Chapter 1
Quo Vadis Formal Verification?

Reiner Hähnle

The KeY system has been developed for over a decade. During this time, the field of Formal Methods as well as Computer Science in general has changed considerably. Based on an analysis of this trajectory of changes we argue why, after all these years, the project is still relevant and what the challenges in the coming years might be. At the same time we give a brief overview of the various tools based on KeY technology and explain their architecture.

1.1 What KeY Is

There is the KeY project, the KeY system, and a collection of software productivity tools based on the KeY system that we call the KeY framework (see Figure 1.1 below).

The KeY *project* is a long-term research project started in 1998 by Reiner Hähnle, Wolfram Menzel, and Peter Schmitt at University of Karlsruhe (now Karlsruhe Institute of Technology). After Menzel's retirement Bernhard Beckert joined the project leader team which has been unchanged ever since. This proves that long-term research on fundamental problems is possible—despite dramatic changes in the academic funding landscape—provided that the people involved think it is worthwhile. The question is: why should they think it is?

From the very first publication [Hähnle et al., 1998] the aim of the KeY project was to integrate formal software analysis methods, such as formal specification and formal verification, into the realm of mainstream software development. This has always been—and still is—a very ambitious goal that takes a long-time perspective to realize. In the following we argue why we are still optimistic that it can be reached.

The KeY *system* was originally a formal verification tool for the Java programming language, to be coupled with a UML-based design tool. Semantic constraints expressed in the Object Constraint Language (OCL) [Warmer and Kleppe, 1999] were intended as a common property specification language. This approach had to be abandoned, because UML and OCL never reached a level of semantic foundation

that was sufficiently rigorous for formal verification [Baar, 2003]. In addition, OCL is not an object-oriented language and the attempt to formally specify the behavior of Java programs yields clumsy results.

To formally specify Java programs KeY currently uses the result of another long-term project: the *Java Modeling Language* (JML) [Leavens et al., 2013] which enjoys wide acceptance in the formal methods and programming languages communities.

From the mid 2000s onward, a number of application scenarios for deductive verification technology beyond functional verification have been realized on the basis of the KeY system. These include test case generation, an innovative debugging tool, a teaching tool for Hoare logic, and an Eclipse extension that integrates functional verification with mainstream software development tools. Even though they share the same code base, these tools are packaged separately to cater for the different needs of their prospective users.

1.2 Challenges To Formal Verification

First, let us clarify that in this book we are concerned with *formal* software verification. With this we mean a formal, even mechanical argument, typically expressed in a formal logical system, that a given program satisfies a given property which also had been formalized. In contrast to this, most software engineers associate heuristic, informal techniques such as testing or code reviews with the term "verification" [Sommerville, 2015, Chapter 8].

Formal verification of nontrivial programs is tedious and error-prone. Therefore, it is normally supported by tools which might be dedicated verification tools, such as KeY, or general purpose interactive theorem provers such as Isabelle [Nipkow et al., 2002] or Coq [Dowek et al., 1993]. One can further distinguish between interactive tools where a verification proof is built in dialogue with a human user and tools that work in "batch mode," not unlike a compiler, where a program is incrementally annotated with specifications and hints until its correctness can be automatically proven. Examples of the latter are the systems Dafny [Leino, 2010] and VeriFast [Jacobs and Piessens, 2008]. A full discussion of possible architectures of verification systems, as well as their pros and cons, is found in the survey [Beckert and Hähnle, 2014].

For a long time the term *formal verification* was almost synonymous with *functional verification*. In the last years it became more and more clear that full functional verification is an elusive goal for almost all application scenarios. Ironically, this happened *because of* advances in verification technology: with the advent of verifiers, such as KeY, that mostly cover and precisely model industrial languages and that can handle realistic systems, it finally became obvious just how difficult and time consuming the specification of functionality of real systems is. Not verification but specification is the real bottleneck in functional verification [Baumann et al., 2012]. This becomes also very clear from the case studies in Chapters 18 and 19.

Even though formal verification of industrial target languages made considerable progress, *complete* coverage of languages such as Java, Scala, or C++ in any formal verification tool is still a significant challenge. The KeY tool, while fully covering Java Card (see Chapter 10), makes similar restrictions to most other verification tools: floating-point types are not supported, programs are assumed to be sequential, and generic types are expected to have been compiled away. On the other hand, Java integer types, exceptions and static initialization all are faithfully modeled in KeY.

Some of the restrictions will be, at least partially, addressed in the future. First approaches to formal verification of concurrent Java have been presented [Amighi et al., 2012] and are on their way into KeY [Mostowski, 2015]. On the other hand, to the best of my knowledge, all existing formalizations of Java concurrency assume sequential consistency. A full formalization of the Java memory model seems far away at this moment.

Floating-point types seem a more achievable goal, because formal models of the IEEE floating-point standard are available today [Yu, 2013] and it is only a matter of time until they are supported in verification tools. A more difficult question, however, appears to be how to formally specify floating-point programs.

