MASTER'S THESIS

Formal analysis of the Java Collections Framework

de Boer, M.

Award date:
2021

Formal analysis of the Java Collections framework

Martin de Boer

Presentation date
Friday, July 23, 2021 at 11:30 AM

FORMAL ANALYSIS OF THE JAVA COLLECTIONS FRAMEWORK

by

Martin de Boer

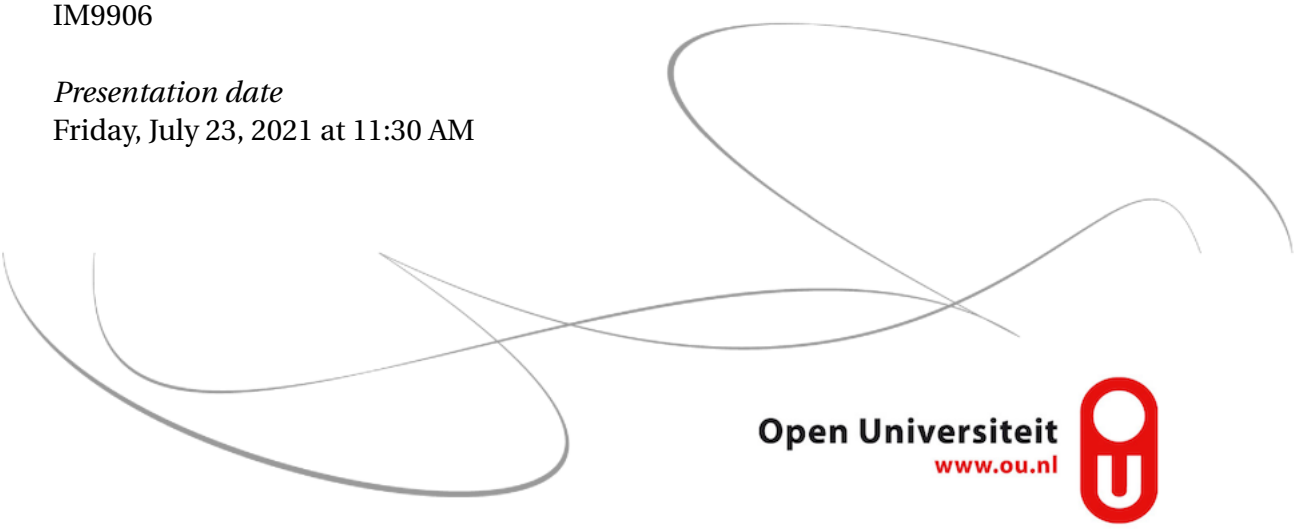
Degree programme
Open University of The Netherlands, Faculty of Management, Science and Technology
Master's Programme in Software Engineering

Graduation committee

Supervisor:	Dr. Stijn de Gouw
Secondary supervisor:	Prof. dr. Tanja Vos
Third assessor:	Dr. ir. Harald Vranken

Course code
IM9906

Presentation date
Friday, July 23, 2021 at 11:30 AM



ACKNOWLEDGEMENTS

I am grateful to a number of individuals who have provided help and support during the course of my graduation project. First and foremost, I would like to thank my supervisor Stijn de Gouw for his guidance, patience, valuable feedback, and for being available for questions at any time. I am grateful for being offered this research project, that I found very interesting and instructive.

I thank Mattias Ulbrich, researcher at the Karlsruher Institut für Technologie (KIT), Alexander Weigl, researcher/PhD student at KIT, and Jonas Klamroth, scientific staff member at KIT. During our regular Zoom meetings, 1 or 2 times a month, also joined by Stijn de Gouw, we had many fruitful discussions about my project as well as formal specification and verification issues in general. They provided me with useful suggestions to resolve some of the issues I ran into. Mattias and Alexander, being the experts on KeY, helped me out with a number of technical issues I encountered. Together with Jonas, being the developer of the model checking tool JJBMC, I was able to detect some errors in an early version of my JML contract specifications by processing them with JJBMC. (This has been to our mutual benefit, I hope.)

I also would like to thank Tanja Vos for providing me with useful feedback on my research proposal, as well as a concept version of my thesis.

Furthermore, I would like to express my gratitude to my employer, New Nexus, for funding my academic venture.

Finally, I would like to thank the three persons closest to me for their support. My daughters, Wendy and Sanne, who studied and lived in the UK for a number of years and master the English language far better than I do; thank you especially for proof reading my thesis a number of times. And last but not least my wife Wilma, whom I will finally be spending more time with now I finished my project, for her moral support.

CONTENTS

List of Figures

List of Tables

LISTINGS

SUMMARY

We may not be actively aware of it most of the time, but software has become an integral part of our everyday lives. We take its correctness for granted. Indeed, we put our trust in software that is essential for our comfort, well-being, safety, or even our survival [1, 2, 3]. It is therefore of the utmost importance that software, particularly library software that is being re-used in countless applications, contains no serious errors.

One way to detect errors in software is formal analysis. The three principal formal analysis methods found in literature [4, 5], are *model checking*, *abstract interpretation*, and *deductive methods*. They all have their strengths and weaknesses, depending on the kind of software under analysis [4]. To analyse the Java Collections Framework, the *deductive method* [6, 7] is the most appropriate method for a complete analysis without restrictions.

We have applied the deductive method, formally specifying the `IdentityHashMap` class of the Java Collections Framework (JDK7) using Java Modelling Language (JML) [8, 9], and formally verifying it with the interactive state-of-the-art theorem prover KeY [10, 11].

To be able to detect mistakes in the JML specifications early in the process, we used a number of other formal analysis tools and unit tests. The reason for this is that deductive verification is tedious and time-consuming work, and we wanted to see how we could speed up the process by detecting errors in the formal specification as early in the process as possible. This *hybrid* approach of using multiple tools to detect specification errors in an early stage enabled us to limit the amount of man hours spent on tedious correction work, especially considering the extensive JML contracts that are required for a class like the `IdentityHashMap`. According to literature, hybrid approaches have been taken before [12], albeit with the objective to compare the verification tools (KeY and OpenJML, for example), and learn about their pros and cons.

As a result of our formal analysis of the `IdentityHashMap` we actually found an overflow error in the `capacity` method. Although the consequences of this overflow error affect the inner workings of the class and its performance, the software does not crash. Nevertheless, we wrote an improved version of the `capacity` method that does not suffer from the overflow error, tested it with a unit test, and proved its correctness with KeY. Furthermore, we found a vulnerability in the serialisation of the class, for which we propose an improvement.

Finally, we contributed to the development of a tool that enables a software bounded model checker for Java (JBMC) to verify contracts written in JML (JJBMC). JJBMC is still in the development phase, but we nevertheless used it to verify the `IdentityHashMap` based on our JML specifications. We detected some bugs in this tool, but we were still able to use it to point out some errors in an early version of our JML specifications as well. A win-win situation.

1

INTRODUCTION

1.1. RATIONALE

1.1.1. THE OMNIPRESENCE OF SOFTWARE AND THE IMPACT OF BUGS

Software is ubiquitous nowadays. From games to health care systems, from household devices to financial systems, from cars and airplanes to communication systems. It is obvious software impacts our daily lives in many ways. We often take working software for granted. However, quite regularly, the software is failing us, due to errors. The impact of software failure depends on the application, and may vary from slight inconveniences on one end of the scale, to life threatening situations on the other end.

Imagine a bug causing a video game to crash. Slightly inconvenient for the player, but no real harm is done. If this happens too often, however, there might be indirect economic consequences for the manufacturer of the game. Software bugs can also have serious immediate economic or financial consequences. Think of down-time of a web server hosting an e-commerce application, affecting the revenue of the company, or bugs in banking software.

Moreover, software errors may also seriously affect our safety and well-being. Malfunctioning of medical devices can have direct life-threatening consequences to a patient. So can have failing communications systems. In June 2019, for example, the Dutch emergency number 112 could not be reached for more than three hours. At least one death was reportedly caused by the fact that emergency services could not be reached, and help arrived too late at the site of the emergency.

Bugs in transportation software are another example that could cause life-threatening situations, possibly on an even larger scale. As an example, see subsection 1.1.2 (Example 1: an overflow error in Boeing 787 software) on the integer overflow bug in the Boeing Company Model 787 aircraft software, discovered in 2015.

If erroneous software is used in military equipment, the consequences could possibly exceed the examples mentioned above. An infamous example of failing software in military equipment is an arithmetic error in the Patriot Missile system in 1991, resulting in 28 deaths and leaving around 100 people wounded. See subsection 1.1.3 (Example 2: an arithmetic error in the Patriot Missile system).

1.1.2. Example 1: an overflow error in Boeing 787 software

On the 1st of May, 2015, the Federal Aviation Administration (FAA) issued an airworthiness directive (AD) for all the Boeing Company Model 787 aircraft (the Dreamliner) [1]. An integer overflow was triggered after 248 days of continuous power, causing the general control units (GCUs) to simultaneously go into failsafe mode, causing the plane to lose all electrical power, resulting in loss of control of the airplane. The short term 'solution' to the problem, according to the AD, was a repetitive maintenance task: electrical power deactivation at intervals not to exceed 120 days.

The cause of this integer overflow isn't hard to guess. According to the FAA "[t]he software counter internal to the generator control units (GCUs) will overflow after 248 days of continuous power". If a 32-bit counter is incremented once every 10 ms, a signed counter will overflow in about 248.6 days [13] ($2^{31}/100/60/60/24 = 248.5513...$).

When the problem was discovered, close to 300 planes of this type were in operation. Boeing 787 variants can seat 242 to 330 passengers, making us shudder at the thought of the nightmare that might have hit the Dreamliner passengers if the bug was trigged at high altitude.

1.1.3. Example 2: an arithmetic error in the Patriot Missile system

An infamous example of failing software in military equipment is an arithmetic error in the Patriot missile system. The goal of this system is to detect, target and hit incoming missiles [2]. On the 25th of February, 1991, in Dhahran, Saudi Arabia, during the Gulf War, an American Patriot Missile battery failed to intercept an incoming Scud missile, resulting in 28 deaths and leaving around 100 people wounded.

To determine whether an incoming object is a Scud, the *range gate algorithm* calculates the area in the sky where the system should look next (based on Scud characteristics like velocity, latitude, longitude, and altitude). This is called the *range gate prediction*. In this case, the radar system correctly detected an airborne object, but failed to identify the object as a Scud missile, because the range gate prediction algorithm contained a bug.

To determine the *range gate prediction*, both the timestamp and the velocity must be expressed as real numbers. However, the timestamps of two consecutive radar pulses being compared were converted to real numbers differently. One correctly, but the other proportionate to the operation time (in tenths of seconds) of the system. In this particular case, the system had been in operation for 100 consecutive hours, corresponding to an integer timestamp in tenths of seconds of $100 \times 60 \times 60 \times 10 = 3,600,000$. To convert this integer to a real number in seconds, it was multiplied by 1/10 using a 24-bit register. The binary representation of 1/10 is 0.0001100110011001100110011001100..., which is more than 24 bits, so it had to be truncated. The truncated value differs from the actual value by 0.000000000000000000000011001100..., or about 0.00000095367431618 42 decimal. This seems like a small number, but when multiplied by 3,600,000, the calculation is off by approximately 0.3433 seconds. Considering the velocity of a Scud missile is about 3,750.2563 miles per hour, or 1,676.5146 meters per second, it is obvious the missile had travelled roughly 575 meters past the predicted range gate ($1,676.5146 \times 0.3433$), undetected by the Patriot Missile battery.

1.1.4. The special case of Off-The-Shelf software

A special kind of software is software that is used to build other software. Think of software libraries containing software components for general use, often referred to as Off-The-Shelf (OTS) software or Commercial Off-The-Shelf (COTS) software. This kind of software might ultimately be used in a multitude of applications, developed by hundreds or even thousands of companies around the globe. These applications could be used for entertainment purposes like games, but also in cars, planes, and medical or military devices.

In recent years Java has become one of the most widely used programming languages across many industries. In July 2021, Java was the second most popular programming language, according to the TIOBE Index [1]. Obviously, the OpenJDK, the most widely used implementation of the Java standard library, contains components that are the building blocks of millions of software applications world-wide. Therefore, bugs in the OpenJDK have a global impact.

1.1.5. Testing and formal analysis

The necessity of finding and fixing software bugs is obvious from the examples above, and this is especially the case for OTS software like the OpenJDK. A thorough software process is, therefore, indispensable, and testing is an essential part of that process. However, testing is experimental, and although it can show the presence of (some) errors, it cannot prove the absence of (all) errors. Typically, overflow errors like the ones mentioned above, and out-of-bounds errors are often overlooked during testing (absence of coverage being the main problem). Overflow and out-of-bounds errors are typical for collections, especially when these collections grow very large. It is of course possible to detect overflow errors by thoroughly testing the software, but it is extremely hard – perhaps virtually impossible – to design a sufficient test set to cover all the cases in which such errors might occur. This is especially true in the case of black box testing. This problem is underlined by the problem of *state space explosion* (see appendix A, section A.1, Model checking), due to the number of interacting components that can assume many different values, resulting in a very large number of possible states of a system. So, there is a need for formal methods to guarantee reliable software. Formal analysis enables us to formally prove the correctness of software. In practice, however, formal analysis, especially when it involves the approach of deductive verification, is perceived as tedious, time-consuming, and inefficient [14, 15]. Some valid counterarguments can be made against these objections:

- Because OTS software components like classes in the OpenJDK Collections Framework are being used in countless numbers of applications world-wide, it is worth our while to formally verify them. This makes many of these classes outstanding candidates to formally analyse. A similar observation was made by Polikarpova *et al.* [16], concerning *containers*, libraries of general-purpose data structures.

- Many of the components in the OpenJDK are not prone to regular or rigorous adjustments due to ever-changing user requirements. Formal specifications that fully capture the behaviour of such components will, therefore, be fairly stable and will not regularly require adjustments involving more tedious and time-consuming work.

[1] `https://www.tiobe.com/tiobe-index/`

- The impact of software errors, as is demonstrated in the examples above, can have serious financial and even life-threatening consequences. Preventing these is certainly worth our while.

- The costs of debugging and fixing software errors are already significant [17, 18], and it is obvious that the more software is being developed, the more time is spent on debugging and fixing errors in that software. Formally proving the correctness of OTS software that is reused in countless applications globally, might reduce these costs by preventing bugs in the first place.

1.1.6. Formal analysis methods and the case for deductive verification

The principal validation methods mentioned in literature [4, 5], apart from simulation and testing, are *model checking*, *abstract interpretation*, and *deductive methods*. (A global description of each of these three methods is provided in appendix A, Formal analysis methods.) Determining which technique was most suitable for our research and why, depended on which properties we wanted to analyse. Candidate properties were deadlocks, overflows, rounding errors, race conditions, unhandled exceptions, and violations of stated assertions, to name a few. What kind of errors could we suspect to be present in the OpenJDK Collections Framework library? Recent related work [14, 19, 20] had shown that typical errors that are expected to be found are numerical (overflow or rounding) errors. In fact, some of the most dramatic examples of software bugs, some of which were mentioned above, are due to numerical (overflow or rounding) errors [1, 2, 3, 21]. Analysing the OpenJDK Collections Framework library for these kinds of numerical errors was therefore an obvious choice.

To be able to find errors like these, it was critical to make as few adjustments to the original code as possible, because any adjustment might have prevented us from finding the original bug. Furthermore, adjustments might not just unintentionally have hidden or removed errors, but might also have introduced new ones. The first requirement for selecting a formal analysis approach was, therefore, to be able to stay as close as possible to the original syntax and semantics of the Class Under Analysis (CUA).

The second requirement was that we wanted to be able to verify infinite, unbounded collections. Typically, these are the kinds of objects that are prone to overflow errors, and constituted our list of CUA candidates. Our formal analysis approach had, therefore, to support infinite, unbounded collections. (Even if we would subsequently decide to verify a finite collection, e.g. the `java.util.ArrayList` class, we would need to be aware of the maximum size of instances of the class. An `ArrayList` object can have a maximum length of `MAX_INTEGER` (i.e. 2,147,483,647) elements. Elements of such an object can in turn also be `ArrayList` objects of the same length, etc. This might have resulted in a tremendously large state space.) To summarise, the following two requirements had to be met:

R1 : minimal diversion from the original syntax and semantics of the CUA
R2 : support for verifying (virtually) infinite, unbounded collections

Having determined the properties we wanted to analyse, and having derived two major requirements from them, we were able to argue which approach best suited our purposes and why.

Model checking (see appendix A, section A.1, Model checking) requires the conversion of the system into a model, an abstraction of the system, complemented with temporal logic, asserting how the system should behave over time. Typically, such a model, that is automatically verified by a model checker, lacks irrelevant details to save time and memory. The technique is especially suitable for finite state concurrent systems, and quite capable of detecting deadlocks, for example. The first drawback of model checking, in relation to our research, was the creation of a model of the system. Our first requirement (**R1**) was to stay as close as possible to the original syntax and semantics of the CUA. This requirement seemed to be violated by creating a model, especially when an abstraction of the design would have been made to save memory and execution time. Furthermore, and this is a second, more general drawback in model checking, it suffers from the possibility of a *state space explosion*. This clashed with the second requirement (**R2**), that we wanted to be able to verify (virtually) infinite, unbounded collections, or collections that might grow very large in size. Model checking was, therefore, not a suitable approach for our research.

Abstract interpretation is more scalable and does, therefore, not suffer from the problem of state space explosion. In abstract interpretation a program is described in terms of *abstract properties* and *abstract operations*, abstractions of a program's concrete semantics, that make up a so-called *abstract domain*. Being approximations, the abstract semantics in the abstract domain are, obviously, less precise. And this lack of precision comes at a cost. This is illustrated in appendix A (section A.2, Abstract interpretation), with an example which shows that overflow errors might go unnoticed. Furthermore, it is obvious that our first requirement (**R1**), to stay as close as possible to the original syntax and semantics of the CUA, would obviously not be met if we had needed to create an abstract approximation of the program to be analysed. Abstract interpretation was, therefore, also not suitable for our research.

Deductive methods (see appendix A, section A.3, Deductive methods) are based on the so-called Hoare triplets [6]. Hoare triplets consist of two assertions, a precondition P and a postcondition R, as well as a command or program C. To express that if precondition P holds before the initiation of program C, postcondition R holds after its completion, we can write the following triplet: $P\{Q\}R$. Dwyer *et al.* [4] have made a comparison between model checking, abstract interpretation and deductive methods. The most important drawbacks of deductive methods they mention are:

- they are not suitable for concurrent systems,
- they are difficult to work with, and
- automation is limited.

The first drawback was not relevant for our purpose. When formally analysing a class in the OpenJDK Collections Framework library, concurrency is not a concern. The second drawback was partly eliminated by the findings of De Gouw *et al.* [14, 19], who had shown that contract-based deductive verification is feasible, when applying the proper tools. JML [9, 22] was used to formally specify pre- and postconditions and invariants, and the state-of-the-art Java verification tool KeY [10], a semi-automatic, interactive theorem prover, was used to find 99% of the proof steps automatically. This also disproved the third drawback of limited automation.

Based on the requirements stated above, we decided the deductive method was our best candidate. Requirement **R1**, minimal diversion from the original syntax and semantics of

the CUA, was warranted because, when using JML, the pre- and postconditions as well as the invariants are specified inside the original Java code [23]. We would not be defining any model (like with model checking) or any other kind of abstraction (like with abstract interpretation) of the program we wanted to analyse. Requirement **R2** could be warranted also, because we did not need to worry about a state space explosion. We concluded, therefore, that the deductive method was the preferred method to apply when formally analysing (a) class(es) from the OpenJDK Collections Framework.

Table 1.1 summarises the most important characteristics (strengths, weaknesses, and irrelevances) of the above-mentioned formal analysis methods, specifically in relation to our research.

Model checking	Abstract interpretation	Deductive methods
⊕ Supports fully automated verification. ⊕ Does not require significant mathematical expertise to set up a model, or to specify properties.	⊕ More scalable than model checking. ⊕ Suitable for infinite, unbounded systems (requirement **R2**).	⊕ Tooling that supports verification of Java software available. ⊕ Minimal diversion from the original syntax and semantics: the code is the model (requirement **R1**). ⊕ Suitable for infinite, unbounded systems (requirement **R2**).
⊙ Supports verification of concurrent systems.		⊙ Difficult to extend to concurrent systems.
⊖ Only suitable for finite state systems. ⊖ Suffers from scalability (state space explosion) challenges. ⊖ Verifies an abstraction (model) of the system. ⊖ Not tailor-made for Java.	⊖ Verifies an over-approximation of the program, leading to inconclusive error reports and some error types going unnoticed. ⊖ Requires significant mathematical expertise, to set up abstractions. ⊖ Not tailor-made for Java.	⊖ Tedious work, automation is limited.

⊕ = strength, ⊙ = irrelevant, ⊖ = weakness

Table 1.1: Pros and cons of formal analysis methods

1.1.7. Deductive verification tools: the case for JML and KeY

There is a wide range of deductive verification tools available, three of which are being considered in appendix B (Deductive verification tools) on page V. Based on the findings in appendix B, we concluded that both Isabelle/HOL and Coq suffer from a number of drawbacks: their learning curves are quite steep, they are not tailor-made for Java, and the available theories/logics for Java that are available lack the required features. It must be noted, however, that Coq is able to process proof obligations generated by *Krakatoa*, a tool that supports the use of JML. Burdy *et al.* [23] observe a number of vital advantages of the use of JML for Java development, for example:

- JML's syntax and semantics are very close to Java,
- gradual introduction is easy, because no formal model has to be constructed (the source code *is* the formal model),
- as mentioned above, there is no discrepancy between the actual code and the formal model, and
- there is tool support available for JML.

If, however, JML would be the specification language of our choice, there was also the KeY tool [10, 23] to seriously consider as the verification tool to use for our research. KeY supports the use of JML and has, reassuringly, been shown to be very effective for formally verifying Java code in the related work mentioned in section 1.3 (Related work) on page 7. Thanks to Stijn de Gouw's ties with peers of the Karlsruher Institut für Technologie (KIT), where the KeY tool is being developed, we would be able to consult the tool experts during our research. This would, obviously, be a great advantage. We therefore decided to use JML as our specification language, and KeY as our verification tool.

1.2. Objective

Errors in OTS like Java's OpenJDK can have far-reaching consequences, because it is used in countless numbers of applications, globally. As we have argued above, formal analysis, especially deductive verification, seemed to us the best way to guarantee the correctness of this kind of software. Based on these considerations, the objective of our research was to formally analyse a class in the OpenJDK, by formally specifying the code with JML and verifying it with KeY. The aim of the analysis was to either prove the selected class to be correct, or, if any error should be found, to correct the error, prove its correctness with KeY and a unit test, and propose this fix to the community.

As mentioned above, formal analysis is can be time-consuming and tedious. Therefore, it seemed wise to limit our objective to the analysis of a single class. We decided the ideal candidate to formally analyse would be the `IdentityHashMap` in the Collections Framework. This decision was part of the research, and our considerations are described in section 2 (Method) on page 9. Furthermore, as a secondary objective, we wanted to find out if we could limit the effort of formally specifying and verifying that candidate, by considering and applying other tools as well (in an attempt to quickly detect specification errors).

1.3. Related work

Similar research has been done by De Gouw *et al.* on the TimSort algorithm provided by the Java Standard Library [14, 19]. De Gouw *et al.* applied the deductive verification approach,

using JML to write the formal specifications for the algorithm, and the state-of-the-art Java formal verification tool KeY [10] to verify the code. Their analysis showed that the TimSort algorithm contained an error.

Related research was done by Hans-Dieter A. Hiep *et al.* [20], by formally verifying Java's LinkedList with JML and KeY, and by Mostowski *et al.* [24], who have formally verified security properties of the Java Card system, also using KeY. Mostowski performed two Java Card case studies but did not find and eliminate any software bugs. More recent work by Mostowski on the Java Card API reference implementation [25], however, did result in the detection of a bug in one of the commercially sold cards.

Pottier *et al.* [15] formally verified their own implementation of a hash table (written in OCaml), using Characteristic Formulae for ML, a tool for the interactive verification of OCaml programs, based on Coq (CFML). Although the subject of their project was not a Java class, and it was not verified with JML and KeY, there is one similarity to our project: the formal verification of a hash table. However, this hash table was specifically implemented as part of their project, and not part of a widely used general purpose utility component. Also, some errors that typically can occur in unbounded data structures were left out of scope. For example, one of the assumption made was that the authors "pretend[ed] that OCaml integers are unbounded, which is not true: they are 31- or 63-bit integers in two's-complement representation".

The novelty of our research was that, to our knowledge, this was the first time a hash map which is undoubtedly used in many software applications worldwide was formally verified.

2

METHOD

This chapter describes the method we used. In section 2.1 we will provide some background information on the used technology and explain some of the related terminology. The research questions for the project will be addressed in section 2.2 (page 18). Finally, we will describe the research process that we applied to answer these research questions in section 2.3 (page 20).

2.1. RESEARCH CONTEXT

In this section, some background information on the used technology is provided, and some of the jargon that is related to the subject of formal analysis, specifically the deductive approach, is explained.

2.1.1. HOARE LOGIC

In 1969, C.A.R. Hoare published a paper in which he proposed several axioms to help prove the correctness of a computer program [6], commonly referred to as Hoare logic. Central to Hoare logic, is the so-called Hoare triplet:

$$P\{Q\}R.$$

In this triplet, the predicates P and R represent the states before and after the execution of the program Q, respectively. The reading of the triplet is: whenever P is true before the execution of Q, then the assertion of R will be true afterwards, or Q does not terminate. Hoare logic lies at the core of formal specification and deductive verification of computer programs.

2.1.2. DYNAMIC LOGIC

Apart from correctness, other aspects of programs, like termination or determinism, for example, require an extension to Hoare logic [26]. In *modal logic*, two modal operators, $\Box$ (necessarily) and $\Diamond$ (possibly) are introduced. $\Box p$ asserts that proposition p is necessarily true, and $\Diamond p$ asserts that proposition p is possible true. *Dynamic logic*, a form of modal logic, uses a slightly different notation for these modal operators, adding a reference to the program that is being executed. If it is necessarily the case that, starting from state p, the execution of the program q results in a state r in which the formula φ holds (i.e. $r \vDash \varphi$), we write:

$$p \vDash [q]\varphi.$$

If it is possibly the case that, starting from state p, the execution of the program q results in a state r in which the formula φ holds (i.e. $r \vDash \varphi$), we write:

$$p \vDash \langle q \rangle \varphi.$$

Dynamic logic thus enables us not only to express correctness, but also to express determinism, termination and equivalence. See the following examples, where π represents a program, φ represents an input condition, and ψ represents an output condition:

correctness:	$\varphi \rightarrow [\pi]\psi$
determinism:	$\langle \pi \rangle \varphi \rightarrow [\pi]\varphi$
termination:	$\langle \pi \rangle \mathit{TRUE}$
equivalence:	$[\pi_1]\varphi \leftrightarrow [\pi_2]\varphi$

2.1.3. SEQUENT CALCULUS

A sequent is an expression of the form $\varphi_1, ..., \varphi_n \circ \psi_1, ..., \psi_k$. The part left of the $\circ$ is referred to as the antecedent, and the right part is referred to as the consequent. Informally, $\varphi_1, ..., \varphi_n \circ \psi_1, ..., \psi_k$ corresponds to $\bigwedge_{i=1}^{n} \varphi_i \vDash \bigvee_{j=1}^{k} \psi_j$. In other words, if all the formulas in the antecedent are true, then at least one of the consequent formulas must be true.

Using sequent calculus, it can be shown that $\Sigma \vDash \Lambda$ by reducing the formulas in $\Sigma \circ \Lambda$, where Σ and Λ represent (possibly empty) sequences of formulas. Following a set of reduction rules, a sequent can be reduced to one or two other sequents, that can in turn also be reduced, et cetera, until reduction is no longer possible, resulting in a tree of sequents [26]. The tree can be considered *closed* if all its leave sequents close (meaning one of the formulas in the sequent is in the antecedent as well as in the consequent). If one ore more leave sequents do not close (none of the formulas in the sequent is in the antecedent as well as in the consequent, and reduction is not possible), the tree is *open*. Open sequents are considered counterexamples. The existence of such a counterexample proves that $\Sigma \nvDash \varphi$.

2.1.4. CONTRACTS

The use of JML enables us to specify contracts for classes and their clients (design by contract (DBC)) [8]. The contracts are specified in the program code itself. Preconditions define the obligations of the client, and postconditions define their rights. A precondition of a method specifies what must be true whenever a client calls it, and its postcondition specifies what must be true when it terminates. Designing software by using these contracts has several advantages. It prevents building in inefficient defensive checks, for example, because clear contracts eliminate their necessity. Also, DBC pre- and postconditions clearly show who is to blame when an error occurs. Furthermore, it supports modularity of reasoning, and allows for specifying intent without bothering about performance or other inessential implementation details.

Some of these characteristics are, of course, especially advantageous when contracts are designed before or during the design of the software. In our project, we designed the contracts based on software that was already written.

2.1.5. JAVA MODELLING LANGUAGE (JML)

According to Leavens *et al.*, JML is a "formal behavioral interface specification language for Java" [8]. It allows us to specify contracts in Java code, in the form of invariants, and pre- and postconditions. To get an impression of how a (simple) contract can be specified in the Java code using JML, let's consider a simple example of a method that we actually formally specified with JML in the `IdentityHashMap`, the `isEmpty` method (see listing 2.1). This method returns `true` if the `IdentityHashMap` contains no entries (i.e. its `size` field equals 0), or `false` in any other case (see line 7).

```
1 /*@ also
2   @   public normal_behavior
3   @     \ensures
4   @       \result <==> size == 0;
5   @*/
6 public /*@ strictly_pure @*/ boolean isEmpty() {
7   return size == 0;
8 }
```

Listing 2.1: The `isEmpty` method with JML specifications

The JML (placed between the `/*@` and `@*/` delimiters respectively) specifies a number of things. Firstly, the specification of a method must start with the keyword `also` (see line 1) if the method is already declared in the parent class (or interface), which is the case here. (The method overrides the `isEmpty` method implemented in the abstract `java.util.AbstractMap` class.) Generally speaking, the keyword `also` is intended to specify that a method specification is in addition to some specifications of the method that are given in the parent class or interface. The keyword `also` is also used to combine separate specification cases within a specification. Secondly, the keyword `normal_behavior`, on line 2, represents a so-called *heavyweight specification case*. For this example it suffices to know that it specifies the behaviour of the method under normal circumstances (i.e. no exception is expected to be thrown). Thirdly, the `\ensures` clause specifies a predicate that represents the postcondition of the method (lines 3 and 4). The `\result` keyword (line 4) represents the so-called *result expression* in JML for non-void methods. Its value is the value that must be returned, and its type is the return type of the method (in this case a boolean). This postcondition of the `isEmpty` method holds when it returns `true` if `size` equals 0, or `false` otherwise.

