ loop_invariant i >= 0 && i <= array.length;
        @ decreasing array.length - i;
        @ assignable i;
        @*/
      for (int i = 0; i < array.length; i++) {
         if (accept(array, i)) {
            return i;
         }
      }
      return -1;
   }

   /**
    * Checks whether the specified array location matches the search criteria.
    * @param array The array in which the search is performed.
    * @param index The current array index to check.
    * @return {@code true} location matches search criteria, {@code false} otherwise.
    */
   protected abstract boolean accept(int[] array, int index);
}
```

Listing A.9: Class `AbstractSearch`

	All (16)		None (1)		< 2 y. (6)		≥ 2 y. (9)	
	DCR (8)	SED (8)	DCR (1)	SED (0)	DCR (4)	SED (2)	DCR (3)	SED (6)
Yes, Very helpful	0 %	12 %	0 %		0 %	0 %	0 %	17 %
Yes, Helpful	12 %	62 %	0 %		0 %	50 %	33 %	67 %
Yes, Little helpful	62 %	12 %	100 %		75 %	50 %	33 %	0 %
No, Not helpful	0 %	12 %	0 %		0 %	0 %	0 %	17 %
Not considered	25 %	0 %	0 %		25 %	0 %	33 %	0 %

Table A.4.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution

A.2.5 ObservableArray Code Example

```
/**
 * An observable array which delegates all actions to an array.
 * @see ArrayListener
 * @see ArrayEvent
 */
public class ObservableArray {
   /**
    * The array to which all actions are delegated.
    * <p>
    * The array is never {@code null},
    * but the value at an array index might be {@code null}.
    */
   /*@ invariant array != null;
     @*/
   private final /*@ nullable @*/ Object[] array;

   /**
    * The optional available {@link ArrayListener} instances.
    * <p>
    * If no listeners are available, array {@link arrayListeners} might be {@code null} or empty.
    * Also the value at an array index might be {@code null}.
    */
   private /*@ nullable @*/ ArrayListener[] arrayListeners;

   /**
    * Constructs an {@link ObservableArray} taking ownership of the given array.
    * <p>
    * In case that the given array is {@code null} an
    * {@link IllegalArgumentException} will be thrown.
    * @param array The array to which all actions are delegated.
    * @throws IllegalArgumentException if the given array is {@code null}.
    */
   public ObservableArray(Object[] array) {
      if (array == null) {
         throw new IllegalArgumentException("Array is null.");
      }
      this.array = array;
      this.arrayListeners = null;
   }

   /**
    * Sets the value at index to the given {@link Object} and
    * informs all at call time available {@link ArrayListener} about the change.
    * <p>
    * The change is represented as {@link ArrayEvent} which contains all
    * details about the performed modification.
    * @param index The index in the array to modify.
    * @param element The element to set at the given index which might be {@code null}.
    */
   public void set(int index, Object element) {
      array[index] = element;
      fireElementChanged(new ArrayEvent(this, index, element));
   }
```

```
54
55    /**
56     * Informs all at call time available {@link ArrayListener} about an array change
57     * by calling {@link ArrayListener#elementChanged(ArrayEvent)}.
58     * @param e The {@link ArrayEvent} to be passed to the {@link ArrayListener} instances.
59     */
60    private void fireElementChanged(ArrayEvent e) {
61       if (arrayListeners != null) {
62          /*@ loop_invariant i >= 0 && i <= arrayListeners.length;
63            @ decreasing arrayListeners.length - i;
64            @ assignable \everything;
65            @*/
66          for (int i = 0; i < arrayListeners.length; i++) {
67             if (arrayListeners[i] != null) {
68                arrayListeners[i].elementChanged(e);
69             }
70          }
71       }
72    }
73
74    /**
75     * Sets the available {@link ArrayListener} instances.
76     * @param arrayListeners The new {@link ArrayListener} instances which might be {@code null}.
77     */
78    public void setArrayListeners(ArrayListener[] arrayListeners) {
79       this.arrayListeners = arrayListeners;
80    }
81 }
```

Listing A.10: Class `ObservableArray`

```
1  /**
2   * Allows to observe changes on an {@link ObservableArray}.
3   * @see ObservableArray
4   * @see ArrayEvent
5   */
6  public interface ArrayListener {
7     /**
8      * Invoked when the element at an array index has changed.
9      * <p>
10     * Implementations require nothing and are allowed to change everything.
11     * An implementation only guarantees that no {@link Exception} will be thrown.
12     * @param e The {@link ArrayEvent} with all details.
13     */
14    /*@ normal_behavior
15      @ requires true;
16      @ ensures true;
17      @ assignable \everything;
18      @*/
19    public /*@ helper @*/ void elementChanged(ArrayEvent e);
20 }
```

Listing A.11: Interface `AbstractSearch`