A major challenge for all formal verification approaches, most of which are part of academic research projects, is the evolution of industrial languages such as Java. This evolution takes place at a too fast pace for academic projects with their limited resources to catch up quickly. Moreover, the design of new language features takes into account such qualities as usability, marketability, or performance, but never verifiability. Many recent additions to Java, for example, lambda expressions, pose considerable challenges to formal verification. Many others, while they would be straightforward to realize, do not yield any academic gain, i.e., publishable papers. This results in a considerable gap between the language supported by a verification tool and its latest version. KeY, for example, reflects mostly Java 1.5, while the latest Java version at the time of writing this article is 1.8.

A problem of a somewhat different kind constitute the vast APIs of contemporary programming languages. The API of a language is among its greatest assets, because it is the basis of programmer productivity, but it presents a major problem for program analysis: in general neither the source code of an API method is available nor a formal contract describing its behavior. For special cases like Java Card [Mostowski, 2007] or for specific aspects such as concurrency [Amighi et al., 2014b] it is possible to provide formal specifications of API classes, however, in general the usage of APIs poses a serious problem for program analysis. To be fair, though, this is the case not only in formal verification, but already for test case generation or much simpler static analyses. That formal verification of APIs is, in principle, feasible shows the recent complete verification of Eiffel's container library [Polikarpova et al., 2015].

The preceding discussion gives a somewhat mixed prospect for formal verification of functional properties of Java. So, why do we carry on with KeY? As a matter of fact, we believe that there are many good reasons and one can even argue that deductive verification is just beginning to become an interesting technology (see also [Beckert and Hähnle, 2014]). Let us see why.

1.3 Roles of Deductive Verification

A central insight from the last decade or so is that deductive verification technology is not only useful for functional verification, but is applicable to a large number of further scenarios, many of which avoid the problems discussed in the previous section.

1.3.1 Avoid the Need for Formal Specification

If full-fledged functional verification is no longer the main focus, but deductive verification technology is used for analyses that do not require to compare a program against its functionality, then the problem of providing formal specifications is at least alleviated or even vanishes completely.

For example, it is very useful to know that a program does not throw any runtime exceptions, that it terminates, or that it only accesses a certain part of the heap. Such *generic properties* can be specified in a uniform manner for a given program. In the last years a number of specialized verification tools appeared for this class of problems that scale up to real-world problems [Beyer, 2015]. Note, however, that it might still be necessary to provide detailed specifications in order to avoid too many false positives.

Closely related is the problem of *resource analysis*, where the best-case or worst-case consumption for a target program of a given resource is computed. Analyzed resources include runtime and memory, but also bandwidth or the number of parallel threads. Cost analysis tools involve complex constraint solving and typically do not include a formal semantics of the target language. Therefore, the question of soundness arises. In this context, verification tools such as KeY were successfully employed as “proof checkers” [Albert et al., 2012]. This is possible, because the resource analyzer can infer enough annotations (such as the invariants) required for automating the verification process and because resource properties can be expressed in a uniform manner.

About ten years ago, several research groups independently proposed to generate glass-box test cases with code coverage guarantees by symbolic execution of the program under test [Tillmann and Schulte, 2005, Engel and Hähnle, 2007, Albert et al., 2009].[1] In Chapter 12 we describe how test cases can be obtained from a verification attempt in KeY. The embedding into a deductive framework has a number of advantages over using symbolic execution alone:

- full first-order simplification can eliminate unreachable code and hence irrelevant test cases, in particular, when combined with preconditions;
- with suitable loop invariants and contracts even programs with loops and recursive programs can be fully symbolically executed, thus increasing coverage;

[1] None of them had realized at the time that this idea had in essence been already suggested by King [1976].

- test oracles can be specified declaratively and can be implemented by deductive inference.

Yet another application of deductive verification that can dispense with detailed specifications are *relational properties*. Relational properties became increasingly important as an application scenario for verification in recent years. They compare the behavior of two or more programs when run with identical inputs. The crucial point is that it is not necessary to fully specify the behavior of the target programs, but only compare their respective behaviors and ensure they maintain a certain relation (e.g., bisimilarity). That relation is typically fixed and can be expressed in a uniform manner for all given target programs. Therefore, the specification can either be written once and for all or at least it can be computed automatically. This observation was used, for example, by Benton [2004] to formalize program properties and the soundness of program transformations in relational Hoare logic.

Examples of relational properties include information flow [Darvas et al., 2003, 2005], where the property to be proven takes the form of a security policy; another is the correctness of compiler optimizations [Barthe et al., 2013a] and program transformations [Ji et al., 2013] where it must be shown that the original and the compiled/transformed program behave identically on all observable outputs. These scenarios are supported by KeY and are discussed in Chapters 13 and 14, respectively, in this book.

What makes relational properties attractive as an application of verification technology, besides the fact that extensive specifications are not needed, is the high degree of automation that is achievable. The main obstacle against automation is the need to provide suitable specification annotations in the form of loop invariants (or contracts in the case of recursive calls). To prove relational properties, the required invariants are often simple enough to be inferred automatically. In KeY a combination of symbolic execution and abstract interpretation proved to be effective [Do et al., 2016]. How this works is explained in Chapter 6.