A class invariant is a property that must hold in all states of an object, except when control is inside the methods of its class. In other words, in the case of the aforementioned example, the class invariant of the `IdentityHashMap` must hold before and after the execution of `isEmpty`. Listing 2.2 shows (part of) a class invariant, stating that the array `table` must never be `null`, and must always have at least `MINIMUM_CAPACITY * 2` elements, and at most `MAXIMUM_CAPACITY * 2`.

```
1  public class IdentityHashMap
2      extends AbstractMap
3      implements Map, java.io.Serializable, Cloneable {
4
5    /*@ public invariant
6      @   table != null &&
7      @   MINIMUM_CAPACITY == 4 &&
8      @   DEFAULT_CAPACITY == 32 &&
9      @   MAXIMUM_CAPACITY == 536870912 &&
10     @   MINIMUM_CAPACITY * (\bigint)2 <= table.length  &&
11     @   MAXIMUM_CAPACITY * (\bigint)2 >= table.length;
12     ...
13     @*/
14
15   ...
16 }
```

Listing 2.2: Class invariant example

Aside from class invariants, JML also supports loop invariants. Loop invariants apply to for loops, while loops and do-while loops. They are properties that must hold at the beginning and at the end of a loop. Listing 2.3 shows an example of a contract for a simple while loop. The `maintaining` keyword implies that the following predicate should hold in every iteration of the loop (see line 1 and line 4). With every iteration, the result of a so-called *variant function*, immediately following the `decreasing` keyword (line 9), must decrease by at least one (1), and never become negative. This guarantees the termination of the loop.

```
1  /*@ maintaining
2    @   result / (\bigint)2 < minCapacity;
3    @
4    @ maintaining
5    @   (\exists \bigint i;
6    @       0 <= i < result;
7    @       \dl_pow(2,i) == result); // result is a power of two
8    @
9    @ decreasing
10   @   (minCapacity - result);
11   @*/
12 while (result < minCapacity)
13   result <<= 1;
```

Listing 2.3: Loop invariant example

Finally, it is also possible to write formal specifications for parts of methods, using block contracts. The basic idea is to divide a method into blocks, and to prove the correctness of each block independently ('divide and conquer'). As an example, listing 2.4 shows some fragments of the `put` method. On lines 11 – 36 a contract is specified for the block on lines 37 – 41. (The parts of the method that are irrelevant for this example are left out.) The contract specifies a number of preconditions (`\requires`). The first one states that `tab` and `k` must not be `null`, and that `i` is an even number within the boundaries of `tab`. The second precondition states that `k` does not yet exist in `tab` on any even position. (Indeed, on line 39, `k` will be added, and keys are required to be unique.) The next precondition states that `tab` is an array of `Objects`, and the final precondition specifies that `modCount` is within the bounds of the `Integer` type, and was not changed within the same method before this block. The two postconditions (`\ensures`) specify that `modCount` was updated,

and that `k` and `value` have been added to `tab` at positions `i` and `i + 1`, respectively.

```
public /*@ nullable @*/ Object put(Object key, Object value) {
  Object k =  maskNull(key);
  Object[] tab =  table;
  int len =  tab.length;
  int i =  hash(k, len);

  Object item;

  ...

  /*@ public normal_behavior
    @   requires
    @     tab != null &&
    @     i >= 0 && i < tab.length - 1 &&
    @     i % (\bigint)2 == 0 &&
    @     k != null;
    @   requires
    @     // The key does not yet exist in table
    @     (!(\exists \bigint n;
    @         0 <= n < tab.length - 1;
    @         n % 2 == 0 && tab[n] == k));
    @   requires
    @     \typeof(tab) == \type(Object[]);
    @   requires
    @     \dl_inInt(modCount) &&
    @     \old(modCount) == modCount;
    @   ensures
    @     tab[i] == k &&
    @     tab[i + 1] == value;
    @   ensures
    @     // modCount has changed (possibly overflowed, but that is not a problem)
    @     \old(modCount) != modCount &&
    @     \dl_inInt(modCount);
    @   assignable
    @     modCount, tab[i], tab[i+1];
    @*/
   {
     modCount++;
     tab[i] = k;
     tab[i + 1] = value;
   }

   ...
}
```

Listing 2.4: Block contract example

2.1.6. SYMBOLIC EXECUTION

Symbolic execution is a key concept in testing and formal analysis of software, introduced in the '70s of the previous century [27, 28]. Contrary to normal execution of a method, where the input parameters consist of concrete data, the idea behind symbolic execution is to use symbolic values rather than concrete data as input parameters. Also, values of program variables may be represented as symbolic expressions. Symbolic values and expressions can be regarded as classes of values, instead of single values, and, consequently, each symbolic execution result may be equivalent to a large number of normal test results. It can simultaneously explore multiple execution paths of a program, corresponding with classes of input values, whereas 'concrete' execution can only explore one single execution path at a time, based on the provided concrete input values. In other words, each symbolic

run may represent multiple concrete runs.

A symbolic executer maintains a tree of execution states. Every state is comprised of a 3-tuple (*stmt*, π, σ). The first element, *stmt*, represents the next line of code to be executed symbolically. The second, π, represents the *symbolic path constraint(s)* of a state. It is an accumulation of the constraints on the input that trigger the execution of the associated path. The value of π is initially *true*, meaning there is no constraint associated to the first statement of a program. The third element, σ represents a *symbolic store* which maps program variables to symbolic values or expressions.

Consider the example in listing 2.5. It shows a small method `f` that consists of several assignments and conditional statements. At the end of the program, there is an assertion that should hold for all possible runs of the method: the sum of `x`, `y` and `z` is not equal to 3. If the assertion fails, we can conclude an error occurred. The question is: which values of `a`, `b` and `c` will make the assertion fail? (Figure 2.1 on page 16 shows the symbolic execution tree for this example, with all the states labelled with a capital letter, and containing the values for *stmt*, π, and σ.)

In a concrete execution, concrete integer values would be assigned to the formal parameters `a`, `b` and `c`, when the function is called. But in a symbolic execution we assign the symbols α, β and γ to the parameters `a`, `b` and `c`, respectively (line 1). Right after this assignment, the state of the method is as follows (see also state A in the symbolic tree, shown in figure 2.1):

the next statement to execute, *stmt*:	2. `int x = 0, y = 0, z = 0`
the path constraint(s), π:	*true*
the symbolic store, σ:	$\{a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma\}$

The next statement is on line 2, where the value 0 is assigned to `x`, `y` and `z`. This changes σ to $\{a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma, x \mapsto 0, y \mapsto 0, z \mapsto 0\}$, and *stmt* to `3.if (a != 0)`. Because the method does not fork on line 2, no constraints need to be added to π. The resulting state is state B.

```
1  void f(int a, int b, int c) {
2    int x = 0, y = 0, z = 0;
3    if (a != 0) {
4      x = -2;
5    }
6    if (b < 5) {
7      if (a == 0 && c != 0) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```

Listing 2.5: Simple symbolic execution example method

The next statement (line 3) is conditional, and the flow of the rest of the method depends on the value of `a`. However, we do not know the concrete value of `a`, because the symbolic value α is assigned to it. There are two possibilities: either α != 0, or α == 0 (i.e. the condition is either *true* or *false*). Here, the symbolic execution is forked, and for each branch an execution state is created, state C and D, each with a different value for π, and a different value for *stmt*.

Following the path of C, a value of -2 is assigned to `x` (line 4), resulting in an update of σ in state E ($\texttt{x} \mapsto 2$). The next statement from state E is, again, an optional statement, depending on the value assigned to `b`, which is the symbolic value β. Again, the symbolic execution is forked. Two states are being created, F (with the constraint $\beta < 5$ being added to π), and G (with the constraint $\beta \geq 5$ being added to π).

The next statement to execute from state F is the conditional statement `if (a == 0 && c != 0)`. Note that, according to the symbolic path constraint π in state F, this condition can never be true. Indeed, $\alpha \neq 0 \wedge \alpha = 0$ can never be *true*. Here, the symbolic execution has uncovered an infeasible path, depicted by the grey-coloured leave in the symbolic tree. The alternative branch, however, is feasible and will lead to state H when $\alpha \neq 0 \vee \gamma = 0$ is *true*. Note that the path constraint in state H does not differ from the one in state F. This is because the conjunction of π in state F and $\alpha \neq 0 \vee \gamma = 0$ is equivalent with π in state F (i.e. $(\alpha \neq 0 \wedge \beta < 5) \wedge (\alpha \neq 0 \vee \gamma = 0) \iff (\alpha \neq 0 \wedge \beta < 5)$).

In the transition from state H to state I, the value 2 is assigned to the variable `z` (line 10 in listing 2.5, leading to a successful assertion on line 12. The green-coloured leave, originating from state I, shows that the assertion will not fail, because the sum of x + y + z = -2 + 0 + 2 is not equal to 3 whenever $\alpha \neq 0 \wedge \beta < 5$.

By analysing the all the branches in the symbolic execution tree, it becomes clear that the execution path {A, B, D, J, L, M, N} leads to a failing assertion on line 12 of the example method. The path constraints in state N contain the input values that cause an error: if `a` is 0, `b` is less than 5, and `c` is not 0, the assertion will fail. The symbolic execution has thus generated a set of test data, that can now be used to test the method.

Although the concept of symbolic execution was already introduced in the '70s of the previous century [27, 28], it did not become popular at the time, due to computer resource limitations. Programs can have lots of possible paths, and lots of states for every path. (In the example above, a method consisting of just a few lines of code, there are 16 states, divided over 6 paths. Imagine the number of possible paths and states a large program of hundreds or thousands of lines of code, containing (nested) loops as well as (nested) conditional statements would generate.) Memorizing program states may require a lot of memory. Computers lacked the processing power and memory in those days. However, in the 21st century, thanks to Moore's 'law', several symbolic execution tools have been developed and applied. Examples are SAGE (at Microsoft), Mergepoint (for Linux) and KeY (the tool we used in our project) for Java.

2.1.7. Proof obligations

From the contracts, specified in JML (see section 2.1.4, on page 10), proof obligations can be generated. A proof obligation is a formula stating that certain properties must hold for the contract of a method to be internally consistent. If a valid proof obligation can be proved, then the method it refers to is correct. It shows that the precondition in the contract implies that the postcondition holds after a method's execution [11].

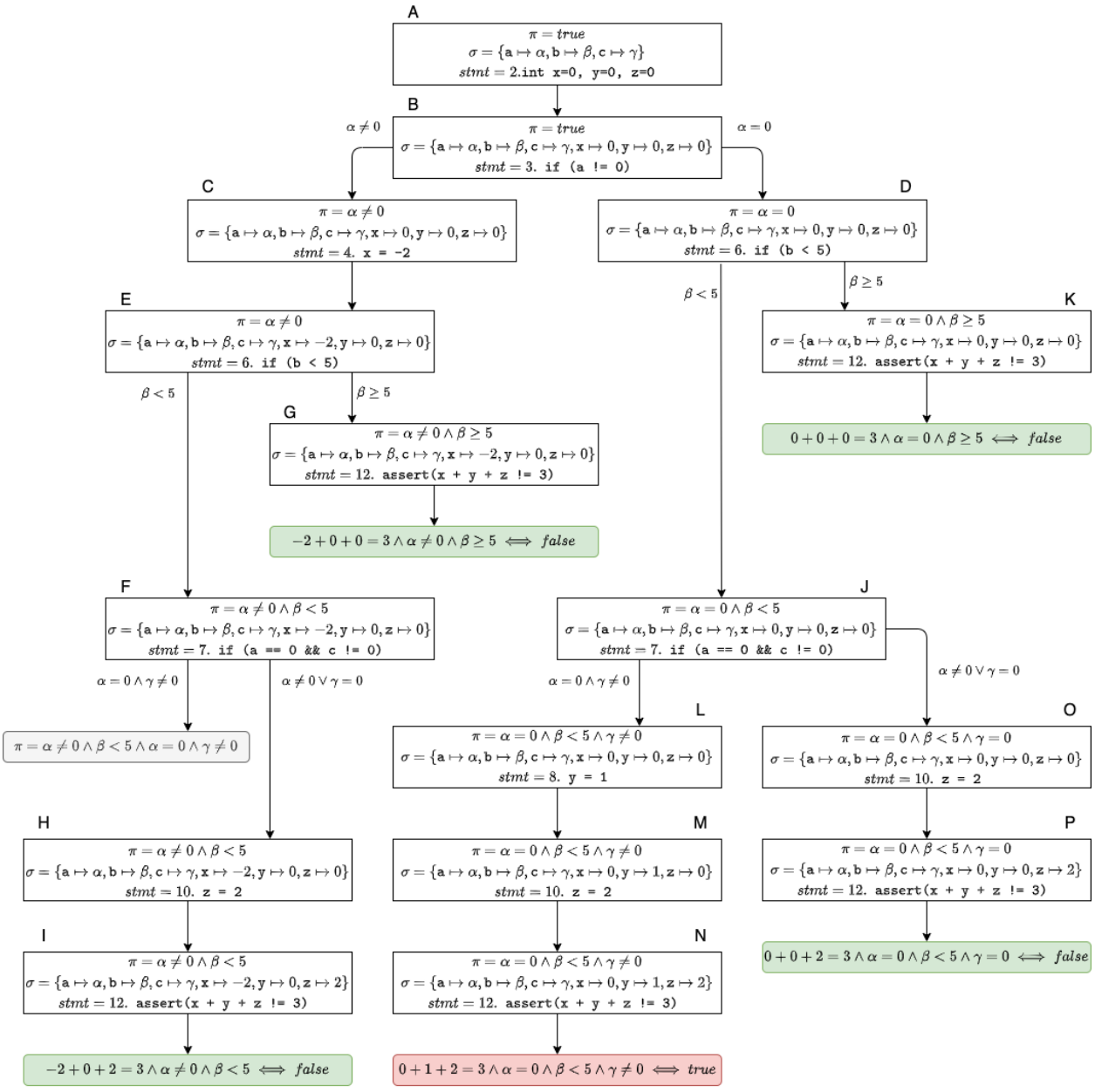


Figure 2.1: Symbolic execution tree of method `f`

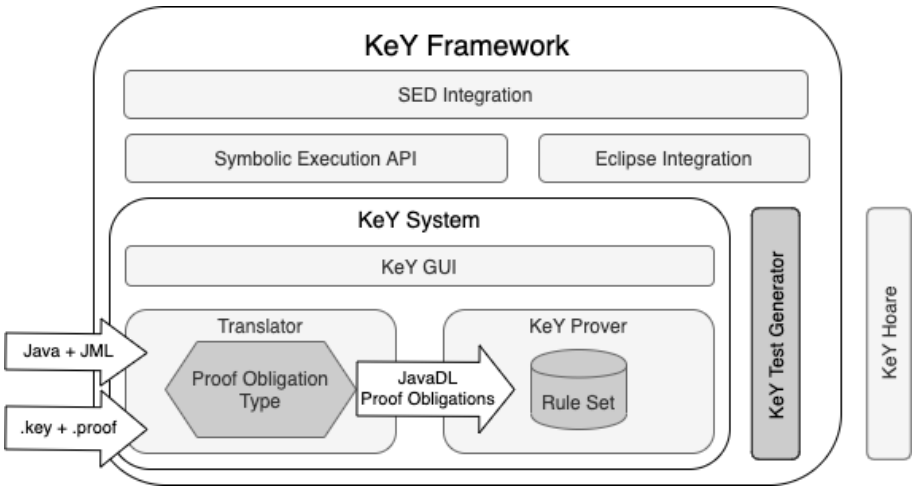


Figure 2.2: The KeY framework architecture

2.1.8. The KeY tool

The KeY tool, or KeY *system*, is a formal verification tool, developed by the KeY *project*[1], that originally started in 1998 [11]. The tool is suitable for Java (with some limitations), and uses JML as its formal specification language. It is part of a set of tools, together forming the KeY *framework* (see figure 2.2).

The KeY system combines the concepts described above. Based on contracts, specified with JML (2.1.4), it generates proof obligations (2.1.7), using a translator. These proof obligations are expressed in JavaDL (dynamic logic (2.1.2)). The KeY prover applies sequent rules (taclets [29]) to generate proof trees. Most of the proof rules in the rule set of the KeY prover symbolically execute (2.1.6) programs in JavaDL formulae.

Figure 2.3 shows the KeY GUI after automatically proving the correctness of the `isEmpty` method (see line 426 – 440 in the source panel on the right). On the left are the proofs panel (top) and the proof tree (bottom), and at the centre of the screen, the sequent panel is displayed.

[1] https://www.key-project.org/

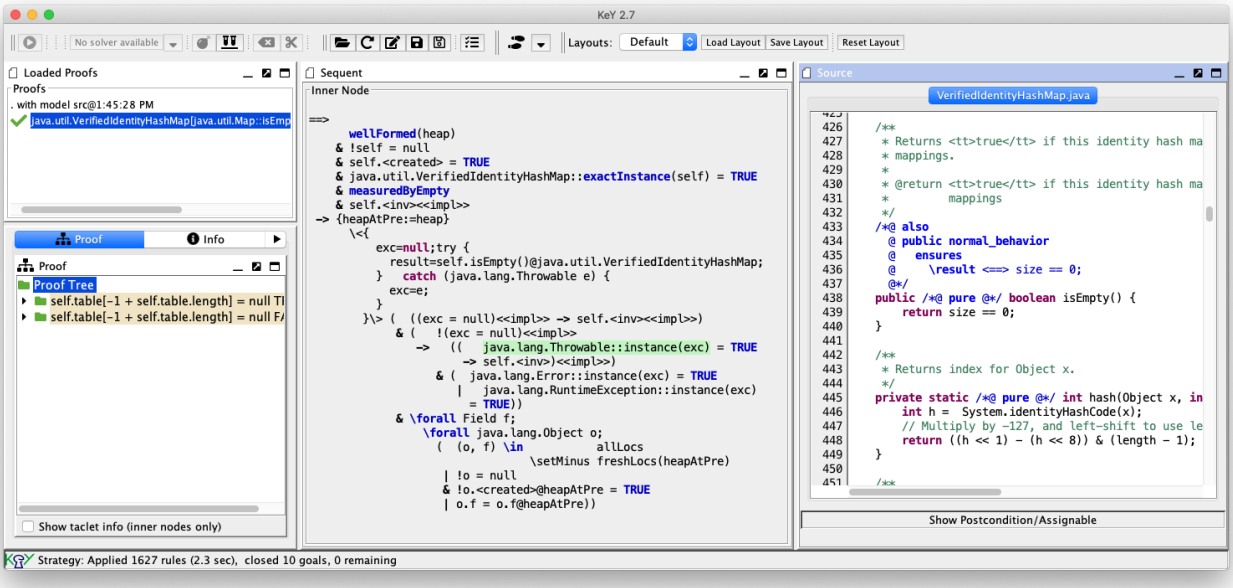


Figure 2.3: The KeY GUI

2.2. Research Questions

2.2.1. RQ1 – The main research question

In our research we focused on formally analysing a class in the OpenJDK library with JML and KeY). The central research question we aimed to answer was:

RQ1 : How can we formally analyse Java libraries with JML and KeY?

We broke down **RQ1** into the following subquestions:

RQ2 : Which Java classes are suitable candidates for formal analysis?
RQ3 : How can we limit the effort of formal specification?
RQ4 : What error(s), if any, can we identify in the CUA?
RQ5 : Can we provide a fixed version of the class that does stand the test of formal analysis?
RQ6 : What is the effort ratio when performing a formal analysis?

Subquestions **RQ2** to **RQ5** were technical questions, and were the main part of the research. **RQ6** can be regarded as an evaluation of the research, resulting in a report that is part of this thesis.

2.2.2. RQ2 – Which Java classes are suitable candidates for formal analysis?

The objective of the research was to formally analyse a class in the Collections Framework in the OpenJDK library. We composed a shortlist of suitable candidate classes, based on a number of characteristics. What would make a class a suitable candidate?

1. Originality. Ideally, a candidate class should not have been properly formally analysed before.

2. Furthermore, it would be interesting to find an error that had not been detected before. Hence, no bug should be reported for the ideal candidate.

3. Analysability. An important requirement was that the tooling used for specification and verification of the code, JML and the KeY-tool, had to be up to the task. Therefore, any candidate had to be suited to be loaded in KeY [2]. The first step in the research method would be to verify this for any of the candidate classes.

2.2.3. RQ3 – How can we limit the effort of formal specification?

Formal analysis, especially formal specification is tedious work (see table 1.1). During our research we wanted to explore ways to limit the amount of repetitious work that goes into formally specifying the CUA. We explored the possibilities of JUnit, JMLUnitNG, OpenJML, and JJBMC, for example, to perform preliminary sanity checks on our JML specifications. By sanity checks, we mean a quick way to check if the JML specifications are syntactically and/or semantically correct, before running a full blown formal verification with KeY.

2.2.4. RQ4 – What error(s), if any, can we identify in the class under analysis?

During the formal analysis procedure we hoped to encounter one or more software errors. Merely the identification of errors would not be sufficient. Proof had to be provided, as well as a solid test case for any detected error. Test cases would also have to provide a way of showing the correctness of a fixed version of the software (see research question **RQ5**).

2.2.5. RQ5 – Can we provide a fixed version of the class that does stand the test of formal analysis?

After detecting any error(s), a fixed version of the software would have to be designed. The fixed version of the software should be accompanied by a solid test case and should stand the test of formal analysis.

2.2.6. RQ6 – What is the effort ratio when performing a formal analysis?

Answering this question should give us some insight in the amount of effort necessary for formally analysing software, based on the chosen method and the chosen tooling. As part of the thesis, an experience report is provided (see section 4.5 (RQ6 – What is the effort ratio when performing a formal analysis?) on page 69).

[2] KeY had a few limitations. For example, the latest version of KeY at the start of the project (version 2.7) supported Java versions up to Java 1.4, so Generics (since J2SE 5.0), lambda expressions (since J2SE 8.0) and Java Reflection, for instance, were not supported. There was, however, a KeY plugin for Eclipse available that can 'strip' generics from the Java code. So, effectively, it was possible to verify classes up to and including version 7 of the OpenJDK library

2.3. RESEARCH PROCESS

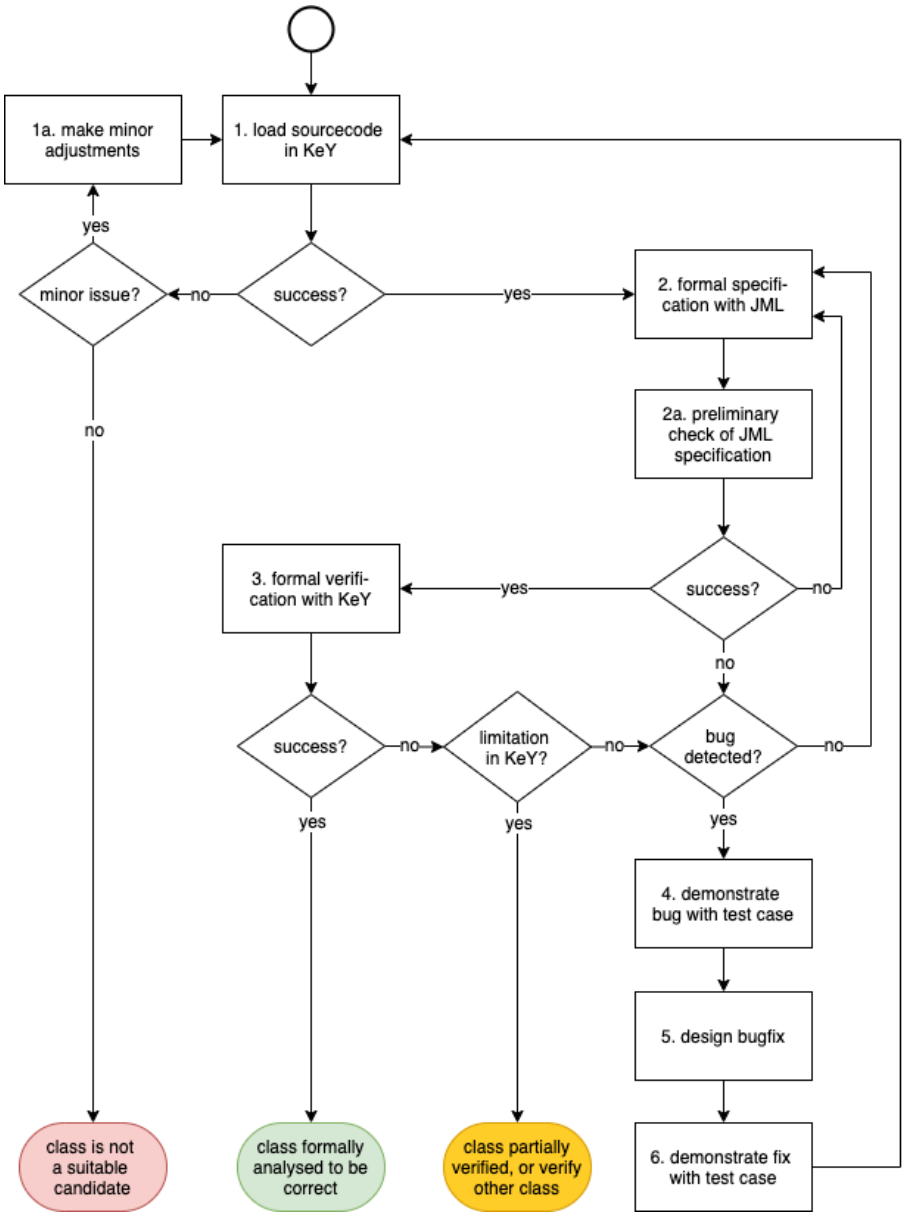


Figure 2.4: The research process

Figure 2.4 shows a flow diagram representing our research process:

- Steps 1/1a represent the iterative steps to find a suitable candidate class to formally analyse (see section 2.3.1 (Finding a suitable candidate for formal analysis) on page 21).
- Steps 2/2a represent the formal specification of the pre- and postconditions and invariants of the class and its methods, as well performing preliminary sanity checks of these specifications (see section 2.3.2 (Formal specification) on page 22).
- After these preliminary sanity checks, we should have a fairly good feeling about our specifications, and proceed to step 3, formal verification with KeY. Successful verifi-

cation would imply the specification is correct, and the class had been proven to be correct. Unsuccessful verification, on the other hand, might be due to one or more errors in the JML-specification (despite our attempts to eliminate those in step 2a).

- If, however the specification would be correct and complete, an error in the code may have been detected. Before concluding this is the case, it would be of importance to rule out any limitations or bugs in (or improper use of) KeY. Detection of a bug would result in steps 4 to 6, followed by another iteration of the process, starting at step 1.

2.3.1. Finding a suitable candidate for formal analysis

Stijn de Gouw, project supervisor and author of some of the related research articles [14, 19, 20] (see section 1.3, on page 7) suggested some promising candidate classes to formally analyse, that fit the preferences mentioned above (see subsection 2.2.2). Furthermore, bugs reported by the Java community also lead to interesting candidates to consider for formal analysis. This resulted in the following shortlist:

- IdentityHashMap – At the start of the project, there was no known research available for the class `java.util.IdentityHashMap`, and there is no known bug history [3]. This would make for an ideal subject class. Furthermore, KeY's Map theory supports the verification of maps, making this class a suitable candidate even more. There was one downside, however: its size. Formally specifying a class of this size would be a considerable amount of work.

- CountingSort – A less extensive candidate to formally specify was the `sort` method of `java.util.DualPivotQuicksort`. However, CountingSort was not expected to contain a bug, which made it a less interesting candidate [4].

- Classes containing a reported bug – A third option would be to take a class that had been reported to contain a bug that had not been fixed. The task, then, would be to formally prove the reported bug, and provide a fix that had to be be proven correct and tested successfully.

From this shortlist, a final candidate was picked, as part of the research process (see section 2.3, on page 20). We made an attempt to load the class in KeY [5]. In Eclipse (version 2020-03) we created a new KeY project (to be able do this, the KeY plugin for Eclipse had to be installed), and chose a project specific JRE: Java SE 7 [1.7.0_80][6]. We downloaded a

[3] https://bugs.openjdk.java.net did not contain any relevant open or closed bug in IdentityHashMap (checked on Feb. 14th, 2020).

[4] https://bugs.openjdk.java.net did not contain any open or closed bug in DualPivotQuickSort (checked on Feb. 14th, 2020).

[5] Initially, we used the 2.6.3 version of KeY, downloaded from KeY project website, http://key-project.org. During the project, we were granted access to the GitLab repository of the KeY project, and we cloned the latest master version from the repository (version 2.7). We also used the KeY plugin for Eclipse for JML syntax highlighting, KeY-related context menus, et cetera

[6] We experimented with several newer versions, but ran into problems when removing generic and generating stubs. KeY does not support generics, so these had to be stripped. The KeY plugin for Eclipse supports this automatically. Also, to verify a class in isolation, related classes have to be replaced by stubs. This, too, is supported by the KeY plugin.

JDK7 version of the `IdentityHashMap` [7] (see figure 2.5) and added it to the project, in the appropriate package. Next, we had to make the following (minor) adjustments:

1. Rename the class – to prevent the class name from clashing with the name of the original class in the JDK library in the same package, we renamed our class under analysis to `VerifiedIdentityHashMap`.

2. Generate stubs – to be able to analyse a class in isolation, all related classes (parent classes, implemented interfaces, return types, parameter types, et cetera) had to be stubbed. Stubs contain empty methods with default specifications (i.e. all preconditions as well as postcondition are *true*). The KeY plugin for eclipse enabled us to generate stubs automatically. It provides a context menu for the project with the menu item *Generate stubs*. Initially this failed, with two errors. Two minor adjustments in the code were necessary to fix this: two occurrences of `AbstractMap.SimpleEntry<>` (on lines 1148 and 1160) had to be changed to `AbstractMap.SimpleEntry<Object, Object>`. After this minor adjustment, the stubs were generated successfully.

3. Strip generics – because KeY does not support generics, we had to strip the generics in the `VerifiedIdentityHashMap`. This was also done automatically by choosing the *Strip generics* item from the project's context menu. Stripping the generics resulted in a trivial problem, that was easily fixed. The resulting code contained the following on line 504 and 505 (the `putAll` method):