```
1  /**
2   * An event caused by an {@link ObservableArray} and observed via
3   * {@link ArrayListener} instances.
```

```
 * @see ObservableArray
 * @see ArrayListener
 */
public class ArrayEvent {
   /**
    * The {@link ObservableArray} on which the event occurred.
    */
   private final ObservableArray source;

   /**
    * The modified index in the array.
    */
   private final int index;

   /**
    * The new element.
    */
   private final Object newElement;

   /**
    * Constructor.
    * @param source The {@link ObservableArray} on which the event occurred.
    * @param index The modified index in the array.
    * @param newElement The new element.
    */
   public ArrayEvent(ObservableArray source, int index, Object newElement) {
      this.source = source;
      this.index = index;
      this.newElement = newElement;
   }

   /**
    * Returns the {@link ObservableArray} on which the event occurred.
    * @return The {@link ObservableArray} on which the event occurred.
    */
   public ObservableArray getSource() {
      return source;
   }

   /**
    * Returns the modified index in the array.
    * @return The modified index in the array.
    */
   public int getIndex() {
      return index;
   }

   /**
    * Returns the new element.
    * @return The new element.
    */
   public Object getNewElement() {
      return newElement;
   }
```

```
58 }
```

Listing A.12: Class `ArrayEvent`

	All (15)		None (1)		< 2 y. (5)		≥ 2 y. (9)	
	DCR (8)	SED (7)	DCR (0)	SED (1)	DCR (2)	SED (3)	DCR (6)	SED (3)
Yes, Very helpful	12 %	14 %	0 %		50 %	0 %	0 %	33 %
Yes, Helpful	0 %	29 %	100 %		0 %	0 %	0 %	33 %
Yes, Little helpful	0 %	43 %	0 %		0 %	67 %	0 %	33 %
No, Not helpful	0 %	0 %	0 %		0 %	0 %	0 %	0 %
Not considered	88 %	14 %	0 %		50 %	33 %	100 %	0 %

Table A.5.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution (`ObservableArray(Object[])`)

	All (15)		None (1)		< 2 y. (5)		≥ 2 y. (9)	
	DCR (8)	SED (7)	DCR (0)	SED (1)	DCR (2)	SED (3)	DCR (6)	SED (3)
Yes, Very helpful	0 %	14 %	0 %		0 %	0 %	0 %	33 %
Yes, Helpful	12 %	43 %	0 %		50 %	67 %	0 %	33 %
Yes, Little helpful	12 %	14 %	0 %		0 %	0 %	17 %	33 %
No, Not helpful	0 %	14 %	100 %		0 %	0 %	0 %	0 %
Not considered	75 %	14 %	0 %		50 %	33 %	83 %	0 %

Table A.6.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution (`set(int, Object)`)

	All (15)		None (1)		< 2 y. (5)		≥ 2 y. (9)	
	DCR (8)	SED (7)	DCR (0)	SED (1)	DCR (2)	SED (3)	DCR (6)	SED (3)
Yes, Very helpful	0 %	29 %	0 %		0 %	0 %	0 %	67 %
Yes, Helpful	0 %	43 %	0 %		0 %	100 %	0 %	0 %
Yes, Little helpful	0 %	14 %	0 %		0 %	0 %	0 %	33 %
No, Not helpful	0 %	14 %	100 %		0 %	0 %	0 %	0 %
Not considered	100 %	0 %	0 %		100 %	0 %	100 %	0 %

Table A.7.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution (`setArrayListeners(ArrayListener[])`)

A.2.6 Stack Code Example

```
1 /**
```

```
 * A stack which stores contained elements in an array.
 * <p>
 * A new element is added on top of the stack by {@link #push(Object)}
 * and the top element can be removed by {@link #pop()}.
 */
public final class Stack {
   /**
    * The elements on the stack.
    * The {@link Object} array is never {@code null} and all array indices
    * {@code >= size} are {@code null}.
    */
   /*@ invariant elements != null;
     @ invariant \typeof(elements) == \type(Object[]);
     @ invariant (\forall int i; i >= size && i < elements.length; elements[i] == null);
     @*/
   private final /*@ nullable @*/ Object[] elements;

   /**
    * The current size of the stack
    * which is always a valid array index in {@link #elements}
    * or the length of {@link #elements} if the stack is full.
    */
   /*@ invariant size >= 0 && size <= elements.length;
     @*/
   private int size;