Uniform specifications and simple invariants contribute towards automation in another crucial way: the resulting proof obligations tend not to require complex quantifier instantiations. Given sufficient support for reasoning over theories that occur in the target program, full automation is achievable for many relational problems. Chapter 5 explains by selected examples how theory reasoning has been integrated into the deduction machinery of KeY.

1.3.2 Restricted Target Language

In Section 1.2 we pointed out that real-world languages, such as Java, C++, or Scala with their vast scope, their idiosyncrasies and the dynamics of their development pose significant challenges to formal verification. One obvious reaction to this is to focus on programming languages that are smaller and less prone to change. Examples include Java Card [JavaCardRTE], a Java dialect for small, mobile devices, Real-Time Java [Bollella and Gosling, 2000], and SPARK [Jennings, 2009], an Ada dialect for

safety-critical software. For these languages complete formalizations, including their APIs exist. In KeY we support Java Card (see [Mostowski, 2007] and Chapter 10) and Real-Time Java [Ahrendt et al., 2012].

A different approach is to design a *modeling language* that retains the essential properties of the underlying implementation language, but abstracts away from some of its complexities. Verification of abstract models of the actual system is standard in model checking [Clarke et al., 1999]; PROMELA [Holzmann, 2003], for example, is a widely used modeling language for distributed systems. In the realm of object-oriented programming with concurrency the language ABS (for *Abstract Behavioral Specification*) [Johnsen et al., 2011] was recently suggested as an abstraction for languages such as Java, Scala, C++, or Erlang, several of which it supports with code generator backends. ABS has a concurrency model based on cooperative scheduling that permits compositional verification of concurrent programs [Din, 2014]. It also enforces strong encapsulation, programming to interfaces, and features abstract data types and a functional sublanguage. This allows a number of scalable analyses tools, including resource analysis, deadlock analysis, test generation, as well as formal verification [Wong et al., 2012]. A version of KeY that supports ABS is available [Chang Din et al., 2015].

The formal verification system KeYmaera[2] [Platzer and Quesel, 2008, Fulton et al., 2015] targets a modeling language that combines an abstract, imperative language with continuous state transitions that are specified by partial differential equations. It can be used to formally specify and verify complex functional properties of realistic hybrid systems [Loos et al., 2013] that cannot be expressed in model checking tools.

1.3.3 Formal Verification in Teaching

One important application area of formal verification is education in formal approaches to software development. Here the coverage of the target language and of the APIs are not a critical issue, because the teacher can avoid features that are not supported and is able to supply specifications of the required APIs.

On the other hand, different issues are of central importance when deductive verification systems are used in teaching. The main challenge in teaching formal methods is to convey to students that the learning outcomes are useful and worthwhile. Students should not be force-fed with a formal methods course they find to be frustrating or irrelevant. This puts high demands on the usability and the degree of automation in the used tools. It is also crucial to present the course material in a manner that connects formal verification with relevant every-day problems that one has to solve as a developer. Finally, one has to take care that not too many mathematical prerequisites are needed.

Since 2004 we use the KeY system in a variety of mandatory and specialized courses, where we try to address the issues raised in the previous paragraph. In

[2] KeYmaera branched off from KeY, see also Section 1.5.1.

2007 we created a B.Sc. level course called *Testing, Debugging, and Verification* [Ahrendt et al., 2009a] where we present formal specification and verification as part of a continuous spectrum of software quality measures. This being a B.Sc. course, we wanted to illustrate formal verification in Hoare style [Hoare, 1969], and with a simple while-language, not Java. To our amazement, we could not find any tool support with automatic discharge of first-order verification conditions (after all, we did not want to include automated theorem proving in our course as well!). Therefore, we decided to create a version of the KeY system that combines forward symbolic execution with axiomatic reasoning on a while-language with automatic first-order simplification [Hähnle and Bubel, 2008]. Incidentally, when we were looking for examples for our course to be done with KeY Hoare, we found that a large number of Hoare logic derivations published in lectures notes and text books are slightly wrong: forgotten preconditions, too weak invariants, etc. This is practically inevitable when formal verification is attempted by hand and demonstrates that tool support is absolutely essential in a formal methods course. The KeY Hoare tool is discussed in this book in Chapter 17. The aforementioned course is currently still taught at Chalmers University[3] (with KeY having been replaced by Dafny [Leino, 2010]).

Another course worth mentioning is called *Software Engineering using Formal Methods*. It was created in 2004 by the author of this chapter, initially conceived as a first year computer science M.Sc. course and it is still being taught[4] in this fashion at Chalmers University and elsewhere. The course introduces model checking with PROMELA and SPIN [Holzmann, 2003] in its first part and functional verification of Java with KeY in the second. It is a mix between theoretical foundations and hands-on experimentation. By 2012 KeY was considered to be stable and usable enough (SPIN had reached that state over a decade earlier) to design a somewhat stripped down version of the course for 2nd year B.Sc. students. Since Fall 2012 that course is compulsory for Computer Science majors at TU Darmstadt and taught annually to 250–300 students.[5] The response of the students is on the whole encouraging and evaluation results are in the upper segment of compulsory courses. Other versions of the course, based on material supplied by the KeY team, are or were taught at CMU, Polytechnic University of Madrid, University of Rennes, University of Iowa, RISC Linz, to name a few.