```
504  for (Entry e: m.entrySet())
505    put(e.getKey(), e.getValue());
```

Listing 2.6: Fragment of the `IdentityHashMap` after stripping generics

Since `m.entrySet()` now returned an `Object` instead of an `Entry`, an explicit cast is necessary to satisfy the compiler. This resulted in:

```
504  for (Object o: m.entrySet()) {
505    Entry e = (Entry) o;
506    put(e.getKey(), e.getValue());
507  }
```

Listing 2.7: Fragment of the `IdentityHashMap` after correction

After these minor adjustments we were able to load the `VerifiedIdentityHashMap` in KeY. This meant we had found a suitable candidate class to analyse, and we could start formally specifying the class with JML. In the remainder of this thesis, this class will be referred to as `IdentityHashMap`, the original name of the class.

2.3.2. Formal specification

We based the JML specifications we wrote for the `IdentityHashMap` on the Javadoc inside the code, combined with general knowledge of data structures (Map, Set, Iterator, Array

[7] http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/9b8c96f96a0f/src/share/classes/java/util/IdentityHashMap.java

et cetera) [30]. This way of working does not instantly guarantee 100% error-free specifications. Formal analysis is an iterative process, where errors in the specification may not surface until after time consuming attempts to formally verify the code with KeY. Furthermore, as is clear from the UML diagram in figure 2.5, the `IdentityHashMap` is quite a large class to formally specify. Since formal specification takes up the lion's share of effort in the process of formal analysis [14, 20], we wanted to detect errors in our JML specification as early as possible. By detecting errors early on, we aimed to limit the effort spent in the process. We therefore decided to perform a number of *sanity checks*. In the sections below we describe how we applied JUnit, as well as some available JML tooling we explored, JMLUnitNG, OpenJML, and JJBMC, with varying success.

Preliminary validation of JML specifications with JUnit

In subsection 2.1.5 (Java Modelling Language (JML)), on page 11, we already showed an example of a method we actually formally specified with JML in the `IdentityHashMap`, the `isEmpty` method. Let's return to that example, as showed in listing 2.1, and see how we tested the JML specification of that method, using JUnit. The `ensures` clause (lines 3 – 4) specifies a predicate that represents the postcondition of the method: the return value must be `true` if `size` equals 0, or `false` otherwise. Furthermore, the method is defined `strictly_pure` (see line 6). The JML Reference Manual by Leavens *et al.* [9] explains (strictly) pure methods as follows:

> A pure method that is not a constructor implicitly has a specification that does not allow any side effects. That is, its specification has the clauses
>
> ```
> diverges false;
> assignable \nothing;
> ```
>
> added to each specification case;

The first clause (`diverges false;`) means the method may neither contain an infinite loop, nor return abnormally (abort). (The `diverges` clause is seldom used, and its default value is `false`). The second clause is more interesting, and can be tested in a unit test. By specifying `assignable \nothing` it is stated that no field or variable that is defined outside the method may be changed during its execution. According to the The JML Reference Manual by Leavens *et al.* [9]:

> An assignable clause gives a frame axiom for a specification. It says that, from the client's point of view, only the locations named, and locations in the data groups associated with these locations, can be assigned to during the execution of the method.

In case of the `isEmpty` method, this means that no field of the `IdentityHashMap` or any field of other objects is allowed to be changed.

Aside from the method's postcondition (1) and its purity (2), the class invariant of the `IdentityHashMap` (as well as its inner classes) must hold before and after the invocation of the method `isEmpty` (3). These three aspects of the JML specification, we were able to test with JUnit. For this particular method, we designed the `IdentityHashMapIsEmptyTest`[8]

[8] The complete set of unit test classes developed during our project is part of the project deliverables, and is available on GitHub: https://github.com/m4ndeb2r/IM9906-IdentityHashMapSpecTester.

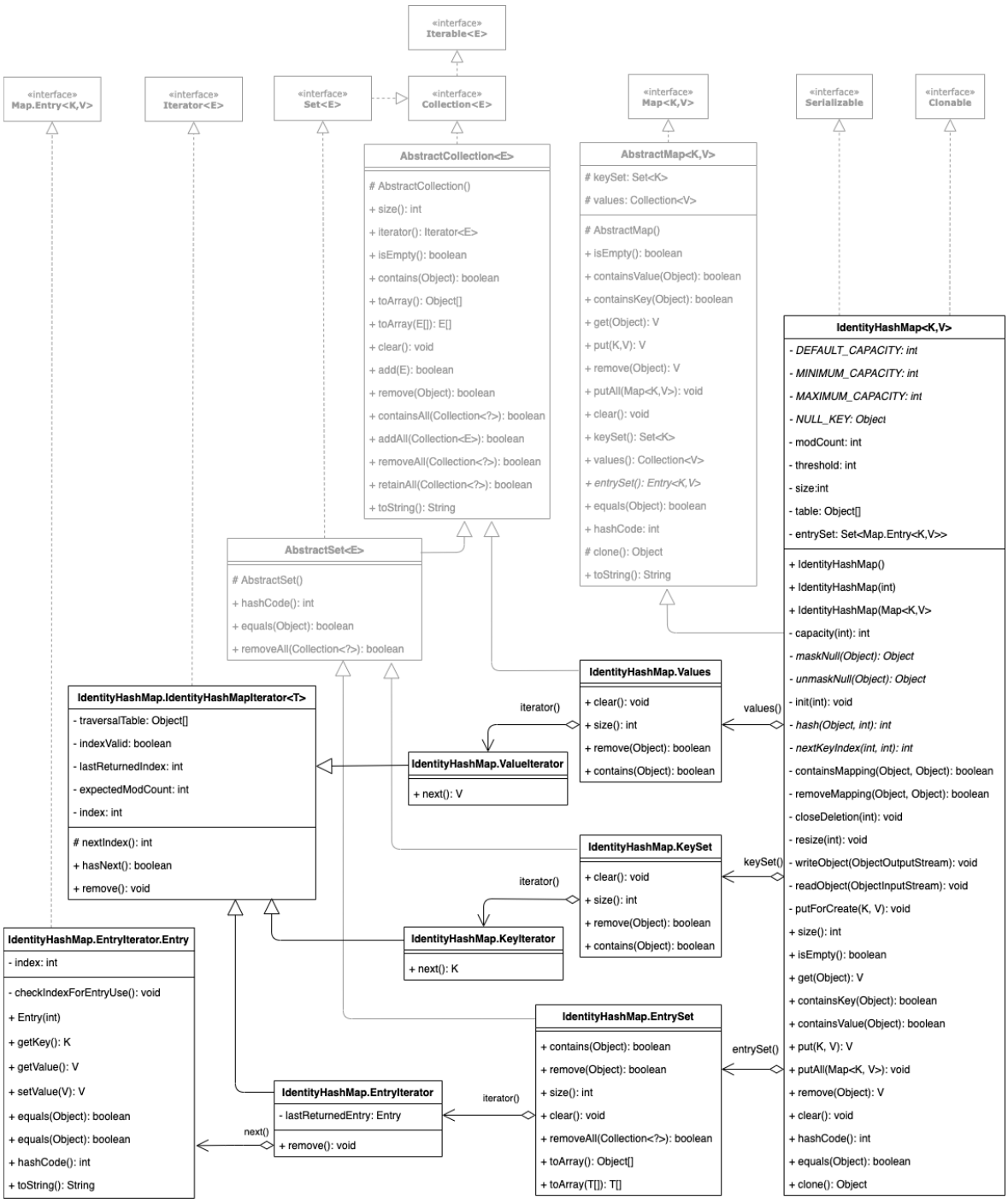


Figure 2.5: A UML diagram of the IdentityHashMap

class, containing a test method `testIsEmptyNormalBehaviour` that tests the aforementioned aspects. For simplicity, we will leave aside the details regarding class invariant testing (3), and focus on the postcondition (1) and the purity (2) of the method `isEmpty` here (see listing 2.1 on page 11 for the `isEmpty` method including the JML specification, and listing 2.8 below, for the unit test code).

```
1  import org.junit.Test;
2  import java.util.IdentityHashMap;
3  import static nl.ou.im9906.ClassInvariantTestHelper.assertClassInvariants;
4  import static nl.ou.im9906.MethodTestHelper.assertIsPureMethod;
5  import static nl.ou.im9906.ReflectionUtils.getValueByFieldName;
6  import static org.hamcrest.MatcherAssert.assertThat;
7  import static org.hamcrest.core.Is.is;
8
9  public class IdentityHashMapIsEmptyTest {
10
11   @Test
12   public void testIsEmptyNormalBehaviour()
13       throws NoSuchFieldException, IllegalAccessException,
14       NoSuchMethodException, NoSuchClassException {
15
16     // Create an empty map, test pre- and postconditions
17     // Precondition: class invariants hold
18     // Postcondition 1: ensures \result <==> size == 0;
19     // Postcondition 2: class invariants hold
20     // Test if the isEmpty method is pure in these circumstances
21     final IdentityHashMap<Object, Object> emptyMap = new IdentityHashMap<>();
22     assertClassInvariants(emptyMap);
23     assertThat((int) getValueByFieldName(emptyMap, "size"), is(0));
24     assertThat(emptyMap.isEmpty(), is(true));
25     assertClassInvariants(emptyMap);
26     assertIsPureMethod(emptyMap, "isEmpty");
27
28     // Create a map, and add an element to the map, and test pre- and
29     // postconditions, and purity again
30     final IdentityHashMap<Object, Object> filledMap = new IdentityHashMap<>();
31     filledMap.put("key1", "value1");
32     assertClassInvariants(filledMap);
33     assertThat((int) getValueByFieldName(filledMap, "size"), is(1));
34     assertThat(filledMap.isEmpty(), is(false));
35     assertClassInvariants(filledMap);
36
37     // Remove the element, and test postcondition again
38     filledMap.remove("key1");
39     assertThat((int) getValueByFieldName(filledMap, "size"), is(0));
40     assertThat(filledMap.isEmpty(), is(true));
41     assertClassInvariants(filledMap);
42   }
43 }
```

Listing 2.8: Using JUnit to test the JML specifications of `isEmpty`

On line 21 an `IdentityHashMap` is declared and instantiated as an empty map. As soon as the map is instantiated, its class invariant and the class invariants of inner classes should hold. This is being checked on line 22 by calling the `assertClassInvariants` method. (This helper method checks if the class invariant of the `IdentityHashMap` and its inner classes hold. Again, for simplicity reasons, the details of this method are ignored here.)

On lines 23 and 24 we assert that the `\ensures` clause `\result <==> size == 0` (see listing 2.1) holds: if `size` equals 0 (which should obviously be the case at this point), then the `\result` must be `true`. On line 25, the class invariant of the `IdentityHashMap` and its inner classes must hold again, and on line 26 the purity of the method `isEmpty` is being tested, by calling the helper method `assertIsPureMethod`, which is imported on line 4. On lines 30 – 35 the exercise on lines 21 – 25 is repeated for an `IdentityHashMap` that is not empty. Here, we expect a `size` equal to 1, and the result of the method must, therefore, be `false`. Finally, on lines 38 – 41, we remove the entry from `filledMap`, and check if `size` contains the value 0 again, and the result of `isEmpty` returns `false`, as expected.

The assertion on line 26 tests the purity of the method, and is worthwhile to take a

closer look at. The helper method `assertIsPureMethod`, which is imported on line 4, is implemented as follows:

```
1  protected static void assertIsPureMethod(
2          Object obj,
3          String methodName,
4          Object... params)
5          throws NoSuchFieldException,
6                 IllegalAccessException,
7                 NoSuchMethodException {
8    assertAssignableNothingClause(obj, methodName, params);
9  }
10
11 ...
12
13 protected static void assertAssignableNothingClause(
14         Object obj,
15         String methodName,
16         Object[] params)
17         throws NoSuchMethodException,
18                IllegalAccessException,
19                NoSuchFieldException {
20   assertAssignableClause(obj, methodName, params, new String[0]);
21 }
```

Listing 2.9: Testing if a method is pure, using JUnit

The method `assertIsPureMethod` expects three parameters. The first, `obj` can be any kind of object. In our example, the actual parameter will obviously be an instance of the `IdentityHashMap`, our CUA. The second one is the name of the method for which we want to test if it is pure (i.e. it does not change any field of any object not defined or created locally inside itself). In this example this actual parameter will have the value "isEmpty" (see listing 2.8, line 26). The third parameter is the list of parameters to pass on to the `isEmpty` method (an empty list in this case, because the `isEmpty` method has no formal parameters). Listing 2.9 shows that the implementation of `assertIsPureMethod` is coherent with the definition in the JML Reference Manual by Leavens *et al.* [9], where it is stated that a pure method implicitly has a specification of `assignable \nothing;`. In our implementation (line 8) we call the method `assertAssignableNothingClause`, which in turn delegates the real work (asserting no field or parameter is being assigned) to the method `assertAssignableClause` (line 20). `assertAssignableClause` is expecting one extra parameter: an array of String values (representing the names of the fields in the `IdentityHashMap` that are assignable according to the JML specifications). In case of a pure method, no field is assignable. Therefore, the fourth actual parameter is an empty array.

The real testing is done in the method `assertAssignableClause` (see listing 2.10), that iterates over all fields in the `IdentityHashMap`, and stores the original values of all these fields (lines 10 – 24). Note that final fields are ignored, because they cannot be assigned anyway (line 19). Also, assignable fields are ignored (lines 20 and 79 – 86), because only the fields that are *not* allowed to be changed have to be checked. Next, it invokes the method under analysis (line 28). Note that, on lines 29 – 34, any `InvocationTargetException` is caught and ignored. The reason for this is that we wanted to be able to handle exceptional behaviour for methods that are not allowed to assign values to any field. (Consider, for example, in JML terms: an `assignable \nothing` clause within an `exceptional_behavior`

heavy weight specification case). Finally, the new values of all the fields that were not marked as final or assignable are collected, and compared with the original values. The original values must be unchanged.

```
protected static void assertAssignableClause(
        final Object obj,
        final String methodName,
        final Object[] params,
        final String[] assignableFieldNames)
        throws NoSuchFieldException,
               IllegalAccessException,
               NoSuchMethodException {

  // Collect the non-assignable fields from the IdentityHashMap, their names,
  // and their original values, before invoking the method under analysis.
  final Field[] fields = obj.getClass().getDeclaredFields();
  final Map<String, Object> oldFieldValues = new HashMap<>();
  for (int i = 0; i < fields.length; i++) {
    // Skip final fields (because they cannot be assigned anyway) as
    // well as the assignable fields (because we do not have to check
    // these).
    final String fieldName = fields[i].getName();
    if (!isFinal(obj, fieldName) &&
        !arrayContains(assignableFieldNames, fieldName)) {
      final Object fieldValue = getValueByFieldName(obj, fieldName);
      oldFieldValues.put(fieldName, fieldValue);
    }
  }

  // Now, invoke the method under analysis.
  try {
    invokeMethodWithParams(obj, methodName, params);
  } catch (InvocationTargetException e) {
    // This might be due to an Exception that is expected in the
    // exceptional_behavior heavy weight specification case of the JML.
    // We still want to check the JML assignable clause. So, let's do
    // nothing, and resume to check the fields and parameters.
  }

  // Check if the fields have not been unexpectedly assigned a value.
  // I.e. (according to our 'loose' interpretation of the term 'assignable')
  // compare the old value with the current value.
  for (String fieldName : oldFieldValues.keySet()) {
    final Object newFieldValue = getValueByFieldName(obj, fieldName);
    final Object oldFieldValue = oldFieldValues.get(fieldName);
    if (isPrimitive(obj, fieldName)) {
      // In case of a primitive field, we cannot use the '==' operator,
      // because getValuesByFieldName returns an object representation
      // of the actual reference to the respective field. We, therefore,
      // use Matchers.is()
      assertOldEqualsNewPrimitive(fieldName, newFieldValue, oldFieldValue);
    } else {
      // In case of a non-primitive field, we can use the '==' operator,
      // because getValuesByFieldName returns the actual reference to the
      // respective object.
      assertOldSameAsNewNonPrimitive(fieldName, newFieldValue, oldFieldValue);
    }
  }
}

private static void assertOldSameAsNewNonPrimitive(
        final String fieldName,
        final Object newFieldValue,
        final Object oldFieldValue) {
  final String msg = String.format(
          "Non-primitive, non-assignable field '%s' unexpectedly assigned.",
          fieldName
  );
```

```
65    assertThat(msg, newFieldValue == oldFieldValue, is(true));
66  }
67
68  private static void assertOldEqualsNewPrimitive(
69          final String fieldName,
70          final Object newFieldValue,
71          final Object oldFieldValue) {
72    final String msg = String.format(
73            "Primitive, non-assignable field '%s' unexpectedly changed.",
74            fieldName
75    );
76    assertThat(msg, newFieldValue, is(oldFieldValue));
77  }
78
79  private static <T> boolean arrayContains(final T[] array, final T value) {
80    for (final T element : array) {
81      if (element == value || value != null && value.equals(element)) {
82        return true;
83      }
84    }
85    return false;
86  }
```

Listing 2.10: Testing the assignable clause, using JUnit

Note that, on line 42 (and 48), we made a distinction between primitive fields and non-primitive fields. We did this for the following reason. When using Java Reflection to access private fields (as is done by the helper method `getValueByFieldName`, that at some point calls Java Reflection's `Field.get` method to retrieve the value of a field), an `Object` is returned. If the field is actually a primitive (`int`, `long`, `boolean`, et cetera), it is being converted to an object, using a wrapper class (`Integer`, `Long`, `Boolean`, et cetera, respectively). Indeed, every call to `getValueByFieldName` returns a newly created instance of such a class. As a consequence, we are not able to successfully compare the old value of a primitive field to the new value of that same primitive field using the '==' operator. It will always return *false*. Therefore, we compare primitives using the `Matcher.is` method (see line 76, inside the `assertOldEqualsNewPrimitive` method on lines 68 – 77). This is, however not the case for non-primitive fields: `getValueByFieldName` will return a reference to the same object every time it is invoked. Therefore, the values of non-primitive fields can be compared using the '==' operator (see method `assertOldSameAsNewNonPrimitive` on lines 57 – 66).

A few questions might arise from reading this implementation, that require to be addressed here. For example, which fields are included in the assignable test (lines 12 – 24, 39 – 54)? And why aren't parameters included?

1. Which fields are included in the assignable test? Although, strictly, we should consider the complete heap, for pragmatic reasons we limit the test to fields of the specific hash map under analysis. On line 12 in listing 2.10 all the declared fields of the `IdentityHashMap` are retrieved. This includes `static` fields. An argument could be made to skip `static` fields, but we wanted to make absolutely sure that no side effects would occur. Indeed, `static` fields might be assigned a value, and if this would happen, we would want it to be detected. Therefore, we decided it would be better to be blunt, and to include all declared fields of the class in our test. Except, of course, the fields that are marked assignable (contained in the input parameter `assignableFieldNames` on line 5), and that are excluded on line 20. Final fields are

also excluded, obviously, because it is not allowed to assign new values to final fields once they are initialised in Java (see line 19).

2. Why aren't parameters included? Since Java only supports the copy-in parameter mechanism [31], there is no need to worry about side effects regarding parameters. Therefore, we decided not to include them in our test.

The purpose of this section has been to explain our method of using JUnit tests combined with Java Reflection to check JML specifications. The results of these are addressed in chapter 3 (Results) and further discussed in chapter 4 (Discussion).

Preliminary validation of JML specifications with JMLUnitNG

Although JUnit tests combined with Java Reflection had their merits, it must be noted, they also had some (technical) limitations. For example, JML contracts had to be manually translated to Java in the JUnit tests. We were, therefore, not able to detect syntax errors in the original JML specifications, and there was, obviously, the possibility of translation errors. We will discuss the downsides of the approach taken above with JUnit in detail in section 4.2.1 (Pros and cons of validating JML specifications with JUnit) on page 66.

In an attempt to tackle the downsides of the approach taken above with JUnit, we decided to also try to use JMLUnitNG [32]. JMLUnitNG is a TestNG-based successor to JMLUnit, a unit testing framework for JML-annotated Java code [23]. JMLUnit suffers from a number of shortcomings, like limited test coverage, excessive memory utilisation and the need to manually write extra code to generate test data objects. JMLUnitNG reduces a number of these issues.

After installing JMLUnitNG (the latest version dates back to 2014 [9]), we ran into issues, however. We were able to resolve these issues, but while resolving them, we discovered the tool was not supported anymore. Because OpenJML provides similar functionality, we decided to place our bets on the latter.

Preliminary validation of JML specifications with OpenJML

OpenJML [22] is an automatic verification tool for JML annotated Java programs. It transforms a program into a static single assignment form, and from this, it generates first-order logic conditions, that are the input for a so-called satisfiability modulo theory (SMT) solver. OpenJML is available as an Eclipse plugin and as a command line tool. We decided to use the latter.

Whereas OpenJML is an automatic verification tool, KeY is an interactive tool. The former allows for automatically generated verification conditions (based on the annotated program) to be sent to a first-order prover. Verification can be done very fast for less complex methods where correct specifications can be given directly. It is, however, less suitable for incremental development of a specification. KeY, on the other hand, allows the user to build up more complicated proofs incrementally. As Boerman *et al.*[12] have observed, OpenJML is not so much suited for preliminary testing of *all* the JML specifications, but because of the aforementioned difference, "[...] there is a high potential to increase verification efficiency if a user can smoothly switch between OpenJML and KeY during the verification process." And this is how we actually applied OpenJML: by switching between both tools, and enhancing the JML step by step. The results of using OpenJML are described in chapter 3 (Results) and are further discussed in chapter 4 (Discussion).

[9] Available on http://insttech.secretninjaformalmethods.org/software/jmlunitng/

Preliminary validation of JML specifications with JJBMC

A third tool we used for preliminary checking of our JML specifications was JJBMC. JJBMC is a tool that is still being developed at both the Forschungszentrum Informatik (FZI) and the Karlsruher Institut für Technologie (KIT). It enables a software bounded model checker, JBMC, to verify contracts written in JML, based on OpenJML. We used JJBMC to perform sanity checks on our JML specifications of the `IdentityHashMap`.

Because JBMC is a bounded model checker, we expected some adjustments to the code would be necessary to prevent the typical problem of state space explosion, that is common with model checking (see 1.1.6, Formal analysis methods and the case for deductive verification, and appendix A, section A.1, Model checking). Nevertheless, we wanted to explore if the tool could help us to speed up the process of formal specification. The results of applying JJBMC for this purpose can be found in chapter 3 (Results) and are further discussed in chapter 4 (Discussion).

3

RESULTS

In this chapter we will describe the results obtained during our project of formally analysing the `IdentityHashMap` of the Java Collections Framework (JDK7). In section 3.1 the implementation of the `IdentityHashMap` is roughly outlined. Section 3.2 describes how we prepared the CUA for verification with KeY, and which methods we were eventually able to verify. It also shows some proof statistics that are briefly elaborated on. Finally, we describe a specific example of a method we verified, the `containsKey` method.

Next, in sections 3.3, 3.4 and 3.5 we describe the results of our attempts to speed up the process of formal analysis. We took a hybrid approach to specify the CUA, and used a number of tools (as mentioned in chapter 2) to perform some preliminary checks on our specification, in an attempt to detect any errors or other shortcomings in an early stage of the process.

The final two sections of this chapter, section 3.6 and 3.7, go into the overflow error we detected in the `capacity` method. We explain the error, what triggers it, the damage it causes when triggered, and how it could be solved.

3.1. IMPLEMENTATION OF IDENTITYHASHMAP

Figure 2.5 on page 24 shows a UML-diagram of the complete `IdentityHashMap`, including its inner classes. Here we will describe the main methods and the inner structure of the main class itself. The purpose of this section is to provide enough insight into the inner workings of the CUA to grasp the remainder of this chapter.

The `IdentityHashMap` implements the `java.util.Map` interface of the Java Collections Framework, using a hash table. Like in any `Map`, any entry consists of a key-value pair (k, v). In the `IdentityHashMap` implementation, two keys k_1 and k_2 are considered equal if and only if $k_1 == k_2$ (reference-equality). This is different from normal implementations (e.g. `HashMap`) that use object-equality.

The entries are stored in a hash table (a private array field named `table`). When an entry (k, v) is added (using the `put` method), a hash h is calculated based on the value of k and the length of `table`, N, where $h \in \{0, 2, 4, ...N\}$. The key k is stored in `table` at index h, and the value v is stored at index h+1. However, there is no guarantee that the hash function produces a unique hash value for any key. In other words, collisions might occur: the calculated position in the hash table is already taken by a previously added entry with a different key. In that case, the new entry will be stored at the next position in the table. If that

position is taken as well, the next position is tried, et cetera, until a free position is found. The next position in `table` to store a key is determined by the method `nextKeyIndex`. This method returns $(i + 2)$ mod N (where i is the current key index and N is the length of `table`). This way of handling collisions is called *linear probing* [30]. Obviously, when an entry $(k_{existing}, v_{new})$, with a key $k_{existing}$ that is already present in the map, is added to the map, it is not considered a collision as such, and the value of the existing entry will be overwritten with v_{new}. This guarantees that all keys in the map are, at all times, unique.

It is allowed to put an entry (k, v) into the `IdentityHashMap`, where k is `null` (v can be any Object value, including `null`). However, to be able to distinguish such an entry from an empty entry, and to guarantee the uniqueness of keys, k is mapped to a constant `NULL_KEY`. This constant is declared and initialised as follows:

```
1  private /*@ spec_public @*/ static final Object NULL_KEY =  new Object();
```

Listing 3.1: Constant `NULL_KEY`, a placeholder for an empty key.

The `maskNull` and `unmaskNull` methods map a key (if necessary) from `null` to `NULL_KEY` and vice versa, respectively (see listing 3.2). These methods are typically used when storing entries (e.g. `put`), searching entries by their keys (e.g. `get`, `containsKey`, `containsMapping`), or removing entries (e.g. `remove`).