   /**
    * Constructor to create an empty stack with the specified maximal size.
    * @param maximalSize The maximal stack size.
    */
   public Stack(int maximalSize) {
      elements = new Object[maximalSize];
      size = 0;
   }

   /**
    * Constructor for cloning purpose which creates an independent stack
    * with the content of the given {@link Stack}.
    * @param existingStack The existing {@link Stack} fulfilling its class invariant
    *                      which provides the initial content.
    */
   public Stack(Stack existingStack) {
      this.elements = existingStack.elements;
      this.size = existingStack.size;
   }

   /**
    * Adds the given {@link Object} to the stack.
    * @param e The {@link Object} to add.
    * @throws IllegalStateException if the stack is full.
    */
   public void push(Object e) {
      if (size < elements.length) {
         elements[size++] = e;
      }
```

```
57        else {
58           throw new IllegalStateException("Stack is full.");
59        }
60     }
61
62     /**
63      * Returns and removes the top entry from the stack.
64      * @return The top stack entry which was removed from the stack.
65      * @throws IllegalStateException if the stack is empty.
66      */
67     public Object pop() {
68        if (size >= 1) {
69           return elements[--size];
70        }
71        else {
72           throw new IllegalStateException("Stack is empty.");
73        }
74     }
75  }
```

Listing A.13: Class `Stack`

	All (16)		None (1)		< 2 y. (6)		≥ 2 y. (9)	
	DCR (6)	SED (10)	DCR (0)	SED (1)	DCR (2)	SED (4)	DCR (4)	SED (5)
Yes, Very helpful	0 %	40 %	0 %		0 %	50 %	0 %	40 %
Yes, Helpful	17 %	40 %	0 %		0 %	50 %	25 %	40 %
Yes, Little helpful	33 %	20 %	100 %		50 %	0 %	25 %	20 %
No, Not helpful	0 %	0 %	0 %		0 %	0 %	0 %	0 %
Not considered	50 %	0 %	0 %		50 %	0 %	50 %	0 %

Table A.8.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution (`Stack(int)`)

	All (16)		None (1)		< 2 y. (6)		≥ 2 y. (9)	
	DCR (6)	SED (10)	DCR (0)	SED (1)	DCR (2)	SED (4)	DCR (4)	SED (5)
Yes, Very helpful	17 %	30 %	0 %		50 %	25 %	0 %	40 %
Yes, Helpful	17 %	50 %	0 %		0 %	75 %	25 %	40 %
Yes, Little helpful	17 %	10 %	0 %		0 %	0 %	25 %	20 %
No, Not helpful	0 %	10 %	100 %		0 %	0 %	0 %	0 %
Not considered	50 %	0 %	0 %		50 %	0 %	50 %	0 %

Table A.9.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution (`Stack(Stack)`)

	All (16)		None (1)		< 2 y. (6)		≥ 2 y. (9)	
	DCR (6)	SED (10)	DCR (0)	SED (1)	DCR (2)	SED (4)	DCR (4)	SED (5)
Yes, Very helpful	17 %	10 %		0 %	0 %	0 %	25 %	20 %
Yes, Helpful	0 %	70 %		0 %	0 %	100 %	0 %	60 %
Yes, Little helpful	17 %	10 %		0 %	0 %	0 %	25 %	20 %
No, Not helpful	0 %	10 %		100 %	0 %	0 %	0 %	0 %
Not considered	67 %	0 %		0 %	100 %	0 %	50 %	0 %

Table A.10.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution (`push(Object)`)

	All (15)		None (0)		< 2 y. (6)		≥ 2 y. (9)	
	DCR (6)	SED (9)	DCR (0)	SED (0)	DCR (2)	SED (4)	DCR (4)	SED (5)
Yes, Very helpful	17 %	11 %			0 %	0 %	25 %	20 %
Yes, Helpful	0 %	56 %			0 %	75 %	0 %	40 %
Yes, Little helpful	17 %	33 %			0 %	25 %	25 %	40 %
No, Not helpful	0 %	0 %			0 %	0 %	0 %	0 %
Not considered	67 %	0 %			100 %	0 %	50 %	0 %

Table A.11.: Feedback about Helpfulness of the Symbolic Execution Tree and the Concrete Execution (`pop()`)

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010. ISBN 978-0-521-89556-9.

[2] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with Dynamic Slicing and Backtracking. *Softw., Pract. Exper.*, 23(6):589–616, 1993.

[3] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Philipp Rümmer, and Peter H. Schmitt. Verifying Object-Oriented Programs with KeY: A Tutorial. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO'06, pages 70–101, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74791-5, 978-3-540-74791-8.

[4] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY Platform for Verification and Analysis of Java Programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments*, volume 8471 of *Lecture Notes in Computer Science*, pages 55–71. Springer International Publishing, 2014. ISBN 978-3-319-12153-6. doi: 10.1007/978-3-319-12154-3_4.

[5] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Vladimir Klebanov, and Peter H. Schmitt, editors. *The KeY Book: Deductive Software Verification in Practice*. LNCS. Springer, January 2016. Draft Version, Revision #1993, To appear.

[6] J. Stuart Aitken and Thomas F. Melham. An analysis of errors in interactive proof attempts. *Interacting with Computers*, 12(6):565–586, 2000. doi: 10.1016/S0953-5438(99)00023-5.

[7] J. Stuart Aitken, Philip D. Gray, Thomas F. Melham, and Muffy Thomas. Interactive Theorem Proving: An Empirical Study of User Activity. *Journal of Symbolic Computation*, 25(2):263–284, 1998. doi: 10.1006/jsco.1997.0175.