What this shows is that formal verification can be successfully taught even in large classes and to students without a strong background or interest in mathematics. The user interface of the KeY system, in particular its GUI, plays a major role here. Another important issue is ease of installation: KeY installs via Java webstart technology on most computers with only one click.[6] In this context it cannot be overestimated how important usability and stability are for user acceptance. Students later become professionals and are the prospective users of our technology. Their feedback has been taken very seriously in the development of KeY. In fact, the results

---

[3] www.cse.chalmers.se/edu/course/TDA567

[4] www.cse.chalmers.se/edu/course/TDA293

[5] www.se.informatik.tu-darmstadt.de/teaching/courses/formale-methoden-im-softwareentwurf

[6] If you want to try it out right now, jump to Section 15.2 to see how.

of systematic usability studies done with KeY [Beckert and Grebing, 2012, Hentschel et al., 2016] are reflected in the design of its user interface. KeY's user interface and its various features are explained in Chapter 15. A systematic tutorial to get you started with the verification of actual Java programs is found in Chapter 16.

1.3.4 Avoid the Need for Fully Specified Formal Semantics

There is one application scenario for deductive verification technology that not only dispenses with the need for formal specification, but does not even require any prior knowledge in formal methods and does not necessarily rest on a full axiomatization of all target language constructs. First experiments with a prototype of an interactive debugger based on KeY's symbolic execution engine [Hähnle et al., 2010] resulted in a dedicated, mature tool called *Symbolic Execution Debugger* (SED) [Hentschel et al., 2014a]. It offers all the functionality of a typical Java debugger, but in addition, it can explore all symbolic execution paths through a given program, which it visualizes as a tree. Full exploration of all paths is possible, because the SED can handle loop invariants and method contracts. Also the symbolic environment and heap at each execution step can be visualized. The SED is realized as an Eclipse extension and is as easy to use as the standard Java debugger of Eclipse. The SED is explained in Chapter 11.

In the future we plan to support concurrent Java programs in the SED and programs with floating-point types, both of which are not axiomatized in KeY's program logic. But for the purpose of debugging and visualization this is also not really necessary. To debug concurrent programs it is already useful to concentrate on one thread—this is what debuggers normally do. The underlying verification machinery of KeY can give additional hints that go beyond the capabilities of standard debuggers, for example, which values might have been changed by other threads.

Floating-point data can be represented simply by symbolic terms that contain the operations performed to obtain a current value. As floating-point operations raise no `ArithmeticExcpetion` besides "divide by zero," they do not create control flow that is not represented already in a symbolic execution tree over integer types. Of course, a symbolic execution tree over "uninterpreted" floating-point terms may contain infeasible paths. But this could be detected by test case generation.

1.3.5 Where Are We Now?

Where does deductive verification with KeY stand at this moment? What about our original goal to bring formal verification into the realm of mainstream software development? I believe we are on the right track: with the Symbolic Execution Debugger, with the Eclipse integration of KeY, and with mostly automatic tools such as test case generation. Other research groups are also working towards the

general goal, even though by different means: several test case generation tools based on symbolic execution are commercially used; a static analysis tool using deduction is part of Microsoft's developer tools, a termination analyzer will soon follow; the Spec# programming system includes a verifier and bug finding tool for the C# programming language.

As explained in Section 1.3.1 above, deductive verification technology is also a base technology for security analysis, sound compilation and program transformation, resource analysis and, in the future, to regression test generation, fault propagation as well as possibly other scenarios. We predict that in ten years from now, deductive verification will be the underlying technology, largely invisible to the end-user, in a wide range of software productivity products used in different phases of the development chain.

But what about functional verification—the supposed gold standard? Routine functional specification and formal verification is possible—for restricted languages such as Java Card or SPARK that can be fully formalized, or for languages such as ABS that have been developed with verifiability in mind. Partial functional verification of medium-sized, real C programs is possible [Alkassar et al., 2010, Klein et al., 2010]—with huge effort and with limited reusability. Most importantly, functional verification of complex Java library methods is possible, provided that they fall into Java fragments covered by verification tools: with the help of KeY we found a subtle bug in the default sorting method of Java (as well as Python, Haskell, and several other programming languages and frameworks) and we showed that our fix actually eliminates the bug [De Gouw et al., 2015]. The proof required over two million steps and an effort of several person weeks.

A central limitation of all approaches to functional verification so far is their brittleness in the presence of evolution of the verification target: any change to the program under verification may necessitate to redo a large amount of the verification effort already spent. Even though first approaches to alleviate this problem have been presented [Bubel et al., 2014b], it is far from being solved.

For the reasons spelled out above, we expect that functional verification of executable source code remains a niche application of deductive verification in the foreseeable future, suitable for safety-critical software, where restricted programming languages are acceptable and changes to the verification target are carefully controlled. Another area where formal verification will become widely used are model-centric approaches, where verification of a software model is combined with code generation. The latter is particularly interesting for industry, where model-driven development approaches have been well received.