```
1  /**
2   * Use NULL_KEY for key if it is null.
3   */
4  /*@ private normal_behavior
5    @   ensures key == null ==> \result == NULL_KEY;
6    @   ensures key != null ==> \result == key;
7    @*/
8  public static /*@ strictly_pure @*/ Object maskNull(Object key) {
9    return (key == null ? NULL_KEY : key);
10 }
11
12 /**
13  * Returns internal representation of null key back to caller as null.
14  */
15 /*@ private normal_behavior
16   @   ensures key == NULL_KEY ==> \result == null;
17   @   ensures key != NULL_KEY ==> \result == key;
18   @*/
19 private /*@ spec_public @*/ static /*@ pure nullable @*/ Object unmaskNull(Object
       key) {
20   return (key == NULL_KEY ? null : key);
21 }
```

Listing 3.2: Masking and unmasking `null` keys.

When the `get` method is called to get an entry from the `IdentityHashMap`, the hash method is used again to determine the position of the entry in the array `table`. If the key found at this position does not equal the key of the requested entry (reference-equality!), then the next key index is tried, until the key of the requested entry is found, or an empty element in the array is encountered. Note that it is crucial that the array, at any time, contains at least one empty entry. If this would not be the case, it is possible that the `get` method

will go into an infinite loop if the requested entry is not found [1]. Indeed, the `nextKeyIndex` method, that determines the next key index, returns $(i+2) \bmod N$ (see above).

After removal of an entry (method `remove`) the array should be restored as if the entry was never added in the first place. If an entry (k, v) with k at index i and v at index i+1 is removed, then all of the subsequent entries with the same hash have to be shifted down two positions. The `closeDeletion` method is responsible for keeping all the entries that have the same hash for their key, together as a consecutive sequence, without any gaps between them, just like it would have been when the deleted entry was never added. This is crucial for methods like `get`, `containsKey`, `containsMapping`, et cetera, to function correctly. Indeed, these methods search the array `table`, starting at a position determined by the requested key's hash value, and search every subsequent key index until that key is found or an empty entry is encountered. Unclosed gaps, therefore, would have a fatal impact.

3.2. JML CONTRACTS AND KEY PROOF FILES

3.2.1. PREPARATION OF THE CLASS UNDER ANALYSIS

As described in section 2.3.1 (Finding a suitable candidate for formal analysis) of chapter 2 (see page 21), we started off by preparing the `IdentityHashMap` to be able to load it in KeY, which was a prerequisite for a suitable candidate class for formal analysis. The adjustments to the class were minimal: we renamed it to `VerifiedIdentityHashMap` to prevent it from clashing with the original class in the same package, generated stubs for the related classes, and stripped generics from the class because KeY does not support them. Furthermore, two minor adjustments were made in the code by hand (having to do with the generation of stubs, and a casting problem after stripping the generics), and we were ready to start writing our first JML contract.

3.2.2. SPECIFIED AND VERIFIED METHODS

Because the `IdentityHashMap` is considerably large (see figure 2.5 on page 24) and its semantics intricate, we focused mainly on the methods of the `IdentityHashMap` itself and disregarded the inner classes `EntryIterator`, `EntrySet`, `IdentityHashMapIterator`, `KeyIterator`, `KeySet`, `ValueIterator` and `Values`. Furthermore, we limited formal verification to the most typical methods for a map, i.e. `isEmpty`, `size`, `get`, `containsKey`, `containsValue`, `containsMapping`, et cetera. Table 3.1 shows all methods of the CUA. The check marks in the columns JML and KeY indicate which methods were specified and which methods were proven with KeY during our project [2] [3]. (Some methods were not or incompletely specified, and also not verified, e.g. `closeDeletion`, `equals`, and `hashCode`. These methods are nevertheless included in the table, to give an impression of the level of specification.)

[1] This is also applies to other methods, like `containsKey`, `containsMapping`, and `containsValue`, that all search the array in a similar way.

[2] A complete list of specified, tested and verified methods and inner classes of the `IdentityHashMap`, subdivided per tool, is included in appendix C.

[3] All proof files generated by KeY are part of the project deliverables, and are also available in GitHub. See: https://github.com/m4ndeb2r/IM9906-VerifyingIdentityHashMap

Method	JML	KeY
Class invariant	✓	✓
Object maskNull(Object)	✓	✓
Object unmaskNull(Object)	✓	✓
IdentityHashMap()	✓	
IdentityHashMap(int)	✓	
int capacity(int)	✓	✓
void init(int)	✓	✓
IdentityHashMap(Map<K,V>)	✓	
int size()	✓	✓
boolean isEmpty()	✓	✓
int hash(Object, int)		✓ [4]
int nextKeyIndex(int, int)	✓	✓
V get(Object)	✓	✓
boolean containsKey(Object)	✓	✓
boolean containsValue(Object)	✓	✓
boolean containsMapping(Object, Object)	✓	✓
V put(K, V)	✓	✓
void resize(int)	✓	✓
void putAll(Map<K,V>)	✓	
V remove(Object)	✓	
boolean removeMapping(Object, Object)	✓	
void closeDeletion(int)		
void clear()	✓	✓
boolean equals(Object)		
int hashCode()		
Object clone()	✓	
Set<K> keySet()	✓	
Collection<V> values()	✓	
Set<Map.Entry<K,V> > entrySet()	✓	
void writeObject(ObjectOutputStream)		
void readObject(ObjectInputStream)		
void putForCreate(K, V)		

Table 3.1: Methods specified with JML and proven with KeY

3.2.3. PROOF STATISTICS

Although we did not formally verify the `IdentityHashMap` entirely, we did analyse most of the typical methods for a map. The project led to an extensive analysis of a map structure with KeY that had not been done before, resulting in over 652 thousand proof steps for 17 methods. Table 3.2 shows some statistics about these proofs. For most of the methods, multiple behaviour specification cases of several contracts had to be proven, resulting in multiple proof files per method. For example, the put method statistics are an accumulation of the statistics of 6 separate behaviour specification cases: (1) the exceptional behaviour specification case of the method's top level contract, (2) the normal behaviour specification case of the method's top level contract, (3) the exceptional behaviour specification case of one of the block contracts, and (4 – 6) three normal behaviour specification cases of three block contracts. Table 3.2 contains the accumulated statistics of these proof files per method.

[4] Only one of two contracts proven

Method	Nodes	Br.	IS	SE	QI	OS	OC	BC	LI	TR	JML	LOC
capacity	67,215	69	0	387	85	0	0	0	4	67,211	37	11
clear	11,801	24	0	179	11	0	0	0	2	11,799	19	7
containsKey	21,734	55	0	197	51	0	6	0	2	21,732	19	14
containsMapping	19,372	43	0	150	64	0	4	0	1	19,371	18	14
containsValue	18,993	26	0	280	15	0	0	0	2	18,991	13	7
get	34,632	82	0	260	114	0	6	0	2	34,630	29	14
init	2,141	12	0	75	2	0	1	0	0	2,140	15	4
isEmpty	150	2	0	30	1	0	0	0	0	148	5	3
maskNull	121	2	0	18	0	0	0	0	0	120	4	3
nextKeyIndex	464	2	0	29	0	0	0	0	0	463	5	3
put	371,106	941	166	963	1,790	24,885	15	1	2	404,582	140	25 [5]
resize	99,190	78	0	360	110	0	3	0	2	99,188	45	30
size	122	2	0	22	1	0	0	0	0	120	5	3
unmaskNull	131	1	0	24	0	0	0	0	0	130	4	3
EntrySet.clear	1,831	10	0	18	15	0	3	0	0	1,828	11	3
KeySet.clear	1,831	10	0	18	15	0	3	0	0	1,828	11	3
Values.clear	1,282	8	0	13	7	0	2	0	0	1,280	11	3
Total	652,116	1,367	166	3,023	2,281	24,885	43	1	17	685,561	391	150

Br.: Number of branches in the proof tree, **IS**: Interactive Steps (number of interactively (manually) applied rules), **SE**: Symbolic Execution steps, **QI**: Quantifier Instantiations, **OS**: One-step Simplification applications, **OC**: Operation Contract applications, **BC**: Block Contract applications, **LI**: Loop Invariant applications, **TR**: Total number of Rule applications, **JML**: lines of JML spec. (KeY only, and not including comment lines), **LOC**: Lines Of Code (Java code excluding whitelines and comment lines).

Table 3.2: Lines of code, lines of specification, and KeY statistics per proof

Notice that most methods did (eventually) not need any interactive (manual) steps (see column IS). Methods like `size`, `isEmpty`, or `nextKeyIndex` are very small (see column LOC), and have relatively small specification contracts (column JML), so it is not surprising that these method were automatically proven. Other methods, however, like `get`, `resize`, `containsKey`, `containsMapping`, or `containsValue` are significantly larger. Nevertheless, these were proven automatically as well. This was, however, not immediately the case.

During the analysis, in the incremental process of refining and improving the specifications, these methods needed interactive steps as well. This would regularly result in substantial improvements of a method contract, block contract, loop invariant, or, in a few cases, the class invariant. Improving these specifications in some cases required improving the unit tests as well, and, obviously, running them again. (In cases where the improvements were strictly limited to block contracts or loop invariants, we did not need to re-test, because we did not write any unit tests for block contracts and loop invariants. The reason for this is explained in section 3.3.3 (Block contracts and loop invariants) on page 47.) The final version of the specifications enabled us to prove the methods automatically, except for the `put` method.

The `put` method required some interactive steps, e.g. hiding irrelevant parts of the large JavaDL formulas manually. JavaDL sequent formulas, that are generated by KeY, can be-

[5] The original code has 21 lines of code. To be able to write two JML block contracts, we divided (part of) the body of the method into blocks, adding 4 enclosing curly braces (for two new blocks) on four new lines, resulting in a total of 25 lines.

come very large, and sometimes the contain conditions that are irrelevant for a specific proof. These conditions can be excluded ('hidden') manually for that specific proof.

Notice also that the number of specification lines is significantly larger than the lines of code (see columns JML and LOC, respectively). This doesn't even take into account the number of specification lines of the class invariant (being 48 lines, excluding whitelines and explanatory comment lines).

3.2.4. A DETAILED EXAMPLE: VERIFICATION OF CONTAINSKEY

Here we describe the verification of the `containsKey` method with KeY. First, we describe the class invariant in detail. Subsequently, we discuss the top-level contract of the method and the loop invariant for the while loop inside the method. Finally, the verification with KeY is addressed. (Note that the Java + JML listings in this section contain so-called *conditional* JML, recognisable by the `/*+KEY@ ... @*/` annotation. It is possible to write JML specifications specifically for a particular tool, e.g. KeY or OpenJML. To grasp this section, this can be ignored. For more on conditional JML, see section 3.4 (A preliminary check of the JML specifications with OpenJML) on page 47.)

THE CLASS INVARIANT

The JML specification of the class invariant is shown in listing 3.3. Here we will clarify some parts of the class invariant that are particularly interesting in relation to the `containsKey` method. Because the method is `\strictly_pure` (see listing 3.4 on page 38), it should not break the invariant. But, in order to work correctly, it does rely on a number of conditions in the class invariant.

It is not allowed for an entry in the map to have an empty key (`null`). If an empty element (`null`) is present on an even position in `table`, we assume we are dealing with a vacant entry. This is enforced by the condition on lines 10 – 14: if a key (an element on an even position in `table` is `null`, then the corresponding value (the subsequent element in `table`) must also be `null`, so the key-value pair represents a vacant entry in the map. Obviously, if a key is not `null`, it must be unique (see lines 16 – 21). This does, of course, not apply to empty 'keys'.

Another important condition is that `table` must always contain at least one empty entry (see lines 38 – 43). Without this invariant condition, methods that search the array for a certain key (e.g. `containsKey`, `containsMapping`, or `get`), could end up in an infinite loop. The two conditions on lines 45 – 63 in listing 3.3 are related. They enforce that no gap (empty entry) exists between any two entries (k_n, v_n) and (k_m, v_m) where *hash*(k_n, `table.length`) == *hash*(k_m, `table.length`). See also: section 3.1 (Implementation of IdentityHashMap) on page 31.

```
1  /*+KEY@ // JML specifically for KeY
2    @ public invariant
3    @   table != null &&
4    @   MINIMUM_CAPACITY == 4 &&
5    @   DEFAULT_CAPACITY == 32 &&
6    @   MAXIMUM_CAPACITY == 536870912 &&
7    @   MINIMUM_CAPACITY * (\bigint)2 <= table.length  &&
8    @   MAXIMUM_CAPACITY * (\bigint)2 >= table.length;
9    @
10   @ // For all key-value pairs: if key == null, then value == null
11   @ public invariant
12   @   (\forall \bigint i;
```

```
13    @     0 <= i && i < table.length / (\bigint)2;
14    @     (table[i * (\bigint)2] == null ==> table[i * (\bigint)2 + 1] == null));
15    @
16    @ // Non-empty keys are unique
17    @ public invariant
18    @   (\forall \bigint i; 0 <= i && i < table.length / (\bigint)2;
19    @     (\forall \bigint j;
20    @       i <= j && j < table.length / (\bigint)2;
21    @       (table[2 * i] != null && table[2 * i] == table[2 * j]) ==> i == j));
22    @
23    @ public invariant
24    @   threshold < MAXIMUM_CAPACITY;
25    @
26    @ // Size equals the number of non-empty keys in the table
27    @ public invariant
28    @   size == (\num_of \bigint i;
29    @     0 <= i < table.length / (\bigint)2;
30    @     table[2 * i] != null);
31    @
32    @ // Table length is a power of two
33    @ public invariant
34    @   (\exists \bigint i;
35    @     0 <= i < table.length;
36    @     \dl_pow(2,i) == table.length);
37    @
38    @ // Table must have at least one empty key-element to prevent
39    @ // infinite loops when a key is not present.
40    @ public invariant
41    @   (\exists \bigint i;
42    @     0 <= i < table.length / (\bigint)2;
43    @     table[2 * i] == null);
44    @
45    @ // There are no gaps between a key's hashed index and its actual
46    @ // index (if the key is at a higher index than the hash code)
47    @ public invariant
48    @   (\forall \bigint i;
49    @     0 <= i < table.length / (\bigint)2;
50    @     table[2 * i] != null && 2 * i > hash(table[2 * i], table.length) ==>
51    @     (\forall \bigint j;
52    @       hash(table[2 * i], table.length) / (\bigint)2 <= j < i;
53    @       table[2 * j] != null));
54    @
55    @ // There are no gaps between a key's hashed index and its actual
56    @ // index (if the key is at a lower index than the hash code)
57    @ public invariant
58    @   (\forall \bigint i;
59    @     0 <= i < table.length / (\bigint)2;
60    @     table[2 * i] != null && 2 * i < hash(table[2 * i], table.length) ==>
61    @     (\forall \bigint j;
62    @       hash(table[2 * i], table.length) <= 2 * j < table.length || 0 <= 2 * j <
        2 * i;
63    @       table[2 * j] != null));
64    @
65    @ // All keys and values are of type Object
66    @ public invariant
67    @   \typeof(table) == \type(Object[]);
68    @
69    @ // Fields modCount and threshold are of type integer (limits:
70    @ // Integer.MIN_VALUE and Integer.MAX_VALUE)
71    @ public invariant
72    @   \dl_inInt(modCount) && \dl_inInt(threshold);
73    @
74    @*/
```

Listing 3.3: The class invariant (conditional JML for KeY)

THE METHOD CONTRACT

Listing 3.4 shows the `containsKey` method, including the conditional JML for KeY. The method contract is shown on lines 1 – 8. This contract has no precondition other than the conditions in the class variant. The postcondition for this method states that the return value of the method is *true* if the result of `maskNull(key)` exists in `table` on an even index, or *false* otherwise. As explained in section 3.1, keys are stored in `table` on even positions, and values are stored on odd positions. Note that the `maskNull` method is applied to the actual parameter `key`. This method replaces `key` with `NULL_KEY` if it is `null`.

```
1  /*+KEY@
2    @ also
3    @ public normal_behavior
4    @   ensures
5    @     \result <==> (\exists \bigint j;
6    @         0 <= j < (table.length / (\bigint)2);
7    @         table[j * 2] == maskNull(key));
8    @*/
9  public /*@ strictly_pure @*/ boolean containsKey(Object key) {
10   Object k =  maskNull(key);
11   Object[] tab =  table;
12   int len =  tab.length;
13   int i =  hash(k, len);
14
15   //+KEY@ ghost \bigint hash = i;
16
17   /*+KEY@
18     @ // Index i is always an even value within the array bounds
19     @ maintaining
20     @   i >= 0 && i < len && i % (\bigint)2 == 0;
21     @
22     @ // Suppose i > hash. This can only be the case when no key k and no null
23     @ // is present at an even index of tab in the interval [hash..i-2].
24     @ maintaining
25     @   (i > hash) ==>
26     @   (\forall \bigint n;
27     @     hash <= (2*n) < i;
28     @     tab[2*n] != k && tab[2*n] != null);
29     @
30     @ // Suppose i < hash. This can only be the case when no key k and no null
31     @ // is present at an even index of tab in the intervals [0..i-2] and
32     @ // [hash..len-2].
33     @ maintaining
34     @   (i < hash) ==>
35     @   (\forall \bigint n;
36     @     hash <= (2*n) < len;
37     @     tab[2*n] != k && tab[2*n] != null) &&
38     @   (\forall \bigint m;
39     @     0 <= (2*m) < i;
40     @     tab[2*m] != k && tab[2*m] != null);
41     @
42     @ decreasing (\bigint)len - ((\bigint)len + i - hash) % (\bigint)len;
43     @
44     @ assignable \strictly_nothing;
45     @*/
46   while (true) {
47     Object item =  tab[i];
48     if (item == k)
49       return true;
50     if (item == null)
51       return false;
52     i = nextKeyIndex(i, len);
53   }
54 }
```

Listing 3.4: The `containsKey` method (with conditional JML for KeY)

THE LOOP INVARIANT

The `containsKey` method contains one while loop (see lines 46 –53 in listing 3.4). The corresponding loop invariant is on lines 15 – 45. On line 15, a ghost field named `hash` is defined and initialised with the calculated hash that was stored in variable `i` (the first position in `table` to look for `k`). Ghost fields are only present for the purpose of specification and can, therefore, only be used inside JML annotations. We need the hash value for two invariant conditions as well as proof that the loop terminates (see below).

The first invariant condition is straightforward: the index variable `i` is always an even value within the array bounds. The second and third condition are derived from the class invariant conditions that no gap exists between entries that have an identical hash (i.e. their key hash). In the first iteration, `i` equals `hash`, and `tab[i]` might contain `k`. If is does, the method returns `true`, and the loop ends (line 49). If `tab[i]` is empty, the loop also ends because it is assumed that `k` does not exist (line 51). In any other case, we need more iterations to determine if `k` exists in `table`. Depending on the result of `nextKeyIndex`, that returns (`i` + 2) mod `table.length`, `i` will be either greater than, or smaller than `hash`. In all subsequent iterations where `i` > `hash`, we know that no element in `table` with an even index n that we tried in previous iteration(s) (`hash` $\leq n <$ `i`) either contains `k` or `null` (see lines 22 – 28). Moreover, in all subsequent iterations where `i` < `hash`, we know that also no element with an even index in the ranges [0..`i`-2] and [`hash`..`table.length`-2] contains either `k` or `null` (see lines 30 – 40).

A *variant-function* (see line 42) was used to help prove the termination of the while statement. This (numeric) expression must decrease each iteration (hence the keyword `decreasing`) and must be never less than 0. In the formula on line 42, `i` is the only changing variable. Initially, it equals the value of `hash`. With every iteration it is incremented by 2 unless it reaches or exceeds the value of `len`. In that case `i` becomes 0. (Indeed, `nextKeyIndex` returns (`i` + 2) mod `len`.) From this, we can conclude that (`len` + `i` - `hash`) mod `len` *increases* by 2 with every iteration (unless we go into an infinite loop). This means that `len` - (`len` + `i` - `hash`) mod `len` should *decrease* by 2 with every iteration. This is demonstrated in table 3.3, where the values of the variant-function are shown for 5 iterations and an array of 10 elements. Note that, if in the first iteration `i` equals `hash`, with every step the result of the function decreases with 2. Note also that variant-function would be violated with a 6^{th} iteration. Indeed, `i` would become 4, resulting in a value of 10 for the variant-function (like in the first line in the table).

`i`	`hash`	`len`	`len` - (`len` + `i` - `hash`) mod `len`
4	4	10	10
6	4	10	8
8	4	10	6
0	4	10	4
2	4	10	2

Table 3.3: Variant function value table

VERIFICATION OF CONTAINSKEY WITH KEY

The memory used by KeY can grow extensively, especially when the number of nodes and/or branches in the proof go up. We therefore increased the size of the heap space by using the JVM option `-Xmx` (denoting the maximum size of the heap in Java). To be able to do so, we

first created a jar-file. From the root folder of the KeY project files structure we created this jar in the following way:

```
$ cd ./key/key
$ ./gradlew shadowJar
```

After this, a jar-file was created in the `key.ui` folder, that we started as follows:

```
$ java -Xmx16G -jar ./key.ui/build/libs/key-2.7-exe.jar
```

To formally verify the `containsKey` method, two contracts had to be verified: one concerning the normal behaviour of `java.util.Map::containsKey` and one concerning the normal behaviour of `java.util.IdentityHashMap::containsKey`.

Based on the final version of the JML, as depicted in listing 3.4, we were able to verify both contracts without any manual (intermediate) steps. Firstly, we opened the contract related to the normal behaviour of `java.util.Map::containsKey`, and took the following steps:

1. We applied the 'Finish Symbolic Execution' macro (one of the strategy macros of KeY). Four open goals remained, one of them related to the loop invariant concerning the while-loop iterating over all the keys in the map.

2. Next, we applied the loop invariant rule (context menu items 'Loop Invariant' > 'Apply Rule').

3. We set the maximum number of rule applications of the 'Proof Search Strategy' to 600, and tried to close all provable goals in the proof tree (by applying the strategy macro 'Close All Provable Goals Below' on the root of the proof tree). By keeping the maximum number of rule applications low, we would only close the easy to close branches of the tree (i.e. the 'low hanging fruit'), without wasting too much time. After this, two goals remained open, labelled 'Body Preserves Invariant' and 'Loop Invariant Use Case' respectively.

4. We then tried to close both open goals, one at a time, starting with the one labelled 'Body Preserves Invariant'. We first set the maximum number of rule applications of the 'Proof Search Strategy' to 10,000. Subsequently, we tried to close the goal using the strategy macro 'Close All Provable Goals Below'. KeY applied 6,859 rules and closed the open goal. We applied the same tactic to the open goal labelled 'Loop Invariant Use Case', which was closed by applying 1,281 rules.

Next, we used the exact same technique to prove the contract related to the normal behaviour of `java.util.IdentityHashMap::containsKey`. The only difference was the number of rules applied by KeY to close the two open goals labelled 'Body Preserves Invariant' and 'Loop Invariant Use Case' (9,695 and 1,283, respectively).

3.3. UNIT TESTS FOR JML CONTRACTS

In an attempt to limit the effort of formal specification, we wrote unit tests to perform preliminary checks on the JML contracts we designed for the CUA. The method we used for writing unit tests is described in detail in section 2.3.2 (Preliminary validation of JML specifications with JUnit) on page 23. JUnit, combined with Java Reflection, proved to be a fairly good tool for creating tests to perform preliminary sanity checks of the JML contracts. Sanity checks, no more, no less. But beneficial nonetheless. They enabled us to verify if our intentions in the specification matched the actual behaviour of the code in action and helped to get some thinking flaws out of the way, but also had some limitations, that are discussed in chapter 4 (Discussion) [6].

3.3.1. TESTING THE CLASS INVARIANT

We benefitted particularly from the unit tests that were written to test the class invariant. The `IdentityHashMap`'s class invariant was quite extensive (48 lines of code, not including comment lines and whitelines), and therefore pretty complex and prone to errors. But, due to solid test coverage of the class invariant, the number of times we had to improve it during the verification process was very limited. The conditional JML for KeY is shown in listing 3.3 (page 36).

Besides the class invariant of the main class, we also wrote the class invariants for a number of the inner classes of the `IdentityHashMap`. (For the complete structure of the class, see figure 2.5 on page 24.) The unit tests we wrote cover these class invariants of inner classes as well. The rationale behind this is that at the start and after execution of any method of the `IdentityHashMap`, all these invariants should hold.

We wrote a test class `IdentityHashMapClassInvariantTest` for testing the class invariants. This class tests if all class invariants hold after constructing a map (for every constructor in the class), as well as after calling a few of the methods (`put`, `remove`, `clear`, and `clone`). In this class we did not test all the methods extensively, because we wrote separate test classes for most of the methods, in which we also test the class invariant.

At the core of all these class invariant tests is a helper class `ClassInvariantTestHelper` that does the actual testing. It contains a method `assertClassInvariants` (see listing 3.5) that is executed by almost all other test classes before and after testing any method, and after testing any constructor. This method executes four other methods, testing the class invariants of the main class, and the inner classes `IdentityHashMapIterator`, `EntryIterator`, and `EntryIterator.Entry`, respectively.

[6] The actual unit test code is part of the project deliverables, and is also available on GitHub. See: https://github.com/m4ndeb2r/IM9906-IdentityHashMapSpecTester/

```
/**
 * Checks the class invariants of the main class as well as the inner classes.
 *
 * @param map an instance of the {@link IdentityHashMap}
 * @throws NoSuchFieldException   if any of the expected private fields does
 *                                not exist
 * @throws IllegalAccessException if it was not possible to get access to a
 *                                required private field
 * @throws NoSuchClassException   if any of the expected inner classes does
 *                                not exist
 */
protected static void assertClassInvariants(AbstractMap<?, ?> map)
    throws NoSuchFieldException, IllegalAccessException, NoSuchClassException {
  // Assert invariant checks on the IdentityHashMap level
  assertIdentityHashMapClassInvariant(map);
  // Assert invariant checks on the IdentityHashMap$IdentityHashMapIterator level
  assertIdentityHashMapIteratorClassInvariant(map);
  // Assert invariant checks on the IdentityHashMap$EntryIterator level
  assertEntryIteratorClassInvariant(map);
  // Assert invariant checks on the IdentityHashMap$EntryIterator$Entry level
  assertEntryClassInvariant(map);
}
```

Listing 3.5: The `ClassInvariantTestHelper.assertClassInvariants` method

Here we will zoom in on the first test method, `assertIdentityHashMapClassInvariant`, and ignore the other three for clarity reasons. The test method is shown in listing 3.6. When comparing listing 3.3 (the class invariant of the main class) with listing 3.6 (the test for the class invariant), it should be easy to see the relation between the two. We made the test code as self-explanatory as possible and added abundant Javadoc containing references to the tested JML specification.

```
/**
 * Checks the class invariant of the main class ({@link IdentityHashMap}).
 *
 * @param map an instance of the {@link IdentityHashMap} to test
 * @throws NoSuchFieldException   if any of the expected private fields
 *                                does not exist
 * @throws IllegalAccessException if it was not possible to get access to a
 *                                required private field
 */
private static void assertIdentityHashMapClassInvariant(AbstractMap<?, ?> map)
    throws NoSuchFieldException, IllegalAccessException {
  final int minimumCapacity = (int) getValueByFieldName(map, "MINIMUM_CAPACITY");
  final int maximumCapacity = (int) getValueByFieldName(map, "MAXIMUM_CAPACITY");
  final Object[] table = (Object[]) getValueByFieldName(map, "table");

  // Class invariant for IdentityHashMap:
  //     table != null &&
  //     MINIMUM_CAPACITY == 4 &&
  //     MAXIMUM_CAPACITY == 536870912 &&
  //     MINIMUM_CAPACITY * 2 <= table.length &&
  //     MAXIMUM_CAPACITY * 2 >= table.length
  // Table.length must be between 4 * 2 and 536870912 * 2 (constants
  // MINIMUM_CAPACITY * 2 and MAXIMUM_CAPACITY * 2 respectively).
  assertThat(table, notNullValue());
  assertThat(table.length, greaterThanOrEqualTo(minimumCapacity * 2));
  assertThat(table.length, lessThanOrEqualTo(maximumCapacity * 2));

  // Class invariant for IdentityHashMap:
  //     (\forall int i;
  //         0 <= i && i < table.length - 1;
  //         i % 2 == 0 ==> (table[i] == null ==> table[i + 1] == null));
  // If the key is null, than the value must also be null
  for (int i = 0; i < table.length - 1; i += 2) {
```

```
34      if (table[i] == null) {
35        assertThat(table[i + 1] == null, is(true));
36      }
37    }
38
39    // Class invariant for IdentityHashMap:
40    //     (\forall int i; 0 <= i && i < table.length / 2;
41    //        (\forall int j;
42    //          i <= j && j < table.length / 2;
43    //         (table[2*i] != null && table[2*i] == table[2*j]) ==> i == j));
44    // Every none-null key is unique
45    for (int i = 0; i < table.length / 2; i++) {
46      if (table[2 * i] == null) continue; // Performance+
47      for (int j = i; j < table.length / 2; j++) {
48        if (table[2 * i] != null && table[2 * i] == table[2 * j]) {
49          assertThat(i, is(j));
50        }
51      }
52    }
53
54    // Class invariant for IdentityHashMap:
55    //     threshold < MAXIMUM_CAPACITY
56    final int threshold = (int) getValueByFieldName(map, "threshold");
57    assertThat(threshold, lessThan(maximumCapacity));
58
59    // Class invariant for IdentityHashMap:
60    //      size == (\num_of int i;
61    //         0 <= i < table.length /2;
62    //         table[2*i] != null)
63    // Size equals number of none-null keys in table
64    int expectedSize = 0;
65    for (int i = 0; i < table.length / 2; i++) {
66      if (table[2 * i] != null) {
67        expectedSize++;
68      }
69    }
70    assertThat(map.size(), is(expectedSize));
71
72    // Class invariant for IdentityHashMap
73    //    (\exists int i;
74    //      0 <= i < table.length;
75    //         \dl_pow(2,i) == table.length);
76    // Table length is a power of two
77    assertThat(isPowerOfTwo(table.length), is(true));
78
79    // Class invariant for IdentityHashMap
80    //    (\exists int i;
81    //      0 <= i < table.length / 2;
82    //      table[2*i] == null);
83    // Table must have at least one empty key-element to prevent
84    // infinite loops when a key is not present.
85    boolean hasEmptyKey = false;
86    for (int i = 0; i < table.length / 2; i++) {
87      if (table[2 * i] == null) {
88        hasEmptyKey = true;
89        break;
90      }
91    }
92    assertThat(hasEmptyKey, is(true));
93
94    // Class invariant for IdentityHashMap
95    //    (\forall int i;
96    //      0 <= i < table.length / 2;
97    //        table[2*i] != null && 2*i > hash(table[2*i], table.length) ==>
98    //        (\forall int j;
99    //          hash(table[2*i], table.length) <= 2*j < 2*i;
100   //          table[2*j] != null));
101   // There are no gaps between a key's hashed index and its actual
102   // index (if the key is at a higher index than the hash code)
103   for (int i = 0; i < table.length / 2; i++) {
```