[8] Stuart Aitken, Philip Gray, Tom Melham, and Muffy Thomas. A Study Of User Activity In Interactive Theorem Proving. In Chris Johnson, editor, *Task Centred Approaches To Interface Design: Glasgow Interactive Systems Group Research Review*, pages 195–218. Department of Computing Science, University of Glasgow, August 1995. GIST Technical Report G95.2.

[9] Elvira Albert, Israel Cabanas, Antonio Flores-Montoya, Miguel Gomez-Zamalloa, and Sergio Gutierrez. jPET: An Automatic Test-Case Generator for Java. In *Proceedings of the 18th Working Conference on Reverse Engineering*, WCRE '11, pages 441–442. IEEE CS, 2011. ISBN 978-0-7695-4582-0. doi: 10.1109/WCRE.2011.67.

[10] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF–SE: A Symbolic Execution Extension to Java PathFinder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 134–138. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-71208-4. doi: 10.1007/978-3-540-71209-1_12.

[11] Paul Anderson and Tim Teitelbaum. Software Inspection Using CodeSurfer. In *Proceedings of the 1st Workshop on Inspection in Software Engineering (WISE)*, pages 4–11, Paris, France, July 2001. Software Quality Research Lab, McMaster University.

[12] Irina Mariuca Asavoae, Mihail Asavoae, and Adrián Riesco. Towards a Formal Semantics-Based Technique for Interprocedural Slicing. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, Proceedings*, pages 291–306, September 2014. doi: 10.1007/978-3-319-10181-1_18.

[13] Serge Autexier and Till Mossakowski. Integrating HOL-CASL into the Development Graph Manager MAYA. In Alessandro Armando, editor, *Frontiers of Combining Systems*, volume 2309 of *Lecture Notes in Computer Science*, pages 2–17. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43381-1. doi: 10.1007/3-540-45988-X_2.

[14] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[15] 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, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 49–69, New York, NY, 2005. Springer-Verlag.

[16] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-36749-9. doi: 10.1007/11804192_17.

[17] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, 2015. URL `http://frama-c.com/download/acsl-implementation-Sodium-20150201.pdf`. Version 1.9.

[18] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Lessons Learned From Microkernel Verification – Specification is the New Bottleneck. In Franck Cassez, Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of the 7th Conference on Systems Software Verification*, volume 102 of *EPTCS*, pages 18–32, 2012.

[19] Bernhard Beckert and Christoph Gladisch. White-Box Testing by Combining Deduction-Based Specification Extraction and Black-Box Testing. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland, Revised Papers*, volume 4454 of *LNCS*, pages 207–216. Springer, February 2007. doi: 10.1007/978-3-540-73770-4_12.

[20] Bernhard Beckert and Sarah Grebing. Interactive Theorem Proving - Modelling the User in the Proof Process. In *Proceedings of the Workshop on Bridging the Gap between Human and Automated Reasoning - A workshop of the 25th International Conference on Automated Deduction (CADE-25), Berlin, Germany*, pages 60–73, August 2015.

[21] Bernhard Beckert and Reiner Hähnle. Reasoning and Verification: State of the Art and Current Trends. *Intelligent Systems, IEEE*, 29(1):20–29, Jan 2014. ISSN 1541-1672. doi: 10.1109/MIS.2014.3.

[22] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Number 4334 in Lecture Notes in Computer Science. Springer-Verlag, 2007.

[23] Bernhard Beckert, Sarah Grebing, and Florian Böhl. A Usability Evaluation of Interactive Theorem Provers Using Focus Groups. In *Software Engineering and Formal Methods - SEFM 2014 Collocated*

Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, Revised Selected Papers, pages 3–19, September 2014. doi: 10.1007/978-3-319-15201-1_1.

[24] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2008. ISBN 0321356683, 9780321356680.

[25] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution. *SIGPLAN Not.*, 10(6):234–245, April 1975. ISSN 0362-1340. doi: 10.1145/390016.808445.

[26] Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract Interpretation of Symbolic Execution with Explicit State Updates. In Frank S. de Boer, Marcello M. Bonsangue, and Eric Madeleine, editors, *Revised Lectures, 7th International Symposium on Formal Methods for Components and Objects (FMCO 2008)*, volume 5751 of *LNCS*, pages 247–277. Springer, 2009.

[27] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, June 2005. ISSN 1433-2779. doi: 10.1007/s10009-004-0167-4.

[28] Rod M. Burstall. Program Proving as Hand Simulation with a Little Induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.