But then, who knows? Probably, the world (of verification) will have changed again, when we prepare the third edition of the KeY book in 2025 or so. Already now we see that the view on deductive verification of software became much more differentiated than it was at the start of the KeY project. The question is no longer *whether* deductive verification is useful, but *how* to make best use of it. These are exciting times to work with Formal Methods in Software Engineering!

1.4 The Architecture of KeY

We emphasize that KeY is an integrated, standalone system that doesn't need any external components to function. This is in contrast to, for example, verification condition generators such as Dafny [Leino, 2010] that are part of a tool chain (in Dafny's case Boogie [Barnett et al., 2006] and various SMT solvers). Nevertheless, KeY can be advantageously integrated with other deduction and software productivity tools, as we shall see.

1.4.1 Prover Core

How does the architectureof KeY support the extended application scenarios of deductive verification sketched above? Let us start at the core. Here we find a pretty standard *sequent* or *Gentzen* calculus: a set of schematic rules that manipulate structured implications, called *sequents*, of the form

$$\varphi_1,\ldots,\varphi_m \Longrightarrow \psi_1,\ldots,\psi_n$$

where $0 \le m$, $0 \le n$ and the φ_i, ψ_j are formulas. The semantics of sequents is that the *meaning formula* of the sequent, i.e., the universal closure of the formula $\bigwedge_{i=1}^{m} \varphi_i \to \bigvee_{j=1}^{n} \varphi_j$, is valid in all models. This setup is a standard choice for many interactive theorem provers [Balser et al., 2000, Nipkow et al., 2002].

A specific choice of KeY, however, is that formulas φ_i are from a *program logic* that includes correctness modalities over actual Java source code fragments. Before we come to that, let us note some important design decisions in KeY's logic. First of all, the program logic of KeY is an extension of typed first-order classical logic. All KeY formulas that fall into its first-order fragment are evaluated relative to classical model semantics and all sequent rules over first-order formulas are classically sound and complete. In particular, we assume that all structural sequent rules are valid so that associativity, order, and multiplicity of formulas occurring in sequents are irrelevant. The first-order fragment of KeY's logic and its calculus is explained in detail in Chapter 2. Figure 2.1, for example, lists the rules for classical connectives and quantifiers.

A central design issue for any interactive verifier is how the rules of its calculus are composed into proofs. Here we come to a second peculiarity of KeY that sets it apart from most other systems: the rule set is parametric to the system and can be arbitrarily chosen during initialization. Many verifiers let the system user build new rules from existing ones with the help of meta-rules, so-called tactics [Gordon et al., 1979] with the restriction that only rules whose validity can be proven inside the system are available [Nipkow et al., 2002, Paulin-Mohring, 2012]. This requires some form of higher-order logic. The main advantage is that soundness of the rule base can be reduced to a small set of trusted axioms, for example, those of set theory in the case of Isabelle/HOL [Nipkow et al., 2002].

In KeY we decided to follow a different path and trade off a small trusted foundation for more flexibility and a less steep learning curve for users of the system. Schematic sequent rules in KeY are specified as so-called *taclets* [Beckert et al., 2004]. They contain the declarative, logical content of the rule schemata, but also *pragmatic* information: in which context and when a rule should be applied by an automated reasoning strategy and how it is to be presented to the user. Taclets constitute in essence a tiny domain-specific language for typed, first-order schematic rules. This is explained in detail in Chapter 4. At the core of the KeY system is an efficient interpreter that applies taclets to goal sequents and thereby constructs proof trees. It is called *KeY Prover* in Figure 1.1.

The language of taclets and the KeY Prover are intentionally restricted for efficiency reasons: for example, taclets always have exactly one main formula of a sequent in focus that can be manipulated and automated proof search does not implement backtracking (though proofs can be pruned interactively). As a consequence, one cannot describe most calculi for modal and substructural logics [D'Agostino et al., 1999] with taclets; one could represent a calculus for intuitionistic logic, but not automated proof search in it, etc. In other words, taclets are optimized for their main use case: automated proof search in typed first-order logic and logic-based symbolic execution in JavaDL.

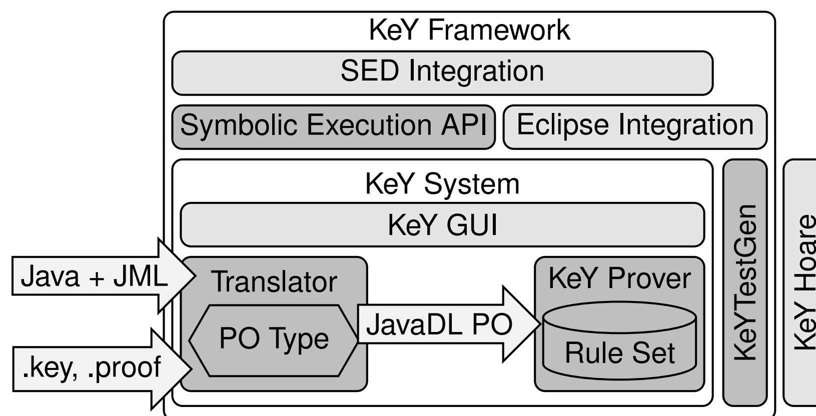

Figure 1.1 Architecture of the KeY tool set

Taclets provide the kind of flexibility we need for the various application scenarios in KeY: there are different rule sets tailored to functional verification, to information flow analysis, etc. Moreover, taclets dispense with the need to support higher-order quantification in the logic. This makes interaction with the prover easier, both for humans and with other programs: for example, as first-order logic is taught in introductory courses on discrete math, it is possible to expose second-year B.Sc. students to KeY in a compulsory course at Technische Universität Darmstadt (see Section 1.3.3). But also the proximity of modeling languages such as JML and of