```
104       final int hash = hash(table[2 * i], table.length);
105       if (table[2 * i] != null && 2 * i > hash) {
106         for (int j = hash / 2; j < i; j++) {
107           assertThat(table[2 * j] != null, is(true));
108         }
109       }
110     }
111
112     // Class invariant for IdentityHashMap
113     //   (\forall int i;
114     //     0 <= i < table.length / 2;
115     //     table[2*i] != null && 2*i < hash(table[2*i], table.length) ==>
116     //     (\forall int j;
117     //       hash(table[2*i], table.length) <= 2*j < table.length || 0 <= 2*j < 2*i;
118     //       table[2*j] != null));
119     // There are no gaps between a key's hashed index and its actual
120     // index (if the key is at a lower index than the hash code)
121     for (int i = 0; i < table.length / 2; i++) {
122       final int hash = hash(table[2 * i], table.length);
123       if (table[2 * i] != null && 2 * i < hash) {
124         for (int j = hash / 2; j < table.length / 2; j++) {
125           final String msg = String.format(
126               "Value (key) in table[%d] was not expected to be null.", 2 * j
127           );
128           assertThat(msg, table[2 * j] != null, is(true));
129         }
130         for (int j = 0; j < i; j++) {
131           final String msg = String.format(
132               "Key in table[%d] was not expected to be null.", 2 * j
133           );
134           assertThat(msg, table[2 * j] != null, is(true));
135         }
136       }
137     }
138   }
```

Listing 3.6: The `ClassInvariantTestHelper.assertIdentityHashMapClassInvariant` method

By running this test code in all our method and constructor unit tests, assuming it is a correct representation of the actual class invariant, we were able to test if the `IdentityHashMap` actually behaved according to our class invariant specifications. This test, therefore, can be seen as the most crucial one in helping us to test our JML contracts.

3.3.2. TESTING THE METHOD CONTRACTS

The method contracts for a fair number of methods were also tested with JUnit. It's important to note that our unit tests only cover the class invariant and the method contracts. There is no test coverage of any block contract or loop invariant inside the method. The reason for this is explained below, in section 3.3.3 (Block contracts and loop invariants). Table 3.4 shows for which methods of the `IdentityHashMap` the method contracts were tested with JUnit [7]. Note that the `closeDeletion` method was not specified formally, but we did test it. In this case, only the class invariant was tested at the start and after the method's execution.

Method	JML	JUnit
Class invariant	✓	✓
Object maskNull(Object)	✓	✓
Object unmaskNull(Object)	✓	✓
IdentityHashMap()	✓	✓
IdentityHashMap(int)	✓	✓
int capacity(int)	✓	✓
void init(int)	✓	✓
IdentityHashMap(Map<K,V>)	✓	✓
int size()	✓	✓
boolean isEmpty()	✓	✓
int hash(Object, int)		
int nextKeyIndex(int, int)	✓	✓
V get(Object)	✓	✓
boolean containsKey(Object)	✓	✓
boolean containsValue(Object)	✓	✓
boolean containsMapping(Object, Object)	✓	✓
V put(K, V)	✓	✓
void resize(int)	✓	✓
void putAll(Map<K,V>)	✓	✓
V remove(Object)	✓	✓
boolean removeMapping(Object, Object)	✓	✓
void closeDeletion(int)		✓
void clear()	✓	✓
boolean equals(Object)		
int hashCode()		
Object clone()	✓	✓
Set<K> keySet()	✓	
Collection<V> values()	✓	
Set<Map.Entry<K,V> > entrySet()	✓	
void writeObject(ObjectOutputStream)		
void readObject(ObjectInputStream)		
void putForCreate(K, V)		

Table 3.4: Methods for which the JML contracts were tested with JUnit

[7] A complete list of specified, tested and verified methods and inner classes of the `IdentityHashMap`, subdivided per tool, is included in appendix C.

As is clear from table 3.5, a considerable amount of lines of test code were written. Most unit tests were written to explicitly test the JML, but some had other purposes (e.g. performance, correctness of improved methods). 40,90% of the test code consisted of utility or helper classes, most of which were used in the JML tests (see, for example, listing 3.5 on page 42, where the `assertClassInvariants` method of the `ClassInvariantTestHelper` class is shown).

Test class	Test subject	LOC	LOC [%]
JML tests			
IdentityHashMapCapacityTest.java	capacity	223	6.67%
IdentityHashMapClassInvariantTest.java	Class invariants	153	4.58%
IdentityHashMapClearTest.java	clear	46	1.38%
IdentityHashMapCloneTest.java	clone	39	1.17%
IdentityHashMapCloseDeletionTest.java	closeDeletion	49	1.47%
IdentityHashMapConstructorsTest.java	Constructors	102	3.05%
IdentityHashMapContainsKeyTest.java	containsKey	64	1.91%
IdentityHashMapContainsMappingTest.java	containsMapping	74	2.21%
IdentityHashMapContainsValueTest.java	containsValue	49	1.47%
IdentityHashMapEntrySetSizeTest.java	Entry.setSize	22	0.66%
IdentityHashMapEqualsTest.java	equals	17	0.51%
IdentityHashMapGetTest.java	get	78	2.33%
IdentityHashMapHashCodeTest.java	hashCode	16	0.48%
IdentityHashMapHashTest.java	hash	32	0.96%
IdentityHashMapInitTest.java	init	40	1.20%
IdentityHashMapIsEmptyTest.java	isEmpty	31	0.93%
IdentityHashMapKeySetContainsTest.java	KeySet.contains	31	0.93%
IdentityHashMapKeySetSizeTest.java	KeySet.size	22	0.66%
IdentityHashMapMaskNullTest.java	maskNull	22	0.66%
IdentityHashMapNextKeyIndexTest.java	nextKeyIndex	56	1.68%
IdentityHashMapPutAllTest.java	putAll	45	1.35%
IdentityHashMapPutTest.java	put	120	3.59%
IdentityHashMapReadObjectOverflowTest.java	readObject	26	0.78%
IdentityHashMapRemoveMappingTest.java	removeMapping	89	2.66%
IdentityHashMapRemoveTest.java	remove	96	2.87%
IdentityHashMapResizeTest.java	resize	99	2.96%
IdentityHashMapSizeTest.java	size	24	0.72%
IdentityHashMapUnmaskNullTest.java	unmaskNull	22	0.66%
IdentityHashMapValuesContainsTest.java	Values.contains	31	0.93%
IdentityHashMapValuesSizeTest.java	Values.size	22	0.66%
Performance and regular unit tests			
ConstructorBugAndPerformanceTest.java	Constructor	68	2.03%
ImprovedReadObjectTest.java	readObject	84	2.51%
SmallIdentityHashMapConstructorTest.java	Constructors	40	1.20%
SmallIdentityHashMapPutTest.java	put	31	0.93%
SmallIdentityHashMapReadObjectOverflowTest.java	readObject	28	0.84%
Other			
Utility classes and helper classes		1352	40.44%
Total		3343	100.00%

Table 3.5: Lines of test code (not including comment lines and whitelines)

3.3.3. Block contracts and loop invariants

Java Reflection is a very useful feature in the Java programming language to allow access to and manipulation of internal (private) properties of a program. It allowed us to verify the values of fields before and after (but not during) the execution of a method or constructor, enabling us to test if pre- and postconditions of methods held, as well as the class invariant. We were, however, not able to test if loop invariants and block contracts held during the execution of methods, due to limitations of JUnit, even when combined with Java Reflection.

3.4. A preliminary check of the JML specifications with OpenJML

Another way in which we attempted to limit the effort of formal analysis, to use a hybrid approach, as suggested by Boerman *et al.* [12]. By using the automatic verification tool OpenJML alongside KeY, we hoped to benefit from advantages of OpenJML for less complex methods. We encountered some other differences between OpenJML and KeY we had to solve. Firstly, we encountered some syntactical issues. In our JML specifications we used keywords that are standard JML, known to KeY, but not recognised by OpenJML. For example: `\min` and `\num_of`. Additionally, to check if a value is a power of 2, we used the keyword `\dl_pow`, which is exclusively known to KeY, and not standard JML.

These issues could quite easily be solved by applying conditional JML. By surrounding an annotation with `/*+KEY@ ... @*/`, it will be considered by KeY, but not by OpenJML, which will ignore the annotation. By surrounding an annotation with `/*+OPENJML@ ... @*/`, it will be the other way around: OpenJML will consider the annotation, and KeY will ignore it. By writing JML annotations specifically for KeY and for OpenJML, we were able to by-pass the aforementioned issues. Listing 3.7 shows an example of how we applied conditional JML for the `clear` method of the class `IdentityHashMap`. Because OpenJML does not support the type *bigint* (see line 10), we defined two separate normal behaviour specification cases, one for KeY (lines 1 – 13) and one for OpenJML (lines 14 – 26). Although OpenJML does support loop invariants, we defined a loop invariant specifically for KeY (see lines 30 – 39). We did this because JJBMC, which also parses the OpenJML specific JML, *does not* support loop invariants. See section 3.5 on page 50 for more on conditional JML in relation to JJBMC.

```
1  /*+KEY@
2    @ also
3    @ public normal_behavior
4    @   assignable
5    @     modCount, size, table, table[*];
6    @   ensures
7    @     \old(modCount) != modCount &&
8    @     \old(table.length) == table.length &&
9    @     size == 0 &&
10   @     (\forall \bigint i;
11   @        0 <= i < table.length;
12   @        table[i] == null);
13   @*/
14 /*+OPENJML@
15   @ also
16   @ public normal_behavior
17   @   assignable
18   @     modCount, size, table, table[*];
19   @   ensures
20   @     \old(modCount) != modCount &&
21   @     \old(table.length) == table.length &&
22   @     size == 0 &&
23   @     (\forall int i;
24   @        0 <= i < table.length;
25   @        table[i] == null);
26   @*/
27 public void clear() {
28   modCount++;
29   Object[] tab =  table;
30   /*+KEY@
31     @ maintaining
32     @   0 <= i && i <= tab.length;
33     @ maintaining
34     @   (\forall \bigint j; 0 <= j < i; tab[j] == null);
35     @ decreasing
36     @   tab.length - i;
37     @ assignable
38     @   table[*];
39     @*/
40   for (int i =  0; i < tab.length; i++)
41     tab[i] = null;
42   size = 0;
43 }
```

Listing 3.7: An example of conditional JML

Besides syntactical differences and unsupported keywords, we also encountered some issues regarding visibility checks. KeY does not complain if a publicly visible specification uses a private variable, while OpenJML does [12]. We solved this by simply complying to the stricter rules of OpenJML regarding visibility and adding `/*@ spec_public @*/` to these variables and methods, making them publicly visible for the specifications addressing them. See listing 3.8, where the private constant `DEFAULT_CAPACITY` is made publicly visible (line 1) to the specification of the default constructor of the `IdentityHashMap` (lines 5–18), where it is used to verify the length of the array `table` after construction.

```
1  private /*@ spec_public @*/ static final int DEFAULT_CAPACITY =  32;
2
3  ...
4
5  /*@ public normal_behavior
6    @   requires
7    @     DEFAULT_CAPACITY == 32;
8    @   ensures
9    @     table != null &&
10   @     table.length == (\bigint)2 * DEFAULT_CAPACITY &&
11   @     keySet == null &&
12   @     values == null &&
13   @     entrySet == null &&
14   @     modCount == 0 &&
15   @     threshold == (DEFAULT_CAPACITY * (\bigint)2) / (\bigint)3 &&
16   @     size == 0 &&
17   @     (\forall \bigint i; 0 <= i && i < table.length; table[i] == null);
18   @*/
19 public IdentityHashMap() {
20   init(DEFAULT_CAPACITY);
21 }
```

Listing 3.8: Applying `spec_public` to make `DEFAULT_CAPACITY` visible to the specification of the default constructor of the `IdentityHashMap`

OpenJML also proved to be stricter regarding inheritance. When running the tool we got a warning that some of the methods in the `IdentityHashMap` override methods in a superclass or interface, and require the specification to contain the keyword `\also` (the keyword `\also` makes it possible to combine heavy weight specification cases). This was easily solved by adding the keyword where it was due.

Finally, OpenJML was able to spot some overflow vulnerabilities in the code. Closer inspection of the code showed that these warnings were not a problem. For example, there is a private integer variable `modCount` that is used for counting modifications to the map. With every modification to the map (i.e. adding or removing an entry), the variable is incremented by 1. This will eventually cause the variable to overflow, resulting in a negative value. However, this is not problematic. The value of `modCount` is exclusively used to check for concurrent modifications, i.e. if the `modCount` changes unexpectedly (regardless if it's greater or smaller), a `ConcurrentModificationException` is thrown.

Although the use of OpenJML proved to be useful for finding small errors like the ones mentioned above (syntax, visibility and inheritance), we were not able to formally verify the `IdentityHashMap` or any of its methods completely. Initially, we used version 0.8.46-20200505 of OpenJML. Running this version resulted in several errors when verifying the `IdentityHashMap`. When running the command `java -jar openjml/openjml.jar -esc -progress IdentityHashMap.java`, we got a `NullPointerException`, and when running `java -jar openjml/openjml.jar -rac -progress IdentityHashMap.java`, we got a "Catastrophic JML internal error". We approached David Cok, the developer of OpenJML to report these issues, and he replied shortly after our email. He suggested to use version 0.8.49 instead of 0.8.46, because several fixes had been made after 0.8.46. Another suggestion was to use the flag `-no-staticInitWarning`. Even so, based on the output we provided, he did see there was a bug for him to fix. Furthermore, Cok did remark that java collections are "a difficult place to start as their semantics is intricate". We applied the suggestions he made and, although there was some improvement (`-rac` gave less error messages), we failed to verify the class or any of its methods completely in a successful way.

In short, OpenJML has been useful for spotting syntactical shortcomings in our JML,

as well as semantical shortcomings related to visibility and inheritance, but we have not successfully completed the verification of any method of the class. The shortcomings in our specification that were detected by OpenJML were nevertheless quite useful, because another tool we used, JJBMC, is based on OpenJML. By eliminating the shortcomings mentioned, we benefitted when running JJBMC, for which the results are discussed in the next section. Furthermore, most of the conditional JML written for OpenJML also applied when running JJBMC.

3.5. A PRELIMINARY CHECK OF THE JML SPECIFICATIONS WITH JJBMC

3.5.1. SOME LIMITATIONS OF JJBMC

JJBMC is currently being developed at the FZI and the KIT. Using it for our project turned out to be mutually beneficial. The tool brought a significant error to the surface in our specifications (see section 3.5.3), and was also able to prove the correctness of a number of methods automatically. At the same time, formally verifying a class with the size and complexity like that of the `IdentityHashMap` also turned out to be a challenge.

Before being able to verify the CUA, our contract specifications needed some adjustments. Because the tool is based on OpenJML, it does not recognize a number of KeY-specific or other keywords (e.g. `\dl_pow` and `\num_of`), and is more strict when it comes to syntax and visibility (scope) checking. We resolved these issues partly by adhering to the stricter rules of OpenJML, and partly by resorting to conditional JML (see section 3.4).

Also, since JJBMC is a model checker, we ran into *the* typical problem with model checkers, which is state space explosion. As mentioned earlier, collections can grow quite large, resulting in an enormous number of possible states. To prevent this from happening, we had to limit the maximum capacity of the `IdentityHashMap` to a very small value and adjust our specifications accordingly. In the original code the maximum capacity of the collection was 536,870,912, which had to be changed to an unrealistic value of 4. As a consequence, also the default capacity of 32 had to be changed to 4 (see listing 3.10).

```
1 private /*@ spec_public @*/ static final int DEFAULT_CAPACITY =  32;
2 private /*@ spec_public @*/ static final int MINIMUM_CAPACITY =  4;
3 private /*@ spec_public @*/ static final int MAXIMUM_CAPACITY =  1 << 29;
```

Listing 3.9: Original capacity values of the IdentityHashMap

```
1 private /*@ spec_public @*/ static final int DEFAULT_CAPACITY =  4;
2 private /*@ spec_public @*/ static final int MINIMUM_CAPACITY =  4;
3 private /*@ spec_public @*/  static final int MAXIMUM_CAPACITY =  4;
```

Listing 3.10: Limited capacity values for verification with JJBMC

Furthermore, loop invariants and block contracts are not supported by JJBMC. Some methods in our CUA contained loop invariants, and a few block contracts were needed to prove methods as well. These were made conditional for KeY. Also, JJBMC does not support exceptional behaviour specification cases. We were, therefore, unable to prove

correct exceptional behaviour (i.e. is the correct exception thrown within the context of a certain precondition?).

A final shortcoming of JJBMC was that it did not support uninterpreted functions. Our method contract specification of the `IdentityHashMap`'s `hash` method (see listing 3.11, line 6) contains an uninterpreted method `genHash`, defined in the file `functions.key` (see listing 3.12). This contract was specified exclusively for KeY for that reason.

```
1  /*+KEY@
2   @ private normal_behavior
3   @   requires
4   @     x != null;
5   @   ensures
6   @     \result == \dl_genHash(x, length) &&
7   @     \result % 2 == 0 &&
8   @     \result < length &&
9   @     \result >= 0;
10  @
11  @ also
12  @ private normal_behavior
13  @   requires
14  @     x == null;
15  @   ensures
16  @     \result == 0;
17  @*/
18 public static /*@ strictly_pure @*/ int hash(Object x, int length) {
19   int h =  System.identityHashCode(x);
20   // Multiply by -127, and left-shift to use least bit as part of hash
21   return ((h << 1) - (h << 8)) & (length - 1);
22 }
```

Listing 3.11: Uninterpreted function `genHash` in method contract of `hash` method

```
1  \functions {
2    int genIdentityHash(any);
3    int genHash(any, int);
4  }
```

Listing 3.12: Source of `functions.key`, containing the uninterpreted function `genHash`

This was a crucial issue, because without being able to use the uninterpreted function JJBMC did not have an abstraction for the `hash` method, that was used in the class invariant of our JML specification. Because the class invariant must hold before and after every method invocation, we could only effectively prove the pure methods (methods that do not have any side effects).

Due to the aforementioned limitations (syntactical and semantical differences in JML dialects, the hazard of state space explosion, missing support for loop invariants and block contracts, and missing support for uninterpreted functions), verification with JJBMC is less rigorous, and has been useful for verification of a limited number of methods.

3.5.2. METHODS VERIFIED WITH JJBMC

Table 3.6 shows the methods of the `IdentityHashMap` that were proven with JJBMC [8].

Method	JML	JJBMC
Class invariant	✓	
Object maskNull(Object)	✓	✓
Object unmaskNull(Object)	✓	✓
IdentityHashMap()	✓	
IdentityHashMap(int)	✓	
int capacity(int)	✓	
void init(int)	✓	✓
IdentityHashMap(Map<K,V>)	✓	
int size()	✓	✓
boolean isEmpty()	✓	✓
int hash(Object, int)		
int nextKeyIndex(int, int)	✓	✓
V get(Object)	✓	
boolean containsKey(Object)	✓	
boolean containsValue(Object)	✓	✓
boolean containsMapping(Object, Object)	✓	
V put(K, V)	✓	
void resize(int)	✓	
void putAll(Map<K,V>)	✓	
V remove(Object)	✓	
boolean removeMapping(Object, Object)	✓	
void closeDeletion(int)		
void clear()	✓	✓
boolean equals(Object)		
int hashCode()		
Object clone()	✓	
Set<K> keySet()	✓	
Collection<V> values()	✓	
Set<Map.Entry<K,V> > entrySet()	✓	
void writeObject(ObjectOutputStream)		
void readObject(ObjectInputStream)		
void putForCreate(K, V)		

Table 3.6: Proven methods of the IdentityHashMap with JJBMC

3.5.3. CONTRACT SPECIFICATION ERROR DETECTED WITH JJBMC

Verifying the CUA with JJBMC resulted in the detection of an error in the JML specifications quite early in the project. JJBMC gave an assertion error for the `containsMapping` method. Apparently, there was something wrong with the contract, or with the method itself. The method and its JML contract are shown in listing 3.13 (note that, for clarity reasons, the listing only shows the OpenJML-specific part of the JML used by JJBMC). The contract aimed to specify the following postcondition (`ensures`): `\result` is `true` if and only if there exist two subsequent elements in `table`, equal to the actual parameters `key` and `value` respectively, where `key` is present in `table` on an even index.

[8] A complete list of specified, tested and verified methods and inner classes of the `IdentityHashMap`, subdivided per tool, is included in appendix C.

```
1  /*+OPENJML@
2    @ private normal_behavior
3    @   ensures
4    @     \result <==> (\exists int i;
5    @       0 <= i < table.length / 2;
6    @       table[i * 2] == key && table[i * 2 + 1] == value);
7    @*/
8  private /*@ spec_public @*/ /*@ strictly_pure @*/ boolean containsMapping(Object
       key, Object value) {
9    Object k =  maskNull(key);
10   Object[] tab =  table;
11   int len =  tab.length;
12   int i =  hash(k, len);
13
14   while (true) {
15     Object item =  tab[i];
16     if (item == k)
17       return tab[i + 1] == value;
18     if (item == null)
19       return false;
20     i = nextKeyIndex(i, len);
21   }
22 }
```

Listing 3.13: The (erroneous) JML of `containsMapping`

A closer look at line 9 indicates what was wrong with the contract. The value of `key` is converted to `NULL_KEY` (a constant `Object` that acts as a placeholder for `null` keys) when its value is `null` (see section 3.1 (Implementation of IdentityHashMap) on page 31). And this is exactly what was missing in our contract.

This can be demonstrated by a counterexample. Suppose we have an `IdentityHashMap`, `emptyMap`, containing 0 mappings. According to our JML contract listing 3.13, the following expressions will have the following results:

```
emptyMap.containsMapping(null, null); // result: true
emptyMap.put(null, null);
emptyMap.containsMapping(null, null); // result: true
```

However, *actual* execution of these expressions will have the following results:

```
emptyMap.containsMapping(null, null); // result: false
emptyMap.put(null, null);
emptyMap.containsMapping(null, null); // result: true
```

It's clear that the latter (*actual* execution of the code) is correct, and the JML contract is not. Indeed, the result of `emptyMap.containsMapping(null, null)` on an empty map should return `false`, because the map is empty and, obviously, contains no mapping at all. This error was solved by correcting the contract, as is shown in listing 3.14, by taking into account the conversion of the actual parameter `key`, if `null`, to the placeholder `NULL_KEY` (line 6) [9].

[9] The same error was present in a number of method contracts, and finding it early probably saved us quite some time. It must also be noted that the verification with JJBMC itself was done by Jonas Klamroth, the developer of JJBMC and scientific staff member at Karlsruher Institut für Technologie (KIT). This joined effort was also key to the achievement.

```
/*+OPENJML@
  @ private normal_behavior
  @   ensures
  @     \result <==> (\exists int i;
  @       0 <= i < table.length / 2;
  @       table[i * 2] == maskNull(key) && table[i * 2 + 1] == value);
  @*/
```

Listing 3.14: The improved JML of `containsMapping`

This error was not only present in the JML contract of the `containsMapping` method, but also in several other methods, like `containsKey`, `get` and `put`.

3.6. Overflow error in the capacity method

We detected a number of overflow situations in the code of the `IdentityHashMap`, most of which were accounted for. For example, as mentioned above in section 3.4, the map contains a field called `modCount` that is incremented every time the map is modified. It is used for detection of concurrent modification. When this field overflows, its value will turn negative, but this does not affect its function in any way. However, we detected one overflow error in the `capacity` method that was not correctly accounted for. This error and its consequences are discussed in detail in the next paragraphs.

3.6.1. The error explained

During the verification of the `capacity` method, KeY was unable to close all proof goals, and closer inspection learned that this had to be due to some error in the code, not the JML contract we specified. The original code of the method is shown in listing 3.15 (the JML specification is left out for readability).

```
/**
 * Returns the appropriate capacity for the specified expected maximum
 * size.  Returns the smallest power of two between MINIMUM_CAPACITY
 * and MAXIMUM_CAPACITY, inclusive, that is greater than
 * (3 * expectedMaxSize)/2, if such a number exists.  Otherwise
 * returns MAXIMUM_CAPACITY.  If (3 * expectedMaxSize)/2 is negative, it
 * is assumed that overflow has occurred, and MAXIMUM_CAPACITY is returned.
 */
private int capacity(int expectedMaxSize)
// Compute min capacity for expectedMaxSize given a load factor of 2/3
{
  int minCapacity =  (3 * expectedMaxSize) / 2;

  // Compute the appropriate capacity
  int result;
  if (minCapacity > MAXIMUM_CAPACITY || minCapacity < 0) {
    result = MAXIMUM_CAPACITY;
  } else {
    result = MINIMUM_CAPACITY;
    while (result < minCapacity)
      result <<= 1;
  }
  return result;
}
```

Listing 3.15: The original `capacity` method

Looking at the Javadoc on lines 6 – 7, the impression is given that the method is overflow-proof ("If (3 * expectedMaxSize)/2 is negative, it is assumed that overflow has occurred"). At first glance, this seems to be confirmed on lines 16 – 17, where `MAXIMUM_CAPACITY` is returned when `minCapacity` (which is initialised with the result of (3 * `expectedMaxSize`) / 2) is negative (see line 12). This assumption, however, is false. This is, perhaps, best explained by looking at an example.

Suppose we pass a value of 1,431,655,765 to the method `capacity`, what would we expect it to return? That would be the smallest power of two between 4 (`MINIMUM_CAPACITY`) and 536,870,912 (`MAXIMUM_CAPACITY`), that is greater than (3 * 1,431,655,765) / 2. If no overflow occurs, we would expect 536,870,912 (`MAXIMUM_CAPACITY`) to be returned, because (3 * 1,431,655,765) / 2 is greater than 536,870,912. If an overflow does occur, we would also expect 536,870,912 (`MAXIMUM_CAPACITY`) to be returned, provided the assumption in the Javadoc is correct that (3 * 1,431,655,765) / 2 results in a negative value. The latter is, however, not the case: multiplying the integer value of 1,431,655,765 by 3 results in -1 (overflow) and dividing an integer -1 by 2 results in 0, which is not a negative value. The smallest power of two between 4 (`MINIMUM_CAPACITY`) and 536,870,912 (`MAXIMUM_CAPACITY`) is 4. The method will, therefore, unexpectedly return 4. There is a range of input values that result in an unexpected outcome, due to an (undetected) overflow, as shown in table 3.7.

Range of input values	(erroneous) output value
1,431,655,765 – 1,431,655,768	4
1,431,655,769 – 1,431,655,771	8
1,431,655,772 – 1,431,655,776	16
1,431,655,777 – 1,431,655,787	32
1,431,655,788 – 1,431,655,808	64
1,431,655,809 – 1,431,655,851	128
1,431,655,852 – 1,431,655,936	256
1,431,655,937 – 1,431,656,107	512
1,431,656,108 – 1,431,656,448	1,024
1,431,656,449 – 1,431,657,131	2,048
1,431,657,132 – 1,431,658,496	4,096
1,431,658,497 – 1,431,661,227	8,192
1,431,661,228 – 1,431,666,688	16,384
1,431,666,689 – 1,431,677,611	32,768
1,431,677,612 – 1,431,699,456	65,536
1,431,699,457 – 1,431,743,147	131,072
1,431,743,148 – 1,431,830,528	262,144
1,431,830,529 – 1,432,005,291	524,288
1,432,005,292 – 1,432,354,816	1,048,576
1,432,354,817 – 1,433,053,867	2,097,152
1,433,053,868 – 1,434,451,968	4,194,304
1,434,451,969 – 1,437,248,171	8,388,608
1,437,248,172 – 1,442,840,576	16,777,216
1,442,840,577 – 1,454,025,387	33,554,432
1,454,025,388 – 1,476,395,008	67,108,864
1,476,395,009 – 1,521,134,251	134,217,728
1,521,134,252 – 1,610,612,736	268,435,456

Table 3.7: Actual parameters resulting in erroneous output of the `capacity` method

Actual input parameters ranging from 1,431,655,765 to 1,431,655,768 give a result of 4, values ranging from 1,431,655,769 to 1,431,655,771 give a result of 8, et cetera. All values ranging from 1,431,655,765 to 1,610,612,736 result in some erroneous value (not being the expected 536,870,912 (`MAXIMUM_CAPACITY`). When the input value is within the range 1,610,612,737 to 2,147,483,646 (`Integer.MAX_VALUE`) the overflow is also not detected, but the `capacity` method does, nevertheless, return the expected value of 536,870,912 (`MAXIMUM_CAPACITY`). This happens because the smallest power of two that is greater than `minCapacity` is, coincidentally, also 536,870,912.

3.6.2. The damage caused by the error

What happens when the overflow error in the `capacity` method occurs? The `capacity` method is a private method that is only called from within the `IdentityHashMap` itself, by three public methods:

- the `putAll(Map m)` method, that adds all the entries from the `Map m` to the current map, and reserves enough space by invoking `resize(capacity(n))` (where `n` is the size of `m`), if necessary,
- the overloaded constructor `IdentityHashMap(int expectedMaxSize)`, that initialises a newly created instance by invoking `init(capacity(expectedMaxSize))`,
- the `readObject(java.io.ObjectInputStream s)` method, that is called during deserialisation of an `IdentityHashMap` instance. It reserves space for the entries in the map + 33% for growth, by invoking `init(capacity((size * 4) / 3)` (where `size` is the size of the serialised map).

```
1  public void putAll(Map m) {
2    int n =  m.size();
3    if (n == 0)
4      return;
5    if (n > threshold) // conservatively pre-expand
6      resize(capacity(n));
7
8    for (Object o: m.entrySet()) {
9      Entry e = (Entry) o;
10     put(e.getKey(), e.getValue());
11   }
12 }
```

Listing 3.16: The `putAll` method

In the `putAll(Map m)` method (see listing 3.16) the error will only occur when the size of `m` is within the range 1,431,655,765 – 1,610,612,736 (see table 3.7). On line 2 the size of `m` is stored in an integer `n`, that is passed as an actual parameter to the `capacity` method on line 6. If the overflow error occurs, the private field `table`[10] of the `IdentityHashMap` will possibly be resized to a smaller size than was expected. (Note that the `resize` method never shrinks the array `table`, but only makes it larger if necessary, or keeps it the same size.) This would suggest there might not be enough space reserved for all the entries of `m`

[10]The `IdentityHashMap` stores its entries (key-value pairs) in an array field `table`, with two elements for every entry in the map: one for the key and one for the value (see section 3.1).

to be added to the array. This is, however, not a problem, because the `put(Object key, Object value)` method, invoked for every separate entry to be added (see line 10, listing 3.16), also resizes the array `table` if necessary (see line 22, listing 3.17). Furthermore, the `IdentityHashMap` always reserves a threshold number of empty elements. In short: if the `putAll` method does not resize the `IdentityHashMap` correctly, due to the overflow error in the `capacity` method, the `put` method will correct this. The `IdentityHashMap` will therefore not crash if the overflow error occured in the `putAll(Map m)` method.

There is, however, a different problem when the size of `m` is within the range 1,431,655,765 – 1,610,612,736. The capacity of the `IdentityHashMap` will be exhausted a soon as the number of added entries surpasses 536,870,912 (`MAXIMUM_CAPACITY`). This will occur when the `resize` method, called from the `put` method (see line 22 in listing 3.17), pushes its luck. It will then throw an `IllegalStateException` (see the code snippet from the `resize` method in listing 3.18).

```
1  public Object put(Object key, Object value) {
2    Object k =  maskNull(key);
3    Object[] tab =  table;
4    int len =  tab.length;
5    int i =  hash(k, len);
6
7    Object item;
8
9    while ( (item = tab[i]) != null) {
10     if (item == k) {
11       Object oldValue = (Object) tab[i + 1];
12       tab[i + 1] = value;
13       return oldValue;
14     }
15     i = nextKeyIndex(i, len);
16   }
17
18   modCount++;
19   tab[i] = k;
20   tab[i + 1] = value;
21   if (++size >= threshold)
22     resize(len); // len == 2 * current capacity.
23   return null;
24 }
```

Listing 3.17: The `put` method

```
1  if (oldLength == 2 * MAXIMUM_CAPACITY) { // can't expand any further
2    if (threshold == MAXIMUM_CAPACITY - 1)
3      throw new IllegalStateException("Capacity exhausted.");
4    threshold = MAXIMUM_CAPACITY - 1;  // Gigantic map!
5    return;
6  }
```

Listing 3.18: A snippet from the `resize` method

The constructor `IdentityHashMap(int expectedMaxSize)` also calls the `capacity` method in order to initialise the newly constructed instance of the class. The input parameter of the constructor is directly passed to the `capacity` method (see line 5, listing 3.19), meaning that the initial size of the array field `table` (in case of an overflow error) will have a smaller size than required. This will, like in the `putAll(Map m)` method, not have severe

consequences. Yes, the array field `table` will have less elements than initially required, but every time the `put(Object key, Object value)` method is invoked to add an entry, it will be resized if necessary. Like with the `putAll(Map m)` method, the overflow error is concealed, and the `IdentityHashMap` will not crash.

```
1  public IdentityHashMap(int expectedMaxSize) {
2    if (expectedMaxSize < 0)
3      throw new IllegalArgumentException("expectedMaxSize is negative: "
4                                         + expectedMaxSize);
5    init(capacity(expectedMaxSize));
6  }
```

Listing 3.19: The constructor `IdentityHashMap(int expectedMaxSize)`

The overflow error can, however, be classified as a performance bug. If, indeed, the constructor is called with an actual parameter causing the error (see table 3.7 on page 55), putting entries in the map will result in an unanticipated number of calls to the `resize` method (see listing 3.20). When the `resize` method is executed, a new hash is calculated for the keys of all the entries in the map (line 29), and the entries are put in a different position in a new table (`newTable`, see lines 32 – 33) that will replace the original one. Especially in a large table, this involves quite some reshuffling of entries.