[29] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM. ISBN 1-59593-518-5. doi: 10.1145/1180405.1180445.

[30] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[31] James Cheney. Project Report: Theorem Prover Usability. Technical report, Report of project COMM 641, 2001. URL `http://homepages.inf.ed.ac.uk/jcheney/projects/tpusability.ps`.

[32] Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification*, volume 9333 of *Lecture Notes in Computer Science*, pages 297–305. Springer International Publishing, 2015. ISBN 978-3-319-23819-7. doi: 10.1007/978-3-319-23820-3_21.

[33] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-71940-7, 978-3-540-71940-3.

[34] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Berlin, August 2009. Springer-Verlag.

[35] David R. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-20397-8. doi: 10.1007/978-3-642-20398-5_35.

[36] David R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France*, pages 79–92, April 2014.

[37] David R. Cok and Scott C. Johnson. SPEEDY: An Eclipse-based IDE for invariant inference. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France*, pages 44–57, April 2014. doi: 10.4204/EPTCS.149.5.

[38] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, Revised Selected Papers*, pages 108–128, March 2005. doi: 10.1007/978-3-540-30569-9_6.

[39] Stijn De Gouw, Jurriaan Rot, Frank S. De Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case. In Daniel Kroening and Corina Pasareanu, editors, *Proceedings of the 27th International Conference on Computer Aided Verification (CAV), San Francisco*, volume 9206 of *LNCS*, pages 273–289. Springer, July 2015.

[40] Jonathan De Halleux and Nikolai Tillmann. Parameterized Unit Testing with Pex. In *Proceedings of the 2nd International Conference on Tests and Proofs*, LNCS, pages 171–181. Springer, 2008.

[41] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), Tokyo, Japan*, pages 157–166, September 2006. doi: 10.1109/ASE.2006.26.

[42] Edsger W. Dijkstra. The Humble Programmer. *Commun. ACM*, 15(10):859–866, October 1972. ISSN 0001-0782. doi: 10.1145/355604.361591.

[43] Quoc Huy Do, Richard Bubel, and Reiner Hähnle. Exploit Generation for Information Flow Leaks in Object-Oriented Programs. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany*, pages 401–415, May 2015. doi: 10.1007/978-3-319-18467-8_27.

[44] Marco Drebing. Advanced control functions for the Symbolic Execution Debugger. Bachelor's thesis, Technische Universität Darmstadt, Germany, December 2013. URL `http://hds.hebis.de/ulbda/Record/HEB33697423X`.

[45] R. Geoff Dromey. From Requirements to Design: Formalizing the Key Steps. In *Proceedings of the 1st International Conference on Software Engineering and Formal Methods, SEFM, Brisbane, Australia*. IEEE Computer Society, 2003.

[46] Christian Engel and Reiner Hähnle. Generating Unit Tests from Formal Proofs. In *Proceedings of the 1st International Conference on Tests and Proofs*, TAP'07, pages 169–188, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-73769-3, 978-3-540-73769-8.

[47] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[48] Michael E. Fagan. Advances in Software Inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986. ISSN 0098-5589.

[49] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where Programs Meet Provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[50] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512558.

[51] The Eclipse Foundation. Eclipse Celebrates 10 Years of Innovation. `http://www.eclipse.org/org/press-release/20111102_10years.php`, November 2011. Accessed: December 18, 2015.

[52] The Eclipse Foundation. *Platform Plug-in Developer Guide*, 2015. URL `http://help.eclipse.org/mars/index.jsp`. Eclipse Mars (4.5) Documentation.

[53] Michael Frigge, David C. Hoaglin, and Boris Iglewicz. Some Implementations of the Boxplot. *The American Statistician*, 43(1):50–54, 1989. ISSN 00031305. doi: 10.2307/2685173.

[54] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39(1): 176–210, 405–431, December 1935. ISSN 1432-1823.

[55] Christoph Gladisch. Verification-Based Test Case Generation for Full Feasible Branch Coverage. In Antonio Cerone and Stefan Gruner, editors, *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa*, pages 159–168. IEEE Computer Society, 2008.

[56] Christoph David Gladisch. *Verification-based Software-fault Detection*. PhD thesis, Karlsruhe Institute of Technology, 2011. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023056`.

[57] Patrice Godefroid. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190226.

[58] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065036.