the language of SMT solvers [Barrett et al., 2010] to typed first-order logic make it simple to import and export formulas from KeY's program logic. On the one hand, this makes it possible to have JML-annotated Java as an input language of KeY, on the other hand, using SMT solvers as a backend increases the degree of automation.

An important question is how to ensure soundness of the more than 1,500 taclets loaded in a standard configuration of KeY. Many of them are first-order rewrite rules and have been proven to be sound in KeY itself relative to the few rules given in Chapter 2. Rules that manipulate programs, however, can in general not be validated in KeY's first-order logic. Instead, soundness of a large part of the rules dealing with programs has been proven against a formal semantics of Java external to KeY [Ahrendt et al., 2005]. That paper, as well as Section 3.5.3, discusses the pros and cons of KeY's taclet approach versus the foundational approach implemented in higher-order logic proof assistants.

1.4.2 Reasoning About Programs

The program logic of KeY contains "correctness formulas" of the form $[p]\varphi$, where p is an executable fragment of a Java program and φ a formula that may in turn contain correctness formulas. This means that Java programs occur directly inside sequents and are neither encoded nor abstracted. Informally, the meaning of the formula above is that when started in an arbitrary state, if the program p terminates, then in its final state φ holds. The formula $[p]\varphi$ relates the initial and the final state of the program p, i.e., its big step semantics $[\![p]\!]$. Therefore, the $[\cdot]\cdot$ operator can be seen as an (infinite) family of modal connectives, indexed by p.

Program formulas are closed under propositional connectives and first-order quantification, therefore, it is directly possible to express a Hoare triple of the form $\{\theta\}p\{\varphi\}$ in KeY as $\theta \rightarrow [p]\varphi$. The resulting logic is known as *dynamic logic* and due to Pratt [1977]. Dynamic logic is more expressive than Hoare logic, because it allows one to characterize, for example, program equivalence. A deepened discussion of dynamic logic is contained in [Harel et al., 2000] and Chapter 3. As the programs occurring in our correctness formulas are Java programs, the logic used by KeY is called Java Dynamic Logic, JavaDL for short.

It is possible to design proof rules for JavaDL that analyze formulas of the form $[p;\omega]\varphi$, where p is a single Java statement and ω the remaining (possibly empty) program. For example, p might be a simple assignment of the form x=e, where x is an `int` variable and e a simple `int` expression. Chapter 3 discusses rules that reduce such a program to a statement about the remaining program $[\omega]\varphi$ plus first-order verification conditions. Other rules decompose complex Java statements into simple ones. The rules for program formulas in KeY are designed in such a way that they constitute a symbolic execution engine for Java. Together with an induction principle (in KeY: loop invariants), symbolic execution becomes a complete verification method [Burstall, 1974, Heisel et al., 1987]: Any valid program formula, for example, $\texttt{n} \geq \texttt{i} \rightarrow [\texttt{while (i<n) \{i++\};}]\texttt{i} \doteq \texttt{n}$ can be syntactically reduced

to a finite set of valid first-order formulas with arithmetic. In contrast to model checking, it is neither necessary to abstract the target program, nor can spurious counter examples occur. The price is, of course, that Java programs must be annotated with suitable specifications, including loop invariants. In addition, some quantifier instantiations might not be found automatically, but must be supplied by the user. This requires a certain amount of expertise, at least if KeY is used for functional verification.

While (logic-based) symbolic execution plus invariant reasoning is, in principle, sufficient to formally verify programs, it is not feasible to verify anything but toy programs without a modularization principle, because the number of branches in a symbolic execution tree grows exponentially with the number of decision conditions in a program. For an imperative, object-oriented language such as Java the most common approach to decompose its verification problem into chunks of manageable size is to provide for each method implementation a declarative specification of its behavior in the form of a *contract*. The idea is that a method call is replaced by the contract which the implementer of the method promises to honor. Thus the caller of a method is the contract's client and the callee is its supplier. This idea goes back to Meyer [1992] who propagated it as *design-by-contract* and implemented a runtime assertion checker in the Eiffel language, but did not use contracts for the purpose of verification. Contracts became also part of the OCL [Warmer and Kleppe, 1999] and in this form were implemented in KeY [Ahrendt et al., 2000].