```
1  /**
2   * Resize the table to hold given capacity.
3   *
4   * @param newCapacity the new capacity, must be a power of two.
5   */
6  private void resize(int newCapacity) {
7    int newLength = newCapacity * 2;
8
9    Object[] oldTable = table;
10   int oldLength = oldTable.length;
11   if (oldLength == 2*MAXIMUM_CAPACITY) { // can't expand any further
12     if (threshold == MAXIMUM_CAPACITY-1)
13       throw new IllegalStateException("Capacity exhausted.");
14     threshold = MAXIMUM_CAPACITY-1;  // Gigantic map!
15     return;
16   }
17   if (oldLength >= newLength)
18     return;
19
20   Object[] newTable = new Object[newLength];
21   threshold = newLength / 3;
22
23   for (int j = 0; j < oldLength; j += 2) {
24     Object key = oldTable[j];
25     if (key != null) {
26       Object value = oldTable[j+1];
27       oldTable[j] = null;
28       oldTable[j+1] = null;
29       int i = hash(key, newLength);
30       while (newTable[i] != null)
31         i = nextKeyIndex(i, newLength);
32       newTable[i] = key;
33       newTable[i + 1] = value;
34     }
35   }
36   table = newTable;
37 }
```

Listing 3.20: The `resize` method

To get an impression of the decline in performance, we wrote a unit test that adds a lot of entries to two maps. One map was initialised with an `expectedMaxSize` that did not cause the constructor to suffer from the overflow error. The other one was initialised with an `expectedMaxSize` of 1,431,655,765, triggering the error and causing the constructor to initially create a map with a capacity of 4. Then, we put as many entries in both maps a possible. This process was repeated quite a number of times in the same run to get a fair comparison between the elapsed time for both maps, while averaging out any disturbance from other processes like the garbage collector, for example. We found that, in a typical test run, the elapsed time for a map with an initially (too) small capacity, showed a decline in performance of about 45%, which is significant. Indeed, we can assume the `IdentityHashMap(int expectedMaxSize)` was designed for a reason, being performance.

```
1  /**
2  * Reconstitute the <tt>IdentityHashMap</tt> instance from a stream (i.e.,
3  * deserialize it).
4  */
5  private void readObject(java.io.ObjectInputStream s)
6      throws java.io.IOException, ClassNotFoundException  {
7    // Read in any hidden stuff
8    s.defaultReadObject();
9
10   // Read in size (number of Mappings)
11   int size =  s.readInt();
12
13   // Allow for 33% growth (i.e., capacity is >= 2* size()).
14   init(capacity((size * 4) / 3));
15
16   // Read the keys and values, and put the mappings in the table
17   for (int i =  0; i < size; i++) {
18     java.lang.Object key =  (java.lang.Object) s.readObject();
19     java.lang.Object value =  (java.lang.Object) s.readObject();
20     putForCreate(key, value);
21   }
22 }
```

Listing 3.21: The `readObject` method

The `readObject` method (see listing 3.21) is invoked when an `IdentityHashMap` is deserialised (after serialisation). In any normal situation, a valid instance of the class is read from an external memory. The size of the serialised map (see line 11 in listing 3.21) is extended by 33% to allow for some growth, and the result is passed to the `capacity` method (line 14). The calculation `(size * 4) / 3` will not overflow because (unless the previously serialised `IdentityHashMap` has been tampered with) we can safely assume the size is smaller than `MAXIMUM_CAPACITY`. In that case, the `calculate` method will also not overflow (indeed 536,870,912 (`MAXIMUM_CAPACITY`) is not within the range 1,431,655,765 – 1,610,612,736 (see table 3.7)).

If, however, the serialised map is tampered with, the `readObject` method can easily be made to crash. If, for example, a hacker would use a hex editor to set the number of mappings the serialised map to 1,200,000,000. Then `s.readInt()` would return this number, and the calculation of the actual parameter that is passed to `capacity` *should* result in 1,600,000,000 (4/3 times 1,200,000,000). However, due to an overflow error in the calculation `(size * 4) / 3`, the result *would* return 168,344,234 as a result of an overflow error. There would not be enough elements in the array field `table` to store all the mappings,

and the `putForCreate(key, value)` call on line 20 would crash, because this method, unlike the `put` method, does not resize the map if necessary. In listing 3.22 we can see why. Because the array field `table` is too small, the variable `len` will be too small (line 10). This will lead to a smaller set of possible hashes (calculated on line 11), resulting inevitably in more clashes. In case of a clash, an attempt is made to store the entry in the next free location. Eventually the array `table` will be completely filled. This will cause an infinite loop on lines 14 – 18: there is no element in `tab` equal to `null` and the `nextKeyIndex` method will infinitely circularly traverse `table` looking for the next key index. The while condition will always be `true`, resulting in another call to `nextKeyIndex`, et cetera, et cetera.

In short, the overflow error in the `capacity` method is concealed here also. This time by an overflow error in the `readObject` itself, one that can only occur when the serialised `IdentityHashMap` has been tampered with. This would result in an infinite loop.

```
1  /**
2  * The put method for readObject.  It does not resize the table,
3  * update modCount, etc.
4  */
5  private void putForCreate(java.lang.Object key, java.lang.Object value)
6      throws IOException
7  {
8    java.lang.Object k =  (java.lang.Object)maskNull(key);
9    Object[] tab =  table;
10   int len =  tab.length;
11   int i =  hash(k, len);
12
13   Object item;
14   while ( (item = tab[i]) != null) {
15     if (item == k)
16       throw new java.io.StreamCorruptedException();
17     i = nextKeyIndex(i, len);
18   }
19   tab[i] = k;
20   tab[i + 1] = value;
21 }
```

Listing 3.22: The `putForCreate` method

3.7. AN IMPROVED VERSION OF THE CAPACITY METHOD

Although the `IdentityHashMap` does not seem to crash as a result of the overflow error in the `capacity` method, it can be classified as a performance bug. Therefore, we considered a fix for the error. This fix we implemented was fairly simple. Our improved version (without the JML specification for readability) is shown in listing 3.23. The actual improvement is on line 8. This expression might still overflow, but will, in that case, always return a negative value, resulting in the execution of line 13, where `MAXIMUM_CAPACITY` is returned.

```
1  private int capacity(int expectedMaxSize)
2  // Compute min capacity for expectedMaxSize given a load factor of 2/3
3  {
4    // Original calculation
5    // int minCapacity =  (3 * expectedMaxSize) / 2;
6
7    // Improved calculation
8    int minCapacity = expectedMaxSize % 2 + (expectedMaxSize / 2) * 3;
9
10   // Compute the appropriate capacity
11   int result;
12   if (minCapacity > MAXIMUM_CAPACITY || minCapacity < 0) {
13     result = MAXIMUM_CAPACITY;
14   } else {
15     result = MINIMUM_CAPACITY;
16     while (result < minCapacity)
17       result <<= 1;
18   }
19   return result;
20 }
```

Listing 3.23: The improved version of the `capacity` method

Our fixed version of the `capacity` method has been tested, and verified with KeY. It passed the test and was formally proven to be correct. We did not propose this improvement to the Java community because, as of the JDK9 version of the `IdentityHashMap`, the `capacity` method has a new implementation [11], that does not suffer from the overflow error we detected.

Since we were, of course, curious if the JDK9 version of the `capacity` method would stand the test, we wrote a unit test to test it, and verified it formally with KeY. We replaced the body of the method in the JDK7 version with the body of the JDK9 version. We left the signature and the JML contract unchanged, except for the loop invariant that we removed, since the new body does not contain a loop anymore. We were able to prove the correctness of the method automatically, without any interactive steps needed. (Listing 3.24 shows the method with the original signature and JML contract, and the replaced method body (lines 35 – 38) – just like we loaded it in KeY.)

[11] http://hg.openjdk.java.net/jdk9/jdk9/jdk/file/847d7a6aef45/src/java.base/share/classes/java/util/IdentityHashMap.java

```
1  /*+KEY@
2    @ private normal_behavior
3    @   requires
4    @     (expectedMaxSize * (\bigint)3) / (\bigint)2 < 0 ||
5    @     (expectedMaxSize * (\bigint)3) / (\bigint)2 > MAXIMUM_CAPACITY;
6    @   ensures
7    @     \result == MAXIMUM_CAPACITY;
8    @
9    @ also
10   @ private normal_behavior
11   @   requires
12   @     (expectedMaxSize * (\bigint)3) / (\bigint)2 >= MINIMUM_CAPACITY &&
13   @     (expectedMaxSize * (\bigint)3) / (\bigint)2 <= MAXIMUM_CAPACITY;
14   @   ensures
15   @     \result >= (expectedMaxSize * (\bigint)3) / (\bigint)2 &&
16   @     \result < (expectedMaxSize * (\bigint)3);
17   @
18   @ also
19   @ private normal_behavior
20   @   requires
21   @     (expectedMaxSize * (\bigint)3) / (\bigint)2 >= 0 &&
22   @     (expectedMaxSize * (\bigint)3) / (\bigint)2 < MINIMUM_CAPACITY;
23   @   ensures
24   @     \result >= MINIMUM_CAPACITY &&
25   @     \result < MINIMUM_CAPACITY * (\bigint)2;
26   @
27   @ also
28   @ private normal_behavior
29   @   ensures
30   @     (\exists \bigint i;
31   @       0 <= i < \result;
32   @       \dl_pow(2,i) == \result); // result is a power of two
33   @*/
34 private int capacity(int expectedMaxSize) {
35   return
36     (expectedMaxSize > MAXIMUM_CAPACITY / 3) ? MAXIMUM_CAPACITY :
37     (expectedMaxSize <= 2 * MINIMUM_CAPACITY / 3) ? MINIMUM_CAPACITY :
38     Integer.highestOneBit(expectedMaxSize + (expectedMaxSize << 1));
39 }
```

Listing 3.24: The version of the `capacity` method, based on the JDK9 implementation

3.7.1. AN IMPROVED VERSION OF THE READOBJECT METHOD

Although not in scope of the formal analysis, we provided an improved version of the `readObject` as well, as a part of the project deliverables. Listing 3.25 shows the original version of the method.

In this version, on line 14, an overflow error will occur if `size` multiplied by 4 is larger than `Integer.MAX_VALUE` (regardless of whether the original version or the improved version of `capacity` is invoked). This overflow error is fixed by the check on lines 14 – 16 of the improved version, shown in listing 3.26. Unless the serialised object is tampered with, it is impossible for `size` to be larger than `MAXIMUM_CAPACITY`. If this is detected to be the case, a `StreamCorruptedException` is thrown. Otherwise, we can rest assured that no overflow will occur when the capacity of the map is calculated (again, regardless whether the original or the improved version of `capacity` is invoked).

```
/**
* Reconstitute the <tt>IdentityHashMap</tt> instance from a stream (i.e.,
* deserialize it).
*/
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException  {
  // Read in any hidden stuff
  s.defaultReadObject();

  // Read in size (number of Mappings)
  int size =  s.readInt();

  // Allow for 33% growth (i.e., capacity is >= 2* size()).
  init(capacity((size * 4) / 3));

  // Read the keys and values, and put the mappings in the table
  for (int i =  0; i < size; i++) {
    java.lang.Object key =  (java.lang.Object) s.readObject();
    java.lang.Object value =  (java.lang.Object) s.readObject();
    putForCreate(key, value);
  }
}
```

Listing 3.25: The original version of the `readObject` method

```
/**
* Reconstitute the <tt>IdentityHashMap</tt> instance from a stream (i.e.,
* deserialize it). This is an improved version of the original readObject method.
* It is extended with extra input validation.
*/
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException  {
  // Read in any hidden stuff
  s.defaultReadObject();

  // Read in size (number of Mappings)
  int size =  s.readInt();

  if (size > MAXIMUM_CAPACITY) {
    throw new java.io.StreamCorruptedException();
  }

  // Allow for 33% growth (i.e., capacity is >= 2* size()).
  init(capacity((size * 4) / 3));

  // Read the keys and values, and put the mappings in the table
  for (int i =  0; i < size; i++) {
    java.lang.Object key =  (java.lang.Object) s.readObject();
    java.lang.Object value =  (java.lang.Object) s.readObject();
    putForCreate(key, value);
  }
}
```

Listing 3.26: The improved version of the `readObject` method

4

DISCUSSION

In chapter 2 (Method) the main research question and a number of subquestions were defined. In this chapter we will connect the results from chapter 3 (Results) to these research questions.

The main research question:

RQ1 : How can we formally analyse Java libraries with JML and KeY?

RQ1 was broken down into the following subquestions:

RQ2 : Which Java classes are suitable candidates for formal analysis?
RQ3 : How can we limit the effort of formal specification?
RQ4 : What error(s), if any, can we identify in the CUA?
RQ5 : Can we provide a fixed version of the class that does stand the test of formal analysis?
RQ6 : What is the effort ratio when performing a formal analysis?

4.1. RQ2 - WHICH JAVA CLASSES ARE SUITABLE CANDIDATES FOR A FORMAL ANALYSIS?

Subquestion **RQ2** was already answered in section 2.3.1 (Finding a suitable candidate for formal analysis) of chapter 2 (see page 21). The `IdentityHashMap` was the class we decided to formally specify with JML and verify with KeY. During the project there was no need to reverse this decision, although the class has not been verified completely, due to its size and complexity (see 3.2 (JML contracts and KeY proof files) on page 33). Also, some methods (the `put` method in particular) turned out to require an extensive amount of memory. In an attempt to solve these performance issues (we initially used an iMac with a 3,4 GHz Quad-Core Intel Core i5 processor, and 16GB 1600 MHz DDR3 internal memory), we switched to a High Performance Computing (HPC) cloud supercomputer at SURF [1]. On the HPC cloud

[1] "SURF is a cooperative association of Dutch educational and research institutions in which the members combine their strengths. Within SURF, we work together to acquire or develop the best possible digital ser-

computer the amount of memory was twice as big:

```
$ free -h
              total        used        free      shared  buff/cache   available
Mem:           31G        1.2G         28G        688K        2.0G         29G
Swap:           0B          0B          0B
```

This, however, did not improve the performance (indeed, the performance on the HPC cloud computer turned out to be even worse). What *did* improve the performance was a number of changes in the formal specifications of the CUA. We improved the class invariant and added a number of block contracts to the `put` method that could be proven separately ('divide and conquer'). The result of this approach was that we had to prove more distinct contracts for the `put` method (6 instead of 2), but the size of the distinct proof trees was significantly reduced. This had a beneficial effect on the memory used by KeY. Eventually, the `put` method (like all other methods) was proven on the iMac computer.

Complexity and performance issues aside, the `IdentityHashMap` was certainly a suitable candidate for (partial) formal analysis with JML and KeY. In fact, we successfully used KeY to actually find the kind of bug we had anticipated to find in a collection, although its consequences were limited (see 1.1.6 (Formal analysis methods and the case for deductive verification) on page 4, requirement **R2**).

4.2. RQ3 – How can we limit the effort of formal specification?

As mentioned in section 2.2.3, formal analysis is tedious work. When, during formal verification, an error in a JML contract surfaces, the specification must be corrected, and verified all over again. This can be very time consuming, especially when verification requires interaction with the analyst. Indeed, KeY is an interactive tool, and not all methods can be verified automatically. Typically, the more complex methods are, the more complex the contracts they require, the bigger the chance KeY is not able to automatically prove these methods and interaction is required. Exactly these complex contracts are susceptible for errors and omissions, making the analyst's work more tedious and more time-consuming.

An even bigger issue arises when errors are detected in the class invariant. Changing the class invariant has great consequences. Indeed, the class invariant must hold after each constructor's execution, and at the beginning and end of each method (except constructors and methods that are declared with the `helper` modifier). So, if the class invariant is changed, all these constructors and methods need to be verified all over again.

Therefore, we tried three different approaches to limit the number of errors in our JML as early in the process as possible. We wrote unit tests to test our JML (see section 3.3 (Unit tests for JML contracts) on page 41), we checked our JML specifications using the automatic verification tool OpenJML (see section 3.4 (A preliminary check of the JML specifications with OpenJML) on page 47), and we did the same by using the model checker JJBMC (see section 3.5 (A preliminary check of the JML specifications with JJBMC) on page 50). Each approach had its pros and cons, as will be discussed below.

vices, and to encourage knowledge sharing through continuous innovation." (http://surf.nl)

4.2.1. Pros and cons of validating JML specifications with JUnit

Unit tests have proven to be fairly good sanity checks for the correctness of the JML contracts. By preparing the execution of a method (initialising the state of the class to meet the preconditions of the method as well as the class invariant), executing the method, and validating the state of the class after its execution (postcondition and class invariant), we were able to detect several errors in some of our JML contracts early. Especially regarding the class invariant, as well as the more complex methods, this approach definitely had benefits. There are, however, some pitfalls to be aware of.

- First of all, we encountered some technical limitations of unit tests in Java in general. For example, it is not possible to write unit tests to verify loop invariants or block contracts in JML. Although it is possible to have access to (part of) the inner workings of a class or a method by using Java Reflection, this does not enable us to easily check the state of the program inside a while or for loop, for example (see section 3.3.3 (Block contracts and loop invariants) on page 47). The `put` method deserves special mention here, because it contains a loop invariant as well as several block contracts that we were not able to test. Formally verifying this method required a considerable amount of time (54.94% of all formal verification effort was spent on the `put` method).

- Moreover, as explained in chapters 2 and 3, we also had to take some shortcuts for pragmatic reasons, regarding checking the heap before and after calling a method.

- Furthermore, the unit tests did not directly test the actual JML contracts. We translated the *intended* contracts into unit test code and tested if the method complied to this *translation* of the contract, when executing it. Translating a contract into a unit test manually, however, is obviously prone to errors. It requires rigour to write unit tests that are consistent with the actual JML contract, because this is not automatically verified. In other words, the unit tests only represent the *intended* specification. This may result in false positives (the actual JML contract is correct, but the translation into unit test code contains an error) or false negatives (the actual JML contract is incorrect, but the translation into unit test code is somehow correcting or camouflaging this error).

- Also, it requires discipline (and extra effort) to keep the unit tests consistent with the JML specifications, especially when these are subject to change.

- Even when a unit test is semantically consistent with the JML contract specification of a method, misconceptions may still persist. When the author of the contract is the same person writing the unit test, this is an obvious pitfall, as we have experienced during the project [2].

- Finally, writing unit tests only helps to detect semantical errors. Syntactical errors are not detected by translating the intended contract design into a unit test. Fortunately, there are other tools to provide syntax checking, like OpenJML, JJBMC, or KeY.

[2] One such error was discovered when running the JJBMC tool for a version of the `IdentityHashMap` containing several similar erroneous postconditions (see section 3.5.3) for which the corresponding unit tests contained the same error. The unit tests failed to notice this error, which was, of course, also a flaw.

Although writing unit tests for the purpose of detecting errors in the specifications early in the process can be useful, this approach must not be overrated. As stated above, this approach does have several limitations to keep in mind. Surely, it did speed up the process by detecting errors in the specification early (especially with regard to the class invariant), but it also required extra effort to write and maintain the unit tests. This is definitely something to keep in mind.

It is not trivial to quantify how much we benefitted, on balance, from testing our JML in a preliminary phase of formal specification. It is obvious that being able to eliminate errors in the class invariant as early as possible is crucial, and unit tests certainly proved to be a useful instrument to accomplish this. We did find several small errors that could have had major consequences should we have to detect them later by running into unclosable goals during verification with KeY. To name one example regarding the class invariant, it was challenging to correctly specify that there can be no gaps (vacant entries) between a key's hashed index and its actual index (see section 3.1 (Implementation of IdentityHashMap) on page 31). Testing these specifications proved to be helpful in the sense that unit tests could help confirm the correctness and accuracy of this part of the class invariant.

Finding an error in a class variant later in the process would have meant that all methods that were affected by any required improvements in the class invariant, would have to be verified all over again. Considering these consequences, it is very likely this would have cost us days, if not weeks to detect, fix and formally verify again. Also, writing unit tests considerably improved our understanding of the code as well as the specifications. This, obviously, improved the quality of the specifications, resulting in less errors later in the process. We are confident that, although we spent a significant amount of time writing the tests (see table 4.1 on page 69), this effort was outweighed by the benefits. In section 4.5 a crude cost-benefit analysis is provided.

4.2.2. Pros and cons of validating JML specifications with OpenJML

Some of the disadvantages of validating JML specifications by means of unit tests are eliminated when using OpenJML. While the unit tests we wrote did not test the actual contracts, but a manually translated version of it, OpenJML loads the original Java class (containing the JML contracts) and verifies it automatically. This way we were able to detect syntax errors and invalid visibility modifiers, which wasn't possible with unit tests. The added advantage was limited, however. Indeed, KeY is also perfectly able to detect syntax errors, when the class is loaded, so we do not need a separate tool for that purpose. Furthermore, conforming to OpenJML's stricter validation rules concerning visibility/accessibility modifiers is good, of course. But since KeY imposes less strict rules, these improvements are not essential.

The syntactical differences between OpenJML and KeY were dealt with by applying conditional JML. This required some extra effort (see section 3.4 (A preliminary check of the JML specifications with OpenJML) on page 47). But there is another downside to using conditional JML. Because it inherently introduces redundancy, there is a danger that both versions of a contract (in our case, the KeY version and the OpenJML version) become inconsistent. This could lead to a situation where, for example, OpenJML does not detect an error because the error only exists in a KeY-specific JML annotation. Consistency is something one should constantly stay aware of when using two or more tools that require conditional JML contracts.

As described in section 3.4, we were unable to automatically verify the `IdentityHashMap` with OpenJML. Therefore, (contrary to the aforementioned findings by Boerman *et al.* [12]) the use of OpenJML has not significantly limited the effort of formal analysis of the CUA in our project. Furthermore, because we did not spend much time using the tool OpenJML, the benefits or costs, in terms of time and effort, were negligible (see table 4.1 on page 69). In short, the use of OpenJML resulted in cleaner contracts (visibility/accessibility modifiers, syntax errors), but also required extra effort (conditional JML). On the positive side, because JJBMC is based on OpenJML, the conditional JML contracts we used with OpenJML were also of use when verifying the code with JJBMC.

4.2.3. Pros and cons of validating JML specifications with JJBMC

For verification of the `IdentityHashMap` with JJBMC we could, for a large part, re-use the conditional JML specifications mentioned above. To prevent the typical model checking problem of state space explosion, we also had to make some minor adjustments in the code as well as the JML, for which the extra effort was negligible, however.

JJBMC was not capable of verifying all methods, unfortunately. This was largely because the tool did not have an abstraction for the `hash` method, that is used in the class invariant of our JML specification. This issue was described in detail in section 3.5.1 (Some limitations of JJBMC) on page 50.

We were able to detect a crucial error in our JML specifications using JJBMC quite early in the project. The tool gave an assertion error for the `containsMapping` method (see section 3.5.3 (Contract specification error detected with JJBMC) on page 52). It is difficult to quantify the amount of time that was saved by discovering this specification error with JJBMC. We would absolutely have encountered the error with KeY while verifying the `containsMapping` method, or any of the other methods that suffered from the error, for that matter. We would definitely not have been able to close open goals interactively, which could have been time consuming. It is fair to assume that finding this specification error with JJBMC saved us two or three days of work. See section 4.5 for a crude cost-benefit analysis.

In short, although JJBMC does have some shortcomings when it comes to verifying a class as complex as the `IdentityHashMap` (or any (virtually) unbounded class in the Collections Framework for that matter), it did tackle a blocking error in the JML at an early stage. Furthermore, since JJBMC is based on OpenJML it would also have detected the syntax, visibility/accessibility issues, had we not found them already by using OpenJML separately (see section 4.2.2). Obviously, all shortcomings of writing conditional JML that were mentioned in the context of using OpenJML (see above) apply here also.

4.3. RQ4 – What error(s), if any, can we identify in the CUA?

In chapter 3, section 3.6, we described the overflow error that was identified in the `capacity` method, how it can be triggered, and what the consequences are whenever the overflow occurs. Indirectly, we've also concluded that the `readObject` method can be improved to be more robust, and detect invalid input.

4.4. RQ5 – Can we provide a fixed version of the class that does stand the test of formal analysis?

We provided a fixed version of the `capacity` method, as well as the `readObject` method in the deliverables. The improvements were not provided to the community, because an improved version of the `capacity` method was already present in the JDK9 version of the `IdentityHashMap`. The improved JDK7 version of the `capacity` method, as well as its JDK9 counterpart were formally verified with KeY and proven correct. We also wrote unit tests for both methods, that were successfully executed.

4.5. RQ6 – What is the effort ratio when performing a formal analysis?

During the project, we kept track of the effort that was put into the several tasks. Table 4.1 contains an overview of the effort ratio during the project. This is probably not the typical effort ratio for an average formal analysis project, because of the hybrid approach we took.

As is clear from the figures in the table, most effort went into formal verification with KeY (closely followed by work on the thesis). Note that there are no hard boundaries between the effort that went into formal specification and formal verification. The 10.0% effort ratio for formal specification (JML) relate to the specifications that were written before we actually started verifying. The effort ratio for formal verification (with OpenJML, JJBMC, and KeY) also involved making improvements to existing JML contracts.

Task	effort ratio
Formal specification (JML)	10.0 %
Testing JML specs with JUnit	11.2 %
Formal verification with JMLUnit/JMLUnitNG	0.1 %
Formal verification with OpenJML	3.2 %
Formal verification with JJBMC	3.3 %
Formal verification with KeY	26.6 %
Writing (thesis)	26.2 %
Presentations	4.2 %
Software installation	2.2 %
Research (literature)	7.8 %
Meetings	5.2 %
Total	100.0 %

Table 4.1: Effort ratio of the project

As discussed in section 4.2, it is difficult to quantify the effects of the hybrid approach we took during the project (i.e. measure the benefits and costs and compare them on the same basis). Nevertheless, we are confident that the testing and verification effort with JUnit, JMLUnit/JMLUnitNG, OpenJML, and JJBMC combined (17.8%) would easily be exceeded by the extra effort needed for formal verification with KeY, had we not taken the hybrid approach. Table 4.2 is a rough estimate of the costs and benefits of this approach.

Task	RE (hrs.)	CE (hrs.)	EB (hrs.)	NR (hrs.)
Testing JML specs with JUnit	77	51	100	+49
Formal verification with JMLUnit/JMLUnitNG	1	1	0	-1
Formal verification with OpenJML	22	22	0	-22
Formal verification with JJBMC	23	10	20	+10
Total	123	84	120	36

RE: Raw Effort (the number of hours from a time table of the project), **CE**: Corrected Effort (effort after correcting for fair comparison), **EB**: Estimated Benefit, **NR**: Net Result.

Table 4.2: Costs and benefits of the hybrid approach

The second column (RE) contains the 'raw effort' in hours, i.e. the hours that were taken straight from the time table we recorded during the project. To make a fair comparison, we should correct some of values in this column. Firstly, not all methods were verified with KeY during the project. We should correct the testing effort by estimating how much time we spent on methods we in fact also verified. We estimate that about 33% of the testing effort was spent on methods we did not verify. Therefore, we downgraded the number of testing hours from 77 to 51 in the third column (CE). Secondly, the time spent on JJBMC was partly spent on testing and improving the tool itself. We corrected the number of 23 hours to 10 (column CE). Column EB contains the estimated benefit, i.e. the time gained, due to performing preliminary sanity checks on the JML specifications. The column on the right (NR) contains the net result (EB - CE).

We estimate that, due to our hybrid approach, we spent roughly 36 hours less on (partly) formally verifying the `IdentityHashMap`. Compared to the total amount of hours spent on specification with JML and verification with KeY, 69 hrs. and 182 hrs. respectively, according to our time table, this comes down to a ≈12.54% increase of efficiency (36 / (36 + 69 + 182) * 100%). Again, we stress that this is a rough estimate.

5

CONCLUSIONS AND RECOMMENDATIONS

5.1. CONCLUSIONS

We showed that, with JML and KeY, the most important methods of the `IdentityHashMap` can be formally specified and verified and, in doing so, demonstrated that an overflow error exists in the `capacity` method of the JDK7 version of the class. We implemented a fixed version of this method and proved its correctness by testing and formally verifying it. We also did this for the JDK9 version of the `capacity` method, which is a completely rewritten version.

We also demonstrated three ways to limit the effort of formally specifying contracts, each having its own advantages and disadvantages. In general, we feel this hybrid approach to formally analysing software proved to be beneficial in tackling specification errors in an early stage of the process. Of the tools we used to validate early versions of our specification, JUnit and JJBMC turned out to be the most beneficial. This does not mean, however, that, in general, it can be stated that OpenJML is not a useful tool for this purpose. (Boerman *et al.* [12] have shown results to the contrary.)

5.2. LESSONS LEARNED

We used JUnit and Java Reflection to test our JML contracts. The approach we have taken could, with hindsight, be improved. After we formally specified the bigger part of the `IdentityHashMap`, we wrote unit tests for every specified method, and tested them. We now think we would have been more efficient when limiting the number of tests to the somewhat more complex methods and the class invariant. There are several very simple pure methods (e.g. `isEmpty`, `size`, `maskNull`, `unmaskNull`, et cetera) that have very simple contracts that are, due to their simplicity, less prone to errors. Writing and maintaining unit tests for these methods might not be worthwhile, and should perhaps be considered only if problems during verification occur.

A similar remark could be made with regard to writing JML contracts. We started off with writing formal specifications for almost all methods, including those of the inner classes of the `IdentityHashMap`, as well as the class invariants of the inner classes. Again, with hindsight, we would have been more efficient by focussing on the specification of the class invariant of the main class and the most relevant methods of the CUA first. After testing and verifying them, time permitting, we could have extended our focus to the rest of the

CUA. When finishing our project, there were several specified methods and class invariants we did not formally verify because of a lack of time.

The KeY version we used (2.7) did not support certain modern Java features like, for example, generics or lambda expressions [1]. Generics are available since Java 7, but can be stripped by a KeY plugin for Eclipse. Lambda expressions were introduced in Java 8, and could not be stripped. We therefore decided to formally analyse the JDK7 version of the `IdentityHashMap`, and strip the generics [2]. When looking for a suitable candidate class for formal analysis, we did not check if later versions of the class had been improved or (partly) rewritten. If a candidate for formal analysis turns out to be significantly changed in later Java versions, that would be a valid argument to look for a better candidate [3]. It is, therefore, recommendable to check later versions of any formal analysis candidate class beforehand, in cases like these.

5.3. Future work

Contrary to data on the performance of software processes in general [33], as well as data on the costs of bugs in software [17, 18], there seems to be no statistical data available in literature on the effort and benefits of formal analysis specifically. It would have been interesting to have such information to see if formal analysis 'pays off'. Also, it would be interesting to see if a hybrid approach, where several tools are used in an attempt to limit the effort that goes into specification, like we did in this project, is beneficial, and to what extend. Indeed, we think it is beneficial, but it is, obviously, good (if not indispensable) to have more empirical data available to back this up. These data should take into account several factors, like the formal analysis method (model checking, abstract interpretation or deductive reasoning), tooling, complexity of the software, skills and experience, automatic versus interactive verification, et cetera.

Currently, KeY supports most JML features [4], but the supported features of Java 7 and later are limited [5]. A version of KeY supporting more recent features of Java would be a great improvement to the tool. Furthermore, we have not attempted to formally analyse the `readObject` and `writeObject` methods, because serialisation is not supported by KeY. We would welcome a future version of the tool that does. In our project this would have been a much appreciated feature, because the `readObject` method might probably have suffered from the overflow error we found in the `capacity` method, and was also not immune to some tampering with the serialised `IdentityHashMap` (see also: section 3.7.1 (An improved version of the readObject method) on page 62).

[1] At the time of writing this thesis, newer releases of KeY still did not support these features.

[2] We did make an attempt to prepare the JDK15 version of the `IdentityHashMap`, but the KeY plugin did not allow libraries and source code incompatible with Java7.

[3] Later versions of the `IdentityHashMap` have mainly been extended, while existing methods remained unaltered. Unfortunately (or fortunately, depending on one's perspective) the `capacity` method, containing the overflow error we detected, was rewritten in JDK9. The JDK9 version did no longer contain the error.

[4] See: https://www.key-project.org/jml-support-in-key/

[5] See: https://www.key-project.org/applications/program-verification/

6

REFLECTION

In this chapter, I will briefly reflect on the process (section 6.1) and the delivered end product (section 6.2). I will conclude this chapter with some afterthoughts (section 6.3).

6.1. THE PROCESS

6.1.1. LEARNING JML

Formal analysis by deductive reasoning was new to me, until I started this project. I had to get acquainted with JML as well as the verification tools (KeY, OpenJML and JJBMC). Learning JML was not extremely difficult. Fortunately, I had some good tutorials at my disposal. Chapter 7 of the KeY Book [11] was a good introduction to JML, and the JML reference by Leavens *et al.* [9] proved to be a good and complete reference that I used a lot. Using the KeY plugin for Eclipse was also convenient, because it supports syntax highlighting for JML. (Unfortunately, the plugin does not support syntax highlighting for conditional JML [1]. As soon as the `+KEY` or `+OPENJML` tags are added, the JML annotations are interpreted as regular comments. Furthermore, during my regular Zoom meetings with Mattias Ulbrich, Alexander Weigl, Jonas Klamroth at KIT and my supervisor Stijn de Gouw, any issue I had with JML could be discussed and resolved. Nevertheless, even with all the good advice and help, it's not always straightforward to get your JML contracts right the first time. It usually takes a (verification) tool to detect the errors in the contracts, and this is where JUnit, OpenJML, JJBMC, and ultimately KeY came into play.

6.1.2. WORKING WITH KEY

As mentioned before (see table 1.1: Pros and cons of formal analysis methods on page 6) formal analysis using the deductive method is tedious and time consuming work. This is also confirmed in some literature [4, 14, 15, 19]. It is one thing to be told so, but quite another thing to actually experience it. Despite heeding the warning given to me by my supervisor, and seeing it being repeated in the literature I studied, I was still unpleasantly surprised by the amount of work involved.

Learning to work with KeY was a lot harder than learning JML, in my experience. I used the KeY Book [11] as the main tutorial, but one does not learn to work with KeY from a book. It takes a lot of experience, making mistakes, and 'getting your hands dirty'. Simple

[1] See section 3.4 (A preliminary check of the JML specifications with OpenJML) on page 47.

methods with simple contracts can be automatically proven, but as soon as contracts get more complex, the tool requires interaction to prove the code's correctness or to find the errors in the specification. KeY has a lot of options and configuration settings, and without the dedicated meetings with Stijn, I would not have been able to separate the essential and useful ones from the less important ones. At the same time, for Stijn to know what I had already found out myself and what not, must also not have been obvious. For example, during the project it turned out I missed an essential setting in the so-called *taclet options* to guarantee that a proof is sound. This meant I had to repeat the proof for a number of methods. In short, the learning curve of KeY was a lot steeper than I estimated, and, although my skills seemed to improve throughout the project, I still would not dare to claim I 'master' it.

6.1.3. Contact with peers

Besides the regular one-on-one meetings with my supervisor, Stijn de Gouw, I've had several Zoom meetings with our German peers Mattias Ulbrich, Alexander Weigl and Jonas Klamroth, at KIT. I found these meetings very fruitful. They were pleasant, educational and motivational. Alexander was a great help at setting up KeY and helping me out with some technical issues. Mattias provided me with useful suggestions on JML, in some cases by live programming ('ruining' my specs with insightful ideas, which I later worked out), but also involved others like Jonas Klamroth and Bachelor student Christian Jung (who worked on a similar project), and sent me articles and theses on related subjects. Jonas wrote part of the OpenJML-specific (conditional) JML, based on my KeY specific contracts, to use with JJBMC. One Friday afternoon, together we detected a specification error that was repeated in several method contracts. I am happy I could (to some extend) return the favour by doing some tests with JJBMC on my computer. Late in the project, Christian Jung was introduced to the meetings by Mattias. A companion with similar struggles, which was, in a way, comforting. Especially at times when I felt I made too little progress.

I also had contact with David Cok, the developer of OpenJML. In my attempts to put the tool to use, I ran into some issues like, for example, a bug in the tool itself. I emailed David to report this, and to share some of the other issues I ran into. I got a quick reply with useful suggestions. Although my use of OpenJML was somewhat limited, I nevertheless appreciated his engagement.

6.1.4. My progress

At the start of the project, I was warned that my planning was quite optimistic. So I took into account that the project would probably take longer than originally estimated. However, even my readjusted expectations were exceeded. Indeed, progress was sometimes sluggish. And this, from time to time, made me feel somewhat embarrassed during the progress meetings I had with my supervisor and our German peers in Karlsruhe. They, on the other side, emphasized that formal analysis of a class like the `IdentityHashMap` isn't trivial, which was, to some extend, reassuring. David Cok wrote in an email to me: "I will say that java collections are a difficult place to start as their semantics is intricate." With hindsight I would have done some things more efficiently, as I addressed in section 5.2 (Lessons learned) on page 71.

6.2. THE END RESULT

The KeY tool, I think, is rightfully called a state-of-the-art tool. It is quite a challenge to develop a formal verification tool in general, and once one gets to know KeY and has acquired the skills to work with it interactively, this shows. Notwithstanding my deepest respect for the people who developed (and still develop) the KeY tool, I think it is a pity that it does not support a number of Java features. I was, therefore, not able to formally analyse a recent version of the `IdentityHashMap`. Instead, I was forced to stick with the JDK7 version, which contained an error that had (unfortunately for me) been fixed since JDK9. The work would have been more gratifying if I had found an error that is still present, and provide a fix for that error. However, to be fair, the error found in the `capacity` method is not serious enough to make the `IdentityHashMap` actually crash.

Appendix C shows which methods have been formally verified with KeY. Although most of the methods of the main class have been specified with JML, I did not succeed to verify all methods. This implicitly means that the JML contracts of these methods may still contain errors (due to the incremental way of working). I think it's a pity this wasn't accomplished. Although, on the upside, the number of methods (and, implicitly, class invariants) proven with KeY surpass those proven with OpenJML and JJBMC, and among the proven methods there are several that are typical for a map. I was particularly pleased I was able to prove the `put` method, which wasn't trivial and took quite an effort to complete. Indeed, I was on the verge of giving up on this one. I'm glad I did not.

6.3. SOME AFTERTHOUGHTS

In the introduction of this thesis, I substantiated my decision to cover the subject of formal analysis. In short, I reasoned that, however formal analysis (especially by deductive reasoning) can be tedious work, it can be worthwhile, especially in case of software libraries used in countless applications. Although I still stand by this viewpoint, part of me wonders whether or not, in practice, formal analysis has predominantly been an academic exercise so far. For that reason I suspect that, in my daily profession as software developer, chances are slim I will be applying much of the newly acquired knowledge during this project. On the upside, performing formal analysis of a 'real-life' class in the Java Collections Framework does affect one's level of awareness of (possible) software bugs, and (on a personal level) builds perseverance and character.

A

FORMAL ANALYSIS METHODS

The three principal formal validation methods mentioned in literature [4, 5] are *model checking*, *abstract interpretation*, and *deductive methods*. This appendix provides a global description of each of three methods.

A.1. MODEL CHECKING

A.1.1. THE TECHNIQUE

Model checking is an automatic technique for specifying finite state concurrent systems [34]. In model checking the design of the system is converted into a model, that is accepted by a model checking tool. Conversion into a model is sometimes an automated compilation task [1], but in other cases an abstraction of the design is made, leaving out irrelevant details to save time and memory. The properties that have to be verified by the model checker then need to be specified. Commonly, this is done by applying temporal logic, asserting how the system should behave over time. The main challenge in model checking is that it suffers from the possibility of a *state space explosion*, a problem that can occur when modelling a system under test, containing many interacting components that can assume many different values, resulting in an enormous amount of different possible states of that system. [34, 36].

A.1.2. RELATED WORK

Klaus Havelund *et al.* [35] have published work on the model checking approach while formally analysing and verifying Java software. They describe a translator JPF, that translates Java code into Promela, a modelling language of the model checker Spin. The Spin model checker is able to systematically check possible execution states for deadlocks, race conditions, unhandled exceptions and violations of any stated assertions. In their paper Klaus Havelund *et al.* describe how an example Java program has been translated automatically by JPF, and verified automatically by the model checker. This program, a bounded buffer, represents a finite structure that is being verified. Although the article presents a valuable insight in the workings of JPF and Spin, no open source or commercially sold software is being verified.

[1] Java Path Finder (JPF), for example, is a tool that translates Java into Promela, a modelling language for the model checker tool Spin [35]. (See section A.1.2.)

A.2. ABSTRACT INTERPRETATION

A.2.1. THE TECHNIQUE

Abstract interpretation was invented in the seventies of the previous century by Cousot *et al.* [37]. It is more scalable than model checking [4] and does, therefore, not suffer from the problem of state space explosion. The basic idea behind abstract interpretation is to describe a program's computation in terms of *abstract properties* and *abstract operations*. These abstractions of a program's concrete semantics make up the so-called *abstract domain*. Execution of the abstract operations in the abstract domain gives useful information on the actual program. The abstract semantics in the abstract domain are approximations, and therefore less precise [5]. This is not necessarily a bad thing, as is illustrated by the following simple example. Suppose we want to verify a simple multiplication function. Instead of running the actual function with many different variables (increasing the state space), we could also decide to focus on one aspect of the function, and just verify the sign of the product of the function. In the concrete domain the parameters are two integers consisting of an actual value and a sign. In the abstract domain we might choose to disregard the actual values of the parameters, and only consider their signs (+, – or 0). This kind of abstraction (*sign abstraction*) may lose some precision, but not for our purpose: here, it is sufficient to know the sign of the parameters[2]. Next, an abstract interpreter has to be designed to perform the computation in the abstract domain. This abstract function takes two parameters in the abstract domain, both being one of the three signs, +, – or 0, and returns the resulting sign. Some example invocations and results of the concrete function `mult` and its counterpart in the abstract domain, `mult'`:

```
 mult(3, 4)  =  12        mult'(+, +)  =  +
 mult(0, 4)  =  0         mult'(0, +)  =  0
 mult(3, 0)  =  0         mult'(+, 0)  =  0
mult(-3, 4)  =  -12       mult'(-, +)  =  -
 mult(3, -4) =  -12       mult'(+, -)  =  -
mult(-3, -4) =  12        mult'(-, -)  =  +
```

Abstract interpretation is a 'lattice-theoretic framework' [4, 37], meaning abstract properties are modelled by a complete semi-lattice. To understand how this works, recall the multiplication example above. More specifically, the sign abstraction performed on the integer input parameters. Abstraction means: mapping a set of properties in the concrete domain to a property in the abstract domain. This is referred to as the *abstraction function*, denoted as α. Just as finding an abstract approximation for a concrete property requires an abstraction function, finding a corresponding concrete set for any property in the abstract domain requires a *concretisation function*, denoted as γ. The concrete domain as well as the abstract domain can be represented by a lattice. Figure A.1 shows two lattices. To the left, a representation of the concrete properties in the concrete domain of the multiplication function $L = \langle P, \sqsubseteq_L \rangle$ is shown, with P as the set of tokens containing zero or more concrete properties, and $\sqsubseteq$ as the semi-lattice operation. The right lattice $L' = \langle P', \sqsubseteq_{L'} \rangle$ similarly represents the abstract domain, containing the abstract properties (stored in the tokens in P').

[2] Note, however, that for additions and subtractions this abstraction will lose too much precision, because it is impossible to know the sign of a these operations when the parameters are positive and negative, respectively.

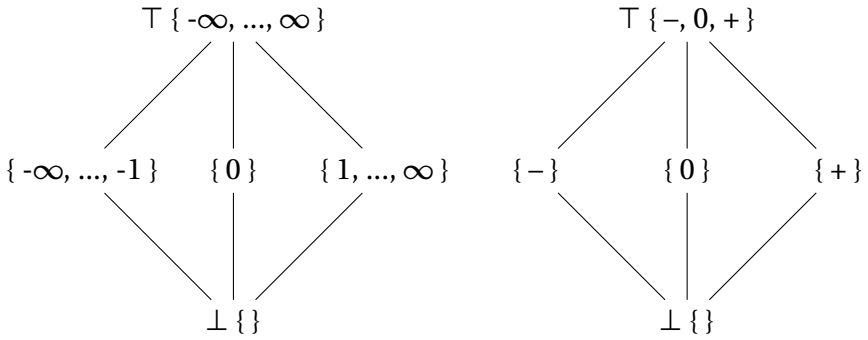


Figure A.1: Two lattices. On the left, $L = \langle P, \sqsubseteq_L \rangle$, a representation of the concrete properties in the concrete domain. On the right $L' = \langle P', \sqsubseteq_{L'} \rangle$, a representation of the abstract properties in the abstract domain.

The abstraction function α maps concrete values (integers) associated with tokens in P to an abstract token in P'. The concretisation function γ is, in a sense, the inverse of α, and maps a token from P' to a set of concrete properties in L. For example:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \alpha(\{3, 4\}) & = & + & \gamma(\top) & = & \{ -\infty, ..., \infty \} \\ \alpha(\{0, 3, 4\}) & = & \top & \gamma(-) & = & \{ -\infty, ..., -1 \} \\ \alpha(\{-3, -4\}) & = & - & \gamma(+) & = & \{ 1, ..., \infty \} \\ \alpha(\{\}) & = & \bot & \gamma(\bot) & = & \{ \} \end{array}$$

The abstract function `mult'` is abstracted from the function `mult` and takes its parameters from the abstract domain. Applying γ to these parameters results in the corresponding concrete set, taken by the concrete function. Applying α to the result of the concrete function results in a corresponding value in the abstract domain. Therefore, if `mult'` is a correct abstraction of `mult`, we can conclude: $mult' = \alpha \circ mult \circ \gamma$. During the verification process, a verification tool iteratively performs the abstract function, starting with each variable bound to $\bot$ in L', and working its way up the lattice until a fixpoint (fixed point or invariant point) is reached.

Dwyer *et al.* [4] mention a number of drawbacks of abstract interpretation. For example, the over-approximating nature of abstract interpretation resulting in inconclusive error reports, and the significant mathematical expertise that is required. To illustrate a downside of the over-approximating nature, consider the `mult` example once more. Note that a verification tool will run the abstract function `mult'`, not the concrete function `mult`, and will therefore never detect that `mult(Integer.MAX_VALUE, 2)` returns -2 (an overflow error). In the abstract domain `mult'(+, +)` is always +, and the overflow error will go unnoticed. Based on these drawbacks, abstract interpretation, like model checking, is, not a suitable approach for our research.

Furthermore, Polikarpova *et al.* [16] remark that "full functional verification of realistic software still largely relies on interactive theorem provers, which require massive amounts of effort from highly-trained experts".

A.2.2. Related work

The abstract interpretation method was successfully applied by Lacan *et al.* [38] to analyse the Ariane 5 software (flight 502 and later). The challenge was to regain confidence in the software of onboard equipment after the dramatic crash of flight 501 on June 4, 1996 [3, 21].

A.3. DEDUCTIVE METHODS

A.3.1. THE TECHNIQUE

In deductive verification so-called Hoare triples play a central role. According to Hoare [6], the validity of the result R of a program Q depends on the state of the system before Q is initiated (precondition P). Hoare introduces the notation

$$P\{Q\}R,$$

meaning "if the assertion P is true before initiation of a program Q, then the assertion R is true on its completion". To support several constructs of an imperative programming language, Hoare provides a number of axioms and inference rules. Two examples are the rule of composition and the rule of iteration, respectively:

Rule of composition:	if $\vdash P\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R$ then $\vdash P\{Q_1;Q_2\}R$
Rule of iteration:	if $\vdash P \wedge B\,\{S\}P$ then $\vdash P\{\textbf{while } B \textbf{ do } S\}\ \neg B \wedge P$

In the rule of iteration, P is called the loop invariant, because it holds before and after every loop iteration.

```
1  class C {
2    /*@ Invariant I @*/
3
4    /*@ requires P
5      @ ensures R
6      @*/
7    void m(Object o) {
8      ...
9      // method body
10     ...
11     /*@ loop_invariant L @*/
12     while (B) {
13       ...
14       // Loop body
15       ...
16     }
17     ...
18   }
19 }
```

Listing A.1: Java + JML example

In JML, a precondition of a method is denoted by the keyword `requires` (see line 4 in listing A.1). The postcondition is denoted by the keyword `ensures` (line 5). When (in the example above) `P` holds before method `m` is called, then `R` must hold on its completion. The class invariant `I` (line 2) must hold before and after method `m` is called. Loop invariant `I` (line 11) should hold before and after every iteration of the while loop.

As is clear from this simple example, with deductive verification we can stay close to the original syntax and semantics, when formally specifying its behaviour. A complete overview of the design of JML is written by Leavens *et al.* in their JML reference manual [9].

A.3.2. RELATED WORK

See: section 1.3 (Related work) on page 7.

B

DEDUCTIVE VERIFICATION TOOLS

This appendix provides a brief description of three deductive verification tools: Isabelle/HOL, Coq and KeY.

B.1. ISABELLE/HOL

Isabelle/HOL is an interactive general purpose theorem prover. HOL stands for higher-order logic. The tool allows for the definition of deductive systems for different logics [39]. Defining a logic in Isabelle constitutes the creation of a so-called *theory*. Theories are stored in files with the extension `.thy`. Any newly defined logic is called an *object-logic*, and is derived from Isabelle's *meta-logic* called `Pure`.

To get a sense of how Isabelle works, we will give a simple example in which we prove that an add function adding two natural numbers is associative. Figure B.1 shows the editor window of the Isabelle tool (Isabelle 2019 for Mac OS X)[1], with the theory file `TestAddAssoc.thy` open.

The theory defined in the file is (also) called `TestAddAssoc` (line 1), and imports the `Main` library of Isabelle. On line 5, a function `add` is defined. The functional programming language used here is Meta Language (ML), which is the default language in Isabelle.

To run the function, the keyword `value` can be used, as can be seen on line 10. Whenever the cursor is positioned at this line, the expression "`add 1 (add 2 3)`" is executed, resulting in the following output:

```
"6"
:: "nat"
```

This is, of course what is to be expected: 1 + 2 + 3 = 6.

Next comes the part where the associativity of the function is proven. The function should be associative, just as the mathematical + operator is. We do this by defining a theorem, in this example called `add_assoc` (line 12 in figure B.1). In this example `add_assoc` is defined as a *simplification goal*, as can be seen by the postfix `[simp]`, enabling us to re-use the goal in other theorems. Right of the semicolon, we define what we want to prove: "`add n (add m p) = add(add n m) p`" (associativity). Next, we apply the induction tactic on `n` (line 13 and 14). With every step in the symbolic run of the function, a different instance

[1] Isabelle 2019 for Mac OS X was downloaded from https://isabelle.in.tum.de/index.html

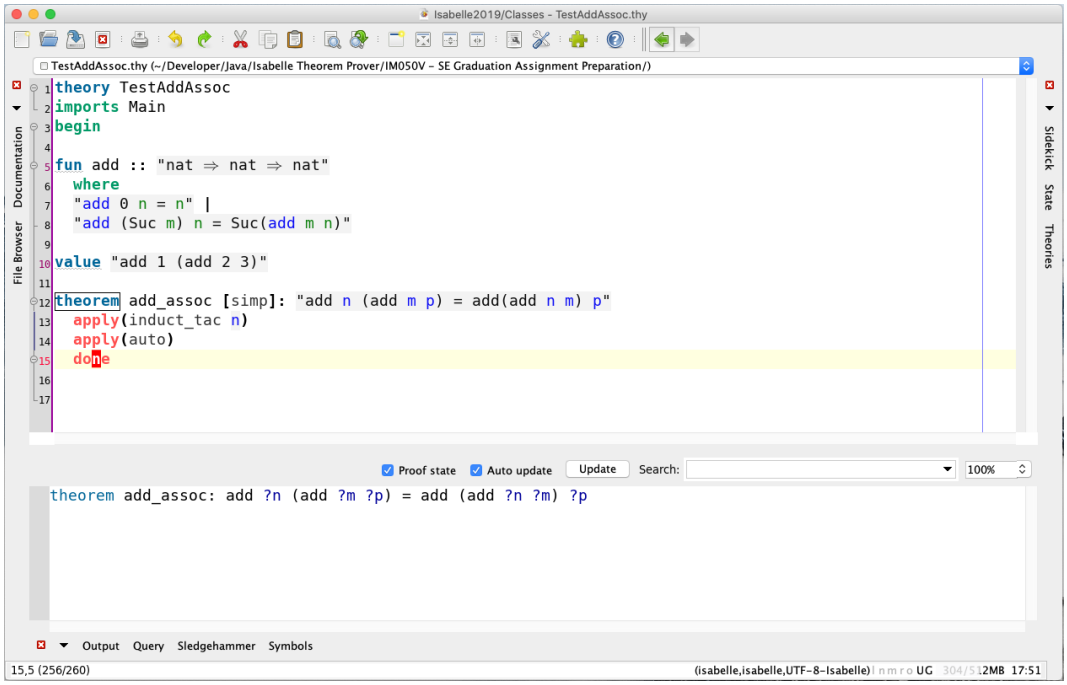


Figure B.1: TestAddAssoc.thy in Isabelle/HOL

of `n` will be used to determine the correctness of the function. By positioning the cursor on line 15, the output in the output panel of Isabelle (the lower half of the screen) in figure B.1 is displayed, indicating the theorem has been proven.

This, however, is not always the case right away. In most cases the goal of a theorem is not satisfied, and one or more subgoals have to be satisfied first. This can be done by writing one or more lemmas. Figure B.2 shows another theory file, called `TestAddCommut.thy`, containing a theory called `TestAddCommut` in which an attempt is made to prove the commutative character of the function `add`. Here, the theorem `add_commut` fails to finish, because Isabelle needs us to satisfy two other subgoals. This means we have at least two lemmas to define. In figure B.3 one lemma is added, partly satisfying Isabelle. When, eventually there are no more subgoals left to satisfy, the theorem can be proven, like in the first example in figure B.1.

Being able to write one's own logics is not intuitive, and requires some training. As an alternative one might use predefined logics, assuming that these are sufficient. Klein *et al.* [40] provide a listing of the Isabelle sources for μJava. μJava is a reduced model of Java Card. The model presented is quite extensive, hard to grasp, and does not cover all the Java features. In short, applying Isabelle/HOL for the purpose of our research project has a number of shortcomings: its learning curve is quite steep, Isabelle/HOL isn't tailor-made for Java, and the available theories/logics for Java that are available lack the required features. During our exploratory research into Isabelle/HOL we have not been able to successfully parse and verify a small example Java class, within the limited time available.

B.2. Coq

Like Isabelle/HOL, Coq is an interactive theorem prover. The CoqIDE enables the user to interactively write theorems and definitions for Coq to prove. Both tools are, to a certain

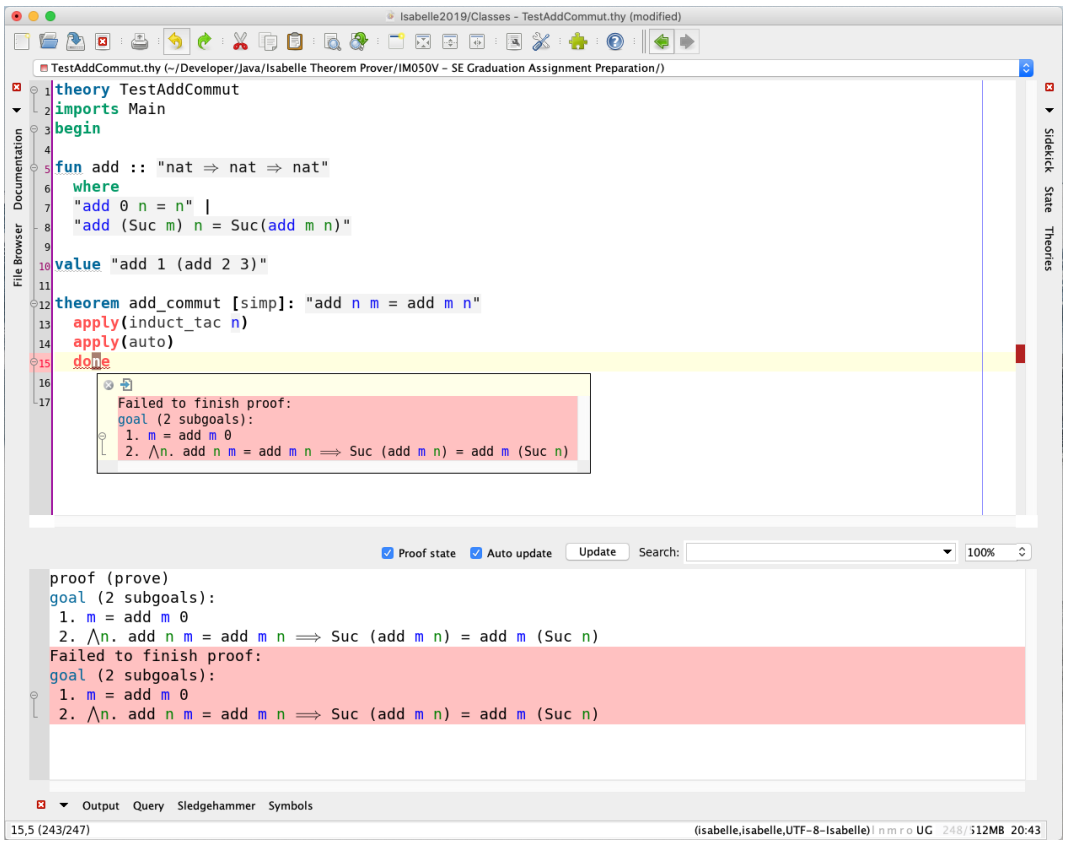


Figure B.2: TestAddCommut.thy in Isabelle/HOL

extend, quite similar. There are, however some small differences. To demonstrate some of the similarities and differences, the example of proving the associativity of a function `add` is repeated, using CoqIDE (CoqIde_8.4pl5 for Mac OS X)[2].

Figure B.4 shows the CoqIDE, containing this example. Lines 1 to 5 contain the definition of the function `add`, accepting two natural numbers as input, and returning a natural number as output. Although the language used in CoqIDE is (like in Isabelle/HOL) a functional language, the syntax is somewhat different. This is because Coq uses a different specification language, called Gallina, whereas Isabelle/HOL uses ML.

Below the function, a theorem `add_assoc` is defined, on lines 7 and 8. Again, this is quite similar to the way it is being done in Isabelle/HOL. Syntactically, there are similarities, but also the interactive way of working with Coq is reminiscent of what we have seen with Isabelle/HOL. Proving the theorem is done on lines 9 to 13 by introduction variables `a`, `b` and `c`, and applying induction on `a`. With every step of the proof, CoqIDE will indicate whether there are any subgoals to satisfy. For example, after running the code lines 1 to 10 (displayed in green), the output panel of the tool indicates there is one subgoal that needs to be satisfied (see figure B.5).

By completing the necessary steps, CoqIDE will eventually indicate that there are no more subgoals, and that the proof is complete. We can conclude our proof with the `Qed.` statement, indicating the proof is complete (see figure B.6).

Like, Isabelle/HOL, Coq is not tailor-made for Java, and suffers from similar drawbacks.

[2]CoqIDE_8.4pl5 for Mac OS X was downloaded from https://coq.inria.fr/coq-84

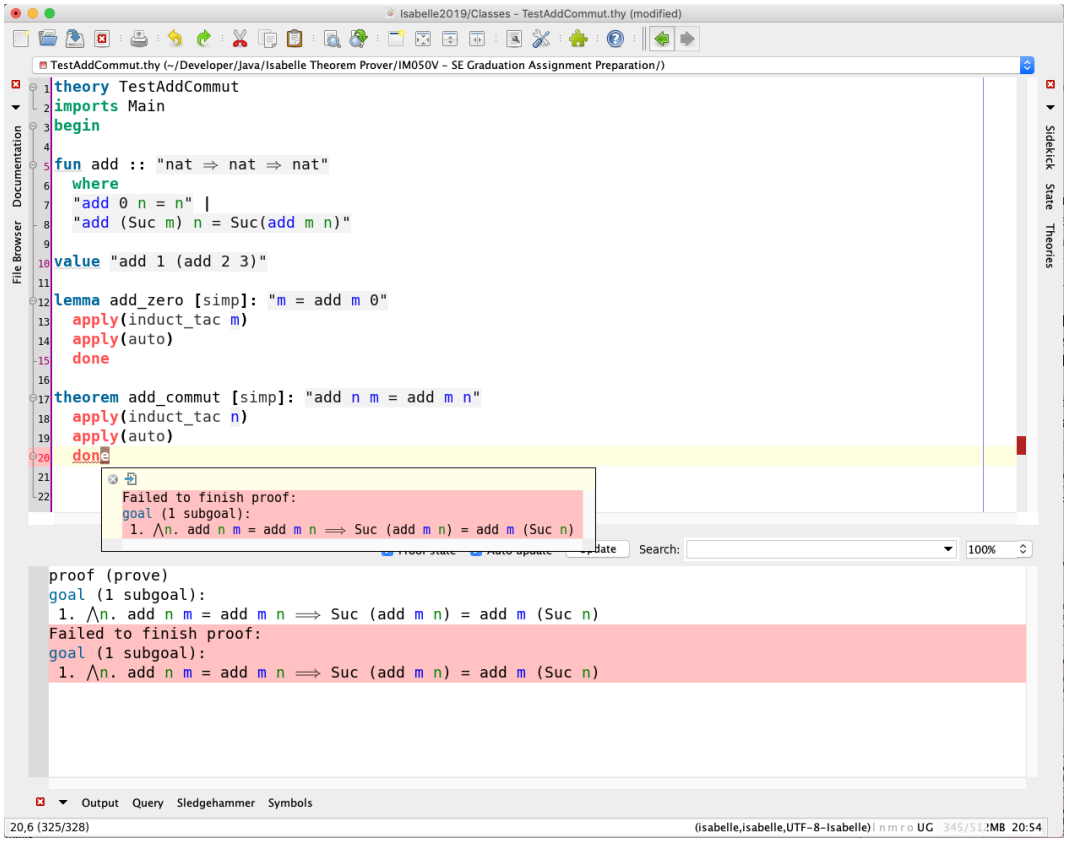


Figure B.3: TestAddCommut.thy with one lemma added in Isabelle/HOL

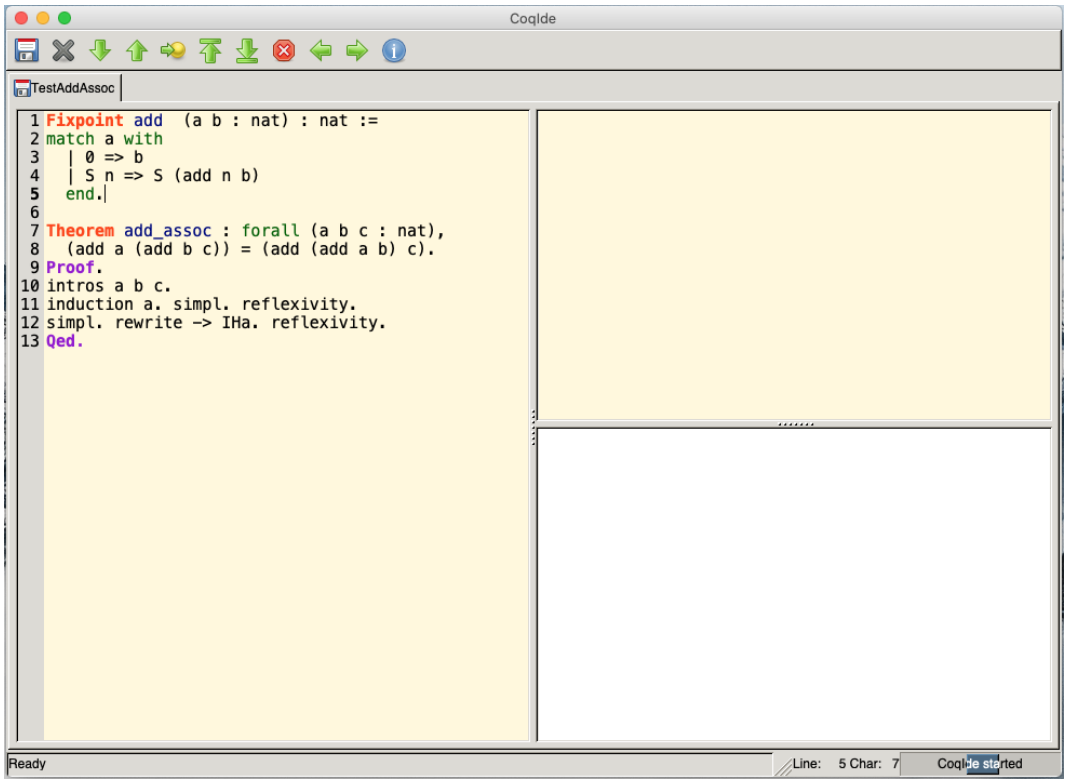


Figure B.4: TestAddAssoc in CoqIDE

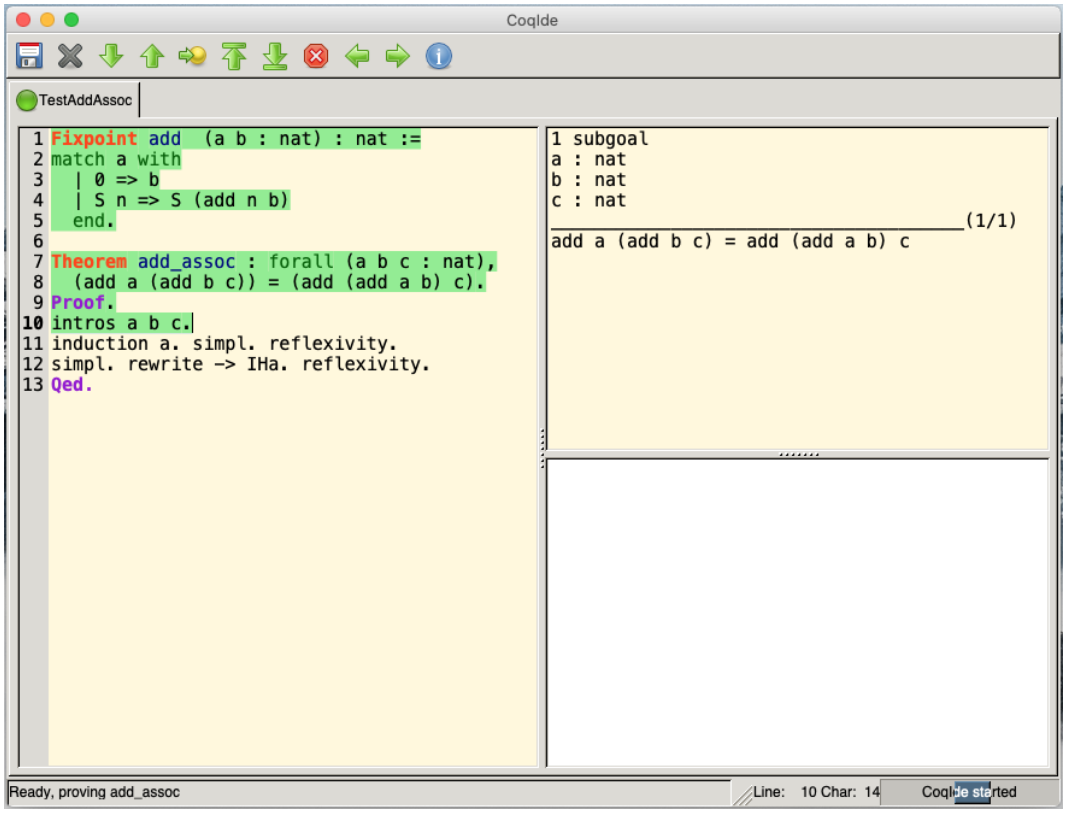


Figure B.5: TestAddAssoc in CoqIDE - 1 unsatisfied subgoal

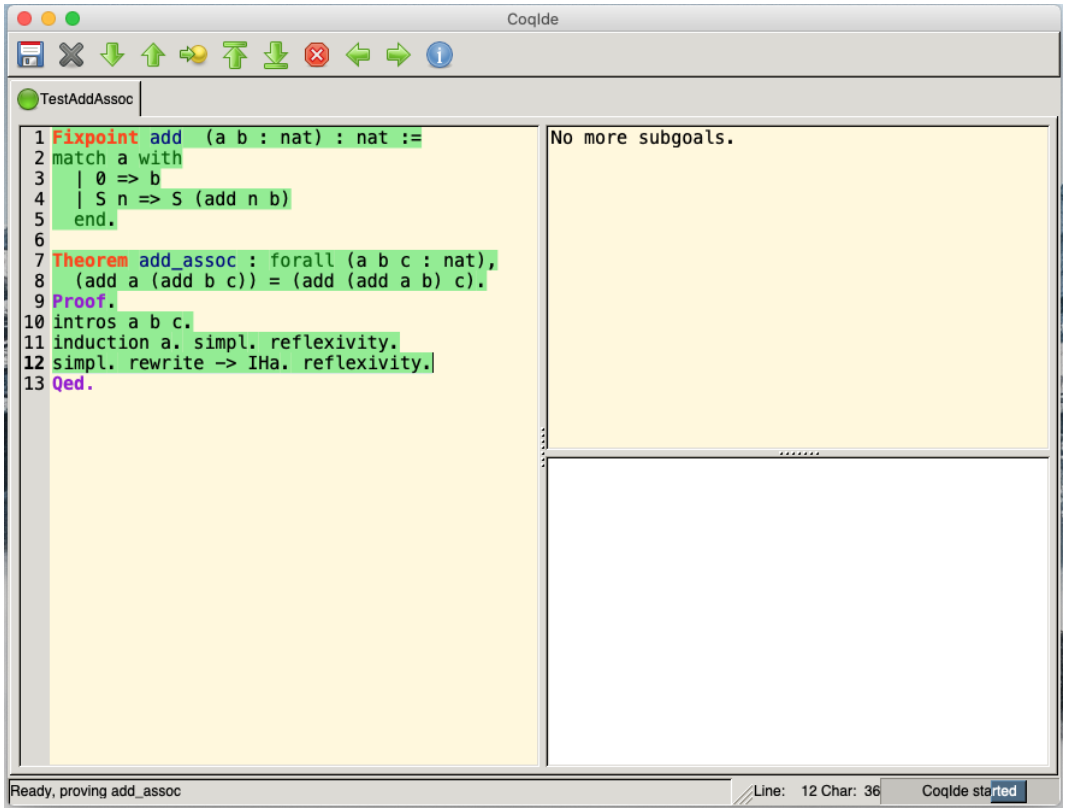


Figure B.6: TestAddAssoc in CoqIDE - no more subgoals

However, as Burdy *et al.* [23] and Filliâtre *et al.* [41] indicate, it is possible to generate proof obligations for Coq using *Krakatoa*, a tool that supports the use of JML. However, this would require extra learning efforts to acquire the skills and knowledge to work with yet another tool.

B.3. KeY

The KeY tool is developed by the KeY project[3], and is part of the KeY framework. The KeY tool is an interactive theorem prover, and is suitable for JML and Java. A description of KeY can be found in section 2.1.8 (The KeY tool), on page 17.

[3] https://www.key-project.org/

C

SPECIFIED, TESTED AND PROVEN METHODS AND INNER CLASSES OF THE IDENTITYHASHMAP

During the project we kept track of the methods and inner classes we specified, which specs were tested with JUnit, and which contracts were successfully verified with OpenJML, KeY and/or JJBMC.

C.1. PROVEN METHODS OF THE IDENTITYHASHMAP

The methods of the `IdentityHashMap` (main class) that were specified with JML, proven and for which the specifications were tested are shown in the table below. None of the methods were fully successfully verified with OpenJML, and most of the methods are verified with KeY and JJBMC.

Method	JML	JUnit	OpenJML	KeY	JJBMC
Class invariant	✓	✓		✓	
Object maskNull(Object)	✓	✓		✓	✓
Object unmaskNull(Object)	✓	✓		✓	✓
IdentityHashMap()	✓	✓			
IdentityHashMap(int)	✓	✓			
int capacity(int)	✓	✓		✓	
void init(int)	✓	✓		✓	✓
IdentityHashMap(Map<K,V>)	✓	✓			
int size()	✓	✓		✓	✓
boolean isEmpty()	✓	✓		✓	✓
int hash(Object, int)				✓ [1]	
int nextKeyIndex(int, int)	✓	✓		✓	✓
V get(Object)	✓	✓		✓	
boolean containsKey(Object)	✓	✓		✓	
boolean containsValue(Object)	✓	✓		✓	✓
boolean containsMapping(Object, Object)	✓	✓		✓	
V put(K, V)	✓	✓		✓	
void resize(int)	✓	✓		✓	

[1] Only one of two contracts proven

void putAll(Map<K,V>)	✓	✓			
V remove(Object)	✓	✓			
boolean removeMapping(Object, Object)	✓	✓			
void closeDeletion(int)		✓			
void clear()	✓	✓		✓	✓
boolean equals(Object)					
int hashCode()					
Object clone()	✓	✓			
Set<K> keySet()	✓				
Collection<V> values()	✓				
Set<Map.Entry<K,V> > entrySet()	✓				
void writeObject(ObjectOutputStream)					
void readObject(ObjectInputStream)					
void putForCreate(K, V)					

Table C.1: Proven methods of the IdentityHashMap per tool

C.2. SPECIFIED METHODS OF THE INNER CLASSES

The tables below show the methods of the inner classes that were specified, and for which the specifications were tested. Note that most of the methods were not verified with KeY. None of the methods were verified with JJBMC.

Method	JML	JUnit	OpenJML	KeY	JJBMC
Class invariant	✓	✓			
boolean hasNext()	✓				
int nextIndex()					
void remove()	✓				

Table C.2: Proven methods of the IdentityHashMap#IdentityHashMapIterator<T> per tool

Method	JML	JUnit	OpenJML	KeY	JJBMC
Class invariant	✓	✓			
K next()					

Table C.3: Proven methods of the IdentityHashMap#KeyIterator<T> per tool

Method	JML	JUnit	OpenJML	KeY	JJBMC
Class invariant	✓	✓			
V next()					

Table C.4: Proven methods of the IdentityHashMap#ValueIterator<T> per tool

Method	JML	JUnit	OpenJML	KeY	JJBMC
Class invariant	✓	✓			
Map.Entry<K,V> next()					
void remove()					

Table C.5: Proven methods of the IdentityHashMap#EntryIterator<T> per tool

Method	JML	JUnit	OpenJML	KeY	JJBMC
Class invariant	✓				
Entry(int index)					
K getKey()	✓				
V getValue()	✓				
V setValue(V)					
boolean equals(Object)					
int hashCode()					
String toString()					
void checkIndexForEntryUse()	✓				

Table C.6: Proven methods of the IdentityHashMap#EntryIterator#Entry per tool

Method	JML	JUnit	OpenJML	KeY	JJBMC
Class invariant					
Iterator<K> iterator()					
int size()	✓	✓			
boolean contains(Object)	✓	✓			
boolean remove(Object)	✓				
boolean removeAll(Collection<?>)					
void clear()	✓			✓	
int hashCode()					

Table C.7: Proven methods of the IdentityHashMap#KeySet per tool

Method	JML	JUnit	OpenJML	KeY	JJBMC
Class invariant					
Iterator<V> iterator()					
int size()	✓	✓			
boolean contains(Object)	✓	✓			
boolean remove(Object)	✓				
void clear()	✓			✓	

Table C.8: Proven methods of the IdentityHashMap#Values per tool

Method	JML	JUnit	OpenJML	KeY	JJBMC
Class invariant					
Iterator<Map.Entry<K,V» iterator()					
void clear()	✓			✓	
boolean contains(Object)	✓				
boolean remove(Object)	✓				
int size()	✓	✓			
boolean removeAll(Collection<?>)					
Object[] toArray()					
<T> T[] toArray(T[])					

Table C.9: Proven methods of the IdentityHashMap#EntrySet per tool

BIBLIOGRAPHY

[1] Federal Aviation Administration (FAA), "80 FR 24789 - Airworthiness Directives; The Boeing Company Airplanes," May 2015. ix, 2, 4

[2] M. Blair, S. Obenski, and P. Bridickas, "Patriot missile defense: Software problem led to system failure at Dhahran," *Report GAO/IMTEC-92–26*, 1992. ix, 2, 4

[3] B. Nuseibeh, "Ariane 5: who dunnit?," *IEEE Software*, no. 3, pp. 15–16, 1997. ix, 4, III

[4] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, and W. Visser, "Formal software analysis emerging trends in software model checking," in *2007 Future of Software Engineering*, pp. 120–136, IEEE Computer Society, 2007. ix, 4, 5, 73, I, II, III

[5] P. Cousot, "Abstract interpretation based formal methods and future challenges," in *Informatics*, pp. 138–156, Springer, 2001. ix, 4, I, II

[6] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969. ix, 5, 9, IV

[7] R. W. Floyd, "Assigning meanings to programs," in *Program Verification*, pp. 65–81, Springer, 1993. ix

[8] G. T. Leavens and Y. Cheon, "Design by Contract with JML," 2006. ix, 10, 11

[9] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, *et al.*, "JML reference manual," 2008. ix, 5, 23, 26, 73, IV

[10] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt, "The KeY tool," *Software & Systems Modeling*, vol. 4, pp. 32–54, Feb 2005. ix, 5, 7, 8

[11] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, "Deductive Software Verification - The KeY Book. LNCS, vol. 10001," 2016. ix, 15, 17, 73

[12] J. Boerman, M. Huisman, and S. Joosten, "Reasoning about JML: Differences between KeY and OpenJML," in *International Conference on Integrated Formal Methods*, pp. 30–46, Springer, 2018. ix, 29, 47, 48, 68, 71

[13] G. J. Holzmann, "Out of bounds," *IEEE Software*, vol. 32, no. 6, pp. 24–26, 2015. 2

[14] S. de Gouw, F. S. de Boer, R. Bubel, R. Hähnle, J. Rot, and D. Steinhöfel, "Verifying OpenJDK's Sort Method for Generic Collections," *Journal of Automated Reasoning*, vol. 62, pp. 93–126, Jan 2019. 3, 4, 5, 7, 21, 23, 73

[15] F. Pottier, "Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic," in *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, (New York, NY, USA), p. 3–16, Association for Computing Machinery, 2017. 3, 8, 73

[16] N. Polikarpova, J. Tschannen, and C. A. Furia, "A fully verified container library," *Formal Aspects of Computing*, vol. 30, pp. 495–523, Sep 2018. 3, III

[17] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Security & Privacy*, vol. 7, no. 2, pp. 87–90, 2009. 4, 72

[18] Cambridge University, "Cambridge University Study States Software Bugs Cost Economy $ 312 Billion Per Year.," 2013. 4, 72

[19] S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle, "OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case," in *Computer Aided Verification* (D. Kroening and Păsăreanu, eds.), pp. 273–289, Springer International Publishing, 2015. 4, 5, 7, 21, 73

[20] H.-D. A. Hiep, O. Maathuis, J. Bian, F. S. de Boer, M. van Eekelen, and S. de Gouw, "Verifying OpenJDK's LinkedList using KeY," in *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, to appear*, 2020. 4, 8, 21, 23

[21] G. Le Lann, "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective," in *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*, pp. 339–346, IEEE, 1997. 4, III

[22] D. Cok, "OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse," *Electronic Proceedings in Theoretical Computer Science*, vol. 149, Apr 2014. 5, 29

[23] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International journal on software tools for technology transfer*, vol. 7, no. 3, pp. 212–232, 2005. 6, 7, 29, X

[24] W. Mostowski, "Formalisation and Verification of Java Card Security Properties in Dynamic Logic," in *Fundamental Approaches to Software Engineering* (M. Cerioli, ed.), (Berlin, Heidelberg), pp. 357–371, Springer Berlin Heidelberg, 2005. 8

[25] Mostowski, Wojciech, "Fully verified Java Card API reference implementation," *Verify*, vol. 259, pp. 136–151, 2007. 8

[26] J. van Benthem, *Logica voor informatica.* Pearson Education, 2003. 9, 10

[27] King, James C., "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, p. 385–394, jul 1976. 13, 15

[28] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on software engineering*, no. 3, pp. 215–222, 1976. 13, 15

[29] M. Giese, "Taclets and the KeY prover," *Electronic Notes in Theoretical Computer Science*, vol. 103, pp. 67–79, 2004. 17

[30] M. T. Goodrich and R. Tamassia, *Data structures and algorithms in Java*. John Wiley & Sons, 2001. 23, 32

[31] D. A. Watt, *Programming language design concepts*. John Wiley & Sons, 2004. 29

[32] D. M. Zimmerman and R. Nagmoti, "JMLUnit: The Next Generation," in *Formal Verification of Object-Oriented Software* (B. Beckert and C. Marché, eds.), (Berlin, Heidelberg), pp. 183–197, Springer Berlin Heidelberg, 2011. 29

[33] A. Fuggetta and E. Di Nitto, "Software process," in *Future of Software Engineering Proceedings*, pp. 1–12, 2014. 72

[34] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*. MIT press, 2018. I

[35] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000. I

[36] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 553–568, Springer, 2003. I

[37] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, ACM, 1977. II

[38] P. Lacan, J. N. Monfort, L. Ribal, A. Deutsch, and G. Gonthier, "Ariane 5-the software reliability verification process," in *DASIA 98-Data Systems in Aerospace*, vol. 422, p. 201, 1998. III

[39] F. M. Dionısio, P. Gouveia, and J. Marcos, "Defining and using deductive systems with Isabelle," *Computing, Philosophy, and Cognition*, pp. 271–293, 2005. V

[40] G. Klein, T. Nipkow, D. von Oheimb, C. Pusch, and M. Strecker, "Java Source and Bytecode Formalizations in Isabelle: μJava," 2019. VI

[41] J.-C. Filliâtre and C. Marché, "The Why/Krakatoa/Caduceus platform for deductive program verification," in *International Conference on Computer Aided Verification*, pp. 173–177, Springer, 2007. X

ACRONYMS

AD airworthiness directive. XIX, 2

CFML Characteristic Formulae for ML, a tool for the interactive verification of OCaml programs, based on Coq. 8

COTS Commercial Off-The-Shelf. 3

CUA Class Under Analysis. 4–6, 18, 19, 26, 31, 33, 41, 50, 52, 64, 65, 68, 71

DBC design by contract. 10

FAA Federal Aviation Administration. XIX, 2

FZI Forschungszentrum Informatik. 30, 50

GCU general control unit. 2

HOL Higher-Order Logic. XX

HPC High Performance Computing. 64, 65

JBMC a software bounded model checker for Java. ix, XVIII, 30

JJBMC a tool that enables JBMC to verify contracts written in JML. ix, XI–XIV, 19, 23, 30, 47, 50–53, 65, 66, 68–71, 73–75

JML Java Modelling Language. ix, IV, X–XIV, XVIII, XX, XXI, 5–8, 10–12, 15, 17–19, 21–24, 26, 29, 30, 33–36, 38–42, 44–47, 49–54, 61, 64–75

JPF Java Path Finder. I

KIT Karlsruher Institut für Technologie. 30, 50, 73, 74

ML Meta Language. V, VII

OTS Off-The-Shelf. 3, 4, 7

SMT satisfiability modulo theory. 29

GLOSSARY

abstract interpretation An approach to formal analysis. An abstract approximation of the program to be analysed is designed to verify its correctness. ix, I–III, XIX, XX, 4–6, 72

abstract interpreter An abstract interpreter is an abstraction of a concrete function that is formally verified with abstract interpretation. It is a function in the abstract domain, and an approximation of its concrete counterpart. Its input are parameters in the abstract domain, and its output is the abstract state after execution. II

airworthiness directive An Airworthiness Directive is a notification to owners and operators of certified aircraft that a known safety deficiency exists and must be corrected. The FAA defines ADs as "legally enforceable regulations issued by the FAA in accordance with 14 CFR part 39 to correct an unsafe condition in a product". XVIII, 2

axiom An axiom is a formula that can be used anywhere in a mathematical proof. XIX

Coq An interactive, general purpose theorem prover, similar to Isabelle/HOL. V–VII, X, XVIII, XX, 7, 8

deductive method See deductive verification. ix, I, 4–6, 73

deductive reasoning See deductive verification. 72, 73, 75

deductive verification An approach to formal analysis. The term refers to a proof technique that uses propositional and first-order logic to deduce a conclusion from something that is known, using axioms and deduction rules (Modus Ponens). ix, IV, XIX, XX, 3, 5, 7, 9

dynamic logic A form of modal logic, and an extension to Hoare logic, that, besides correctness, supports program characteristics like termination, equivalence, and determinism. XX, 9, 10, 17

first-order logic Predicate logic. XIX, XXI

formal analysis A formal software analysis is a mathematically based, and therefore unambiguous, automated technique for reasoning about software semantics. It uses precise mathematical specifications of the intended behaviour of the software. Formal specification, testing and formal verification are part of the process of formal analysis. ix, XIX, XXI, 3, 4, 6, 7, 9, 13, 18, 19, 21, 23, 33, 64, 65, 68, 72–75

formal specification A formal specification of a system is written in a language whose semantics are unambiguous, because they are mathematically defined. ix, XIX, XX, 3, 8, 9, 17–20, 23, 64

formal verification Formal verification is the process of mathematically proving whether (the formal model of) a system behaves according to the formal specifications. Deductive verification is a specific form of formal software verification. Two other approaches are model checking and abstract interpretation. XIX, XX, 8, 17, 19, 20, 65, 66

Gallina A functional language, used as the specification language by Coq. VII

High Performance Computing High Performance Computing is aggregated computing power in which the whole is much more than the sum of a number of PCs. Systems that are at least 10,000 to 100,000 times faster than a regular desktop computer are referred to with the term supercomputer. XVIII, 64

higher-order logic Higher-Order Logic (HOL) is an extension of second-order logic. It can be regarded as a union of first-, second-, third-, ... n-th order logics. V

Hoare logic A formal system of logical rules for reasoning about the correctness of a computer program. XIX, XXI, 9

Isabelle/HOL An interactive, general purpose theorem-proving tool that supports the definition and use of deductive systems for many different logics. V–VII, XIX, 7

Java Card Java Card is a technology designed to incorporate Java in smart card programming. The Java Card programming language is a stripped down version of the Java language. VI, 8

Java Reflection A Java feature to access private components (fields, methods and constants) of Java classes and objects. This feature is exclusive to Java, and does not exist in other programming languages like C or Pascal, for example. It opens the possibility to inspect, or even manipulate, internal properties of a program. 19, 28, 29, 41, 47, 66, 71

JavaDL JavaDL is KeY's dynamic logic language for Java. It introduces formal formulas, based on JML contracts, that encode the correctness of Java methods, and function as proof obligations. 17, 35

JMLUnit JMLUnit is a unit testing framework for JML-annotated Java code. XX, 29, 69, 70

JMLUnitNG JMLUnit Next Generation. 19, 23, 29, 69, 70

JUnit JUnit is a unit testing framework for Java code. XI–XIV, 19, 23, 29, 41, 45, 47, 69–71, 73

KeY A state-of-the-art tool that provides facilities for formal specification with and formal verification of software. The target language of KeY-driven software development is Java. ix, V, X–XIV, XX, XXI, 5, 7, 8, 15, 17–23, 29, 31, 33–36, 38–41, 47, 48, 50, 51, 54, 61, 64–75

KeY plugin A plugin for Eclipse that provides support for creating KeY projects, JML sytax highlighting, generating stubs for related classes, stripping generics (which are not supported by KeY) automatically from classes, et cetera. 21

lattice See semi-lattice. II, III, XXI

modal logic Modal logic is an extension to other logics, like Hoare logic. Two new unary operators are introduced that represent so-called 'modalities'. These modal operators express necessity and possibility. For example, $\Box p$ asserts that p is necessarily true, and $\Diamond p$ asserts that p is possible true. 9

model checking A specific technique of formal analysis. It is an automated technique for checking a formal model to assess whether it adheres to formal specifications. It verifies the formally specified properties by exhaustively checking all possible states that a (software) system could enter during execution. In model checking the formal verification is a good as the model. Model-based testing is used to assess whether the model is an accurate representation of the actual system. ix, I, III, XX, XXII, 4–6, 72

Modus Ponens Modus Ponens, also Modus Ponendo Ponens (Latin for "mode that by affirming affirms"), is a rule of inference. From φ and $(\varphi \rightarrow \psi)$ we can deduce ψ. XIX

OCaml A hybrid programming language, that supports functional, imperative and object-oriented programming. XVIII, 8

OpenJDK Open Java Development Kit, an open-source implementation of the Java Platform, Standard Edition. 3–7, 18, 19

OpenJML OpenJML is a formal verification tool for Java programs, based on formal specifications annotated in JML. ix, XI–XIV, 19, 23, 29, 30, 36, 47–50, 52, 65–71, 73–75

proof obligation A proof obligation is a logical formula associated with a correctness claim for a verification property. XX, 7, 15, 17

second-order logic First-order logic, extended with additional quantifiers over second-order objects (sets, predicates, functions). XX

semi-lattice A semi-lattice is a mathematical structure $\boldsymbol{S}$ of the form $\langle S, \cdot \rangle$, where S is a set of partially ordered tokens, and $\cdot$ represents the semi-lattice operation, defining the binary relation between the tokens in S. II, XXI

semi-lattice operation An associative, commutative and idempotent binary operation, defining the binary relation between the tokens in a lattice. II, XXI

sequent A conditional assertion of the form $\varphi_1, ..., \varphi_n \circ \psi_1, ..., \psi_k$. The part left of the $\circ$ is referred to as the antecedent, and the right part is referred to as the consequent. It asserts that, if all of the formulas in the antecedent are true, then at least one of the consequent formulas must be true as well. XXI, 10, 17

sequent calculus A style of formal logical argumentation, based on sequents. 10

state space explosion State space explosion, or state explosion, is the main challenge to be dealt with in model checking. It is a problem that can occur when modelling a system with many interacting components that can assume many different values, resulting in an enormous amount of different possible states of that system. I, 3, 5

symbolic execution A testing and formal analysis technique where, contrary to normal execution of a program, where the input parameters consist of concrete values, execution is done with symbolic expressions. 13–15