[59] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, January 2012. ISSN 1542-7730. doi: 10.1145/2090147.2094081.

[60] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2013. ISBN 978-0133260229.

[61] Jim Gray. Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7, PN87614, Tandem Computers, June 1985.

[62] Wolfgang Grieskamp, Nikolai Tillmann, and Wolfram Schulte. XRT — Exploring Runtime for .NET Architecture and Applications. In Byron Cook, S. Stoller, and W. Visser, editors, *Proceedings of the Workshop on Software Model Checking (SoftMC 2005), Edinburgh, UK*, volume 144(3) of *Electr. Notes Theor. Comput. Sci*, pages 3–26, 2006.

[63] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A Visual Interactive Debugger Based on Symbolic Execution. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 143–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0116-9. doi: 10.1145/1858996.1859022.

[64] Reiner Hähnle, Ina Schaefer, and Richard Bubel. Reuse in Software Verification by Abstract Method Calls. In *Proceedings of the 24th International Conference on Automated Deduction*, volume 7898 of *LNCS*, pages 300–314. Springer, 2013.

[65] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0262082896.

[66] Martin Hentschel, Richard Bubel, and Reiner Hähnle. Symbolic Execution Debugger (SED). In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification, 14th International Conference, RV, Toronto, Canada*, volume 8734 of *LNCS*, pages 255–262. Springer, 2014.

[67] Martin Hentschel, Reiner Hähnle, and Richard Bubel. Visualizing Unbounded Symbolic Execution. In Martina Seidl and Nikolai Tillmann, editors, *Proceedings of Testing and Proofs (TAP) 2014*, LNCS, pages 82–98. Springer, July 2014.

[68] Martin Hentschel, Stefan Käsdorf, Reiner Hähnle, and Richard Bubel. An Interactive Verification Tool Meets an IDE. In Elvira Albert and Emil Sekerinski, editors, *Integrated Formal Methods*, volume 8739 of *Lecture Notes in Computer Science*, pages 55–70. Springer International Publishing, 2014. ISBN 978-3-319-10180-4. doi: 10.1007/978-3-319-10181-1_4.

[69] David Hovemeyer and William Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, December 2004. ISSN 0362-1340. doi: 10.1145/1052883.1052895.

[70] Marieke Huisman, Wolfgang Ahrendt, Daniel Bruns, and Martin Hentschel. Formal specification with JML, 2014. URL `http://doc.utwente.nl/94636/`.

[71] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0201847485.

[72] Andreas Ibing. Parallel SMT-Constrained Symbolic Execution for Eclipse CDT/Codan. In *Testing Software and Systems - 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, Proceedings*, pages 196–206, November 2013. doi: 10.1007/978-3-642-41707-8_13.

[73] Andrew Ireland, Michael Jackson, and Gordon Reid. Interactive Proof Critics. *Formal Aspects of Computing*, 11(3):302–325, 1999. ISSN 0934-5043. doi: 10.1007/s001650050052.

[74] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008. URL `http://www.cs.kuleuven.be/~bartj/verifast/verifast.pdf`.

[75] Bart Jacobs, Jan Smans, and Frank Piessens. A Quick Tour of the VeriFast Program Verifier. In Kazunori Ueda, editor, *Programming Languages and Systems*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-17163-5. doi: 10.1007/978-3-642-17164-2_21.

[76] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-20397-8.

[77] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 758–766, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31423-0. doi: 10.1007/978-3-642-31424-7_61.

[78] Konrad Jamrozik, Gordon Fraser, Nikolai Tillmann, and Jonathan De Halleux. Augmented Dynamic Symbolic Execution. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 254–257, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2. doi: 10.1145/2351676.2351716.

[79] Ran Ji and Richard Bubel. PE-KeY: A Partial Evaluator for Java Programs. In John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne, editors, *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, Proceedings*, volume 7321 of *Lecture Notes in Computer Science*, pages 283–295. Springer-Verlag, June 2012. ISBN 978-3-642-30728-7.

[80] Ran Ji, Reiner Hähnle, and Richard Bubel. Program Transformation Based on Symbolic Execution and Deduction. In Mario Bravetti Robert M. Hierons, Mercedes G. Merayo, editor, *Proceedings of the 11th International Conference on Software Engineering and Formal Methods (SEFM)*, volume 8137 of *Lecture Notes in Computer Science*, pages 289–304. Springer, 2013. ISBN 978-3-642-40560-0.

[81] Uri Juhasz, Ioannis T. Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. Technical report, ETH Zurich, 2014. URL `http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=JKMNSS14.pdf`.