JML introduced contracts systematically to the Java language; a method contract consists of three parts: a *precondition* specifies when the callee considers the contract to be applicable; a *postcondition* specifies what the callee promises to guarantee in the final state after it returns; finally, an *assignable* clause records the program locations that might have been changed during the execution. Many of the examples in the JML specification and tutorials [Leavens et al., 2013] are geared towards the use of JML in runtime verification. For this reason we found it useful to include Chapter 7 in this book that explains in detail how JML can be used to formally specify functional correctness of Java programs. It can be read with benefit even if one is not interested in KeY and simply wants to learn about formal specification of object-oriented programs.

While contracts provide a natural and effective way to reason about a program by looking at one method at a time, there are two serious challenges in practice: the first is technical and is related to the question of how to specify succinctly the state change effected by a method execution. For example, assignable clauses in practice are not static, but depend on the symbolic heap at call time. There are several technical solutions to this problem, including ownership types [Clarke et al., 1998] and dynamic frames [Kassios, 2006]. The approach of KeY is a variation of the latter and discussed in Chapter 9. The second challenge is to come up with suitable contracts in the first place. As pointed out above this is, at least partially, still an open research issue.

1.4.3 Proof Obligations

A minimal input file of a verification task for KeY might look as simple as this:

KeY

```
\problem{
  \[{ ... a Java program ... }\] φ
}
```

KeY

The system will offer the user to prove validity of the formula given as the problem specification, i.e., partial correctness of the given Java program with respect to the postcondition φ (the square brackets are rendered in ASCII as `\[`, `\]`). How this is done is explained in detail in Chapter 15. It is possible to load such a file with the extension `.key` directly into the prover (see Figure 1.1). In most cases, however, the JavaDL formula to be proven is the result of a translation and a selection of a specific proof obligation. Consider the following snippet from a `.java` file with JML annotations:

Java + JML

```
class C {
  /*@ invariant ι; @*/
  ...
  /*@
    @ requires θ;
    @ ensures φ;
    @*/
  void m(Object o) { ... method body ... }
  ...
}
```

Java + JML

At first glance, this looks similar to the `.key` file above. But there might be many methods declared in it as well as class invariants. One of them has to be selected. And then what to prove? That a selected method satisfies its contract? That the contract is well-defined? But it is also possible to prove *nonfunctional* properties about a given program (see Section 1.3.1), such as secure information flow (see Chapter 13) and correctness of program transformations (see Chapter 14).

So for each given `.java` file there are a plethora of different proof obligations (PO) one might want to look at. It is necessary to first select one of them and then to translate it into JavaDL. This translation is far from trivial: the Java name space must be flattened into a first-order signature, default assumptions of JML (for example, `o!=null` above) must be ensured, well-formedness of the heap and default values must be assumed. Implicit declarations in Java, such as default constructors or "`extends Object`" must be made explicit, etc. The translation of JML (or, rather, the KeY-specific extension of JML) to JavaDL and the generation of various proof obligations is a fully automatic process and explained in Chapter 8.

1.4.4 The Frontends

When you download, install, and start up the KeY system you will see its graphical user interface (GUI), see Figure 1.2. This is the standard frontend of the KeY system. The KeY GUI is a stand-alone Java application. Its usage scenario is to perform functional verification of JML-annotated Java programs. Upon loading a `.java` file the proof obligation selector is launched and selected POs are automatically translated into JavaDL.

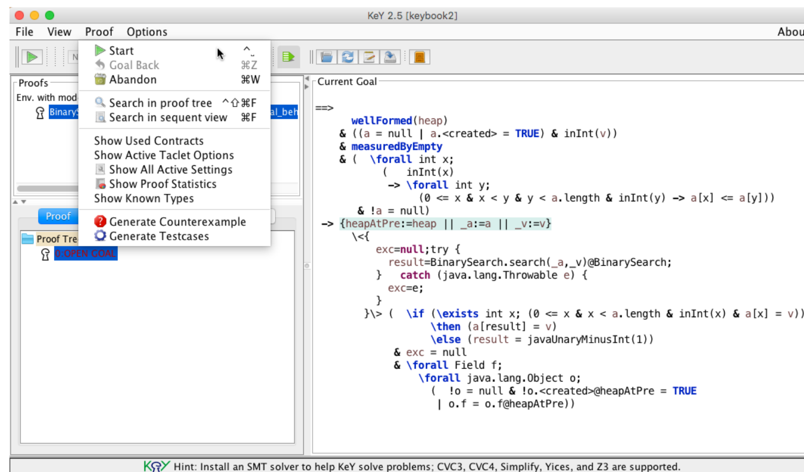

Figure 1.2 GUI of the KeY system with a loaded proof obligation

The test case generation tool, discussed in Chapter 12, is integrated into the KeY GUI (and into the Eclipse GUI, see below), but test case generation typically requires much less user interaction than functional verification. On the other hand, the generated test cases need to be connected to JUNIT (see junit.org) or other unit test frameworks to be executed and managed.

For some of the application scenarios of deductive verification discussed in Section 1.2 the KeY GUI is not suitable. The various tools based on the KeY system also address a variety of different user communities. Therefore, they are packaged separately from the KeY system and provide alternative frontends.

The teaching tool KeY Hoare is a stripped down version of the KeY system that lets students perform formal verification exercises on a simple while-language and provides a Hoare logic-like view of JavaDL proofs [Hähnle and Bubel, 2008]. KeY Hoare has an interface that is similar to the KeY GUI, but is much simplified.