[82] Gada Kadoda, Roger Stone, and Dan Diaper. Desirable features of educational theorem provers - a Cognitive Dimensions viewpoint. In *Proceedings of the 11th Annual Workshop of the Psychology of Programming Interest Group*, 1996.

[83] Stefan Käsdorf. Efficient automatic proof management in Eclipse for KeY. Bachelor's thesis, Technische Universität Darmstadt, Germany, December 2013. URL `http://hds.hebis.de/ulbda/Record/HEB336974558`.

[84] Shmuel Katz and Zohar Manna. Towards Automatic Debugging of Programs. In *Proceedings of the International Conference on Reliable Software*, pages 143–155, New York, NY, USA, 1975. ACM. doi: 10.1145/800027.808434.

[85] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252.

[86] Vladimir Klebanov. Proof Reuse. In Beckert et al. [22].

[87] Stephen Cole Kleene. *Introduction to Metamathematics*, volume 1 of *Bibliotheca mathematica*. North-Holland Publishing Co., 1952.

[88] Andrew J. Ko and Brad A. Myers. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM. ISBN 1-58113-702-8. doi: 10.1145/985692.985712.

[89] Andrew J. Ko and Brad A. Myers. Extracting and Answering Why and Why Not Questions About Java Program Output. *ACM Trans. Softw. Eng. Methodol.*, 20(2):4:1–4:36, September 2010. ISSN 1049-331X. doi: 10.1145/1824760.1824761.

[90] B. Korel. A Dynamic Approach of Test Data Generation. In *Conference on Software Maintenance, Proceedings*, pages 311–317, Nov 1990. doi: 10.1109/ICSM.1990.131379.

[91] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. The Boogie Verification Debugger. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, SEFM'11, pages 407–414, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24689-0.

[92] 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*, volume 523 of *The Springer International Series in Engineering and Computer Science*, pages 175–188. Springer US, 1999. ISBN 978-1-4613-7383-4. doi: 10.1007/978-1-4615-5229-1_12.

[93] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*, May 31, 2013. Draft Revision 2344.

[94] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-17510-4, 978-3-642-17510-7.

[95] K. Rustan M. Leino and Michal Moskal. Usable Auto-Active Verification. `http://fm.csl.sri.com/UV10/submissions/uv2010_submission_20.pdf`, 2010.

[96] K. Rustan M. Leino and Valentin Wüstholz. Fine-Grained Caching of Verification Results. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, volume 9206 of *Lecture Notes in Computer Science*, pages 380–397. Springer International Publishing, 2015. ISBN 978-3-319-21689-8. doi: 10.1007/978-3-319-21690-4_22.

[97] K. Rustan M. Leino and Valentin Wüstholz. The Dafny Integrated Development Environment. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France*, pages 3–15, April 2014. doi: 10.4204/EPTCS.149.2.

[98] B. Lewis. Debugging Backwards in Time. *Arxiv preprint cs.SE*, Proc. AADEBUG 2003, September 2003. URL `http://arxiv.org/abs/cs/0310016`.

[99] Barbara H. Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994. ISSN 0164-0925. doi: 10.1145/197320.197383.

[100] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619670, 9780735619678.

[101] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[102] Martin Möller. Guided Navigation in Symbolic Execution Trees. Bachelor's thesis, Technische Universität Darmstadt, Germany, December 2014. URL `http://hds.hebis.de/ulbda/Record/HEB355490420`.

[103] Till Mossakowski, Serge Autexier, and Dieter Hutter. Development Graphs—Proof Management for Structured Specifications. *J. Logic & Alg. Progr.*, 67(1–2):114–145, 2006.

[104] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[105] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184, New York, NY, USA, 1984. ACM. ISBN 0-89791-131-8. doi: 10.1145/800020.808263.

[106] André Platzer. An Object-Oriented Dynamic Logic with Updates. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, September 2004. URL `http://i12www.ira.uka.de/~key/doc/2004/odlMasterThesis.pdf`.

[107] Guillaume Pothier and Éric Tanter. Back to the Future: Omniscient Debugging. *IEEE Software*, 26 (6):78 – 85, 2009. doi: 10.1109/MS.2009.169.

[108] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable Omniscient Debugging. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 535–552, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297067.

[109] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: 10.1145/1390630.1390635.

[110] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured Specifications and Interactive Proofs with KIV. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 of *Applied Logic Series*, pages 13–39. Springer Netherlands, 1998. ISBN 978-90-481-5051-9. doi: 10.1007/978-94-017-0435-9_1.

[111] Wolfgang Reif and Kurt Stenzel. Reuse of Proofs in Software Verification. In *Foundations of Software Technology and Theoretical Computer Science*, pages 284–293, 1993.

[112] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 445–454, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806864.

[113] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An Extensible and Highly-modular Software Model Checking Framework. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 267–276, New York, NY, USA, 2003. ACM. ISBN 1-58113-743-5. doi: 10.1145/940071.940107.

[114] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1. doi: 10.1109/SP.2010.38.

[115] Christoph Scheben. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014. URL `http://nbn-resolving.org/urn:nbn:de:swb:90-468785`. Karlsruhe, KIT, Diss., 2014.

[116] Christoph Scheben and Peter H. Schmitt. Verification of Information Flow Properties of Java Programs without Approximations. In *Formal Verification of Object-Oriented Software International Conference, FoVeOOS 2011, Revised Selected Papers*, Lecture Notes in Computer Science, pages 232–249. Springer-Verlag, 2012.

[117] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA, 1983. ISBN 0262192187.

[118] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin Slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250748.

[119] Christopher Paul Svanefalk. KeYTestGen2: an automatic, verification-driven test case generator. Bachelor's thesis, University of Gothenburg, Sweden, June 2013. URL `http://gupea.ub.gu.se/bitstream/2077/37089/1/gupea_2077_37089_1.pdf`.

[120] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-based Deductive Verification of Software Product Lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 11–20, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1129-8. doi: 10.1145/2371401.2371404.

[121] Thomas Thüm, Jens Meinicke, Fabian Benduhn, Martin Hentschel, Alexander von Rhein, and Gunter Saake. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 177–186, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2740-4. doi: 10.1145/2648511.2648530.

[122] Thomas Thüm, Tim Winkelmann, Reimar Schröter, Martin Hentschel, and Stefan Krüger. Variability Hiding in Contracts for Dependent Software Product Lines. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '16, pages 97–104, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4019-9. doi: 10.1145/2866614.2866628.

[123] Nikolai Tillmann and Jonathan De Halleux. Pex–White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-79123-X, 978-3-540-79123-2.

[124] Frank Tip. A Survey of Program Slicing Techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[125] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 566–580. Springer, 2015. doi: 10.1007/978-3-662-46681-0_53.

[126] Bart van Delft and Richard Bubel. Dependency-Based Information Flow Analysis with Declassification in a Program Logic. *arXiv.org*, 2015. Available at `http://arxiv.org/abs/1509.04153`.

[127] Rudolf van Megen and Dirk B. Meyerhoff. Costs and benefits of early defect detection: experiences from developing client server and host applications. *Software Quality Journal*, 4(4):247–256, 1995. ISSN 0963-9314. doi: 10.1007/BF00402646.

[128] Dries Vanoverberghe and Frank Piessens. Theoretical Aspects of Compositional Symbolic Execution. In *Fundamental Approaches to Software Engineering: 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany*, volume 6603 of *LNCS*, pages 247–261. Springer, April 2011.

[129] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engg.*, 10(2):203–232, April 2003. ISSN 0928-8910. doi: 10.1023/A:1022920129859.

[130] Nathan Wasser, Richard Bubel, and Reiner Hähnle. Array Abstraction with Symbolic Pivots. Technical report, Department of Computer Science, Technische Universität Darmstadt, Germany, August 2015. URL http://www.se.tu-darmstadt.de/fileadmin/user_upload/Group_SE/Publications/ALBIA/TR_Symbolic_Pivots.pdf.

[131] Mark Weiser. Programmers Use Slices when Debugging. *Commun. ACM*, 25(7):446–452, July 1982. ISSN 0001-0782. doi: 10.1145/358557.358577.

[132] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1979. AAI8007856.

[133] Benjamin Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000021694.

[134] Daniel Winterstein, David Aspinall, and Christoph Lüth. Proof General/Eclipse: A Generic Interface for Interactive Proof. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 1587–1588, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[135] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012. ISBN 978-3-642-29043-5. doi: 10.1007/978-3-642-29044-2.

[136] Darin Wright and Bjorn Freeman-Benson. How to write an Eclipse debugger. http://www.eclipse.org/articles/Article-Debugger/how-to.html, August 2004.

[137] Andreas Zeller. *Why Programs Fail—A Guide to Systematic Debugging*. Elsevier, 2nd edition, 2006. ISBN 978-1-55860-866-5.