There are two KeY frontends in the shape of extensions of the popular software development environment Eclipse: the Symbolic Execution Debugger, mentioned above in Section 1.3.4 and discussed more fully in Chapter 11, is fully immersed into Eclipse. Generation of POs and proving them happens in the background, the

KeY system is completely invisible to the user. This is made possible by a dedicated symbolic execution API which exports the capabilities of the KeY system (described in the previous subsections) to external programs without having to go via the KeY GUI. Finally, the Eclipse integration of the KeY system attempts to integrate formal verification with KeY into the standard development workflow. Currently, there are two Eclipse extensions: the KeY 4 Eclipse Starter that connects existing Java projects with the KeY system so it can be invoked from within Eclipse to verify methods. The second extension is called KeY Resources and extends a standard Eclipse Java project into a KeY project that permits to run proofs in the background and to manage open proof obligations. The Eclipse integration is discussed in Chapter 15.

1.5 The Next Ten Years

A decade has passed since the first lines of the first edition of the KeY book [Beckert et al., 2007] were written. In this introduction we tried to summarize what has happened since then and where we stand at the moment. Now we take a glimpse at the future and discuss what appear to be the most likely developments. The third edition of the KeY book will tell whether we are on target.

1.5.1 Modular Architecture

The architecture of the KeY framework laid out in the previous section suggests that the KeY system can be reused and instantiated for its various incarnations and usage scenarios. To tell the truth, this is not quite the case. In reality, there are a number of *profiles* of the KeY system: for functional verification, for Hoare logic, for information flow, etc. They all started from the same basis, but live in different development branches. The problem with this is obvious: it is difficult to propagate bug fixes and other improvements. The system KeYmaera [Platzer and Quesel, 2008], now developed by André Platzer at CMU Pittsburgh, branched off from KeY at around 2007 and soon the differences between the systems became too large to attempt a merge. This is regrettable, because a lot of improvements were made for each system over the years that would have benefited both of them, but are now too expensive to transfer.

To avoid this situation in the future, a major refactoring of the KeY system has been initiated. There will be an extensible common core, into which all major development branches eventually will be remerged.

A closely related architectural issue concerns the target language. KeY has been developed to verify Java programs, but currently supports at least also the modeling language ABS [Bubel et al., 2014a] and the while-language used in KeY Hoare. A version of KeY for a subset of C was once available [Mürk et al., 2007], but was abandoned: the lack of multi-language support in KeY made it impossible to

attempt a merge and maintaining a separate branch was too expensive. Finally, the sound compilation approach detailed in Chapter 14 requires at least to support Java bytecode. All this suggests that KeY should strive to enable support for multiple target languages. This seems possible, because the language-specific frontends (parsers, pretty-printers) are largely separate and the internal data structures dealing with ASTs are fairly general. Importantly, the taclet concept and the prover core can be made generic.

We expect that the next major release of KeY will have a unified architecture for different extensions and will offer multi-target language support.

1.5.2 Relational Properties Are Everywhere

Two of the various kinds of proof obligations currently supported by KeY are relational in their nature (see Section 1.3.1): information flow and sound program transformation. We argued above that relational properties are a highly interesting scenario for deductive verification, because specifications are uniform and coupling invariants are much easier to derive than functional invariants. We predict that relational verification problems will become a hotspot for research in formal verification in the coming years. Not only are they feasible and practically relevant, but after a closer look they are very widespread, even ubiquitous in software development. To name just three examples:

1. Fault injection is an import testing strategy against external faults for safety-critical systems. Using deductive verification, it can be generalized to a symbolic fault analysis [Larsson and Hähnle, 2007]. In analogy to verification of information flow properties one can then prove properties about the *fault propagation* for a given program.
2. A growing problem for software that must work in many different environments and configurations is to detect and to exclude unwanted feature interactions [Apel et al., 2010]. To compare the behavior of two versions of a program with different features again is a relational problem.
3. Regression verification is a problem of huge practical interest, particularly in modern software development processes, such as continuous deployment. As pointed out above in Section 1.3.1, it is much easier to verify the preservation of behavior among two closely related programs than to establish functional correctness. Therefore, automatic regression verification is an interesting and feasible goal of deductive verification [Felsing et al., 2014].

And there is another important reason why relational verification problems are interesting: they provide a natural bridge to test-based approaches. For example, from a failed attempt at verifying secure information flow, it is possible to extract a candidate for an attack on privacy, an *exploit* [Do et al., 2015]; from a failed attempt to show behavioral equivalence of two versions of a program one can generate a regression test, and so on.

1.5.3 Learning from Others

Recently there has been substantial progress in the field of automata learning related to the problem of learning behavioral structures from sets of computation traces [Isberner et al., 2014]. It is possible to learn automata with at least a limited notion of data types as part of their state. Formal verification tools at the moment almost completely ignore the potential of machine learning, even though, as some first work demonstrates [Howar et al., 2013], it should be possible to alleviate the specification authoring problem (Section 1.2). Vice versa, learning algorithms for state machines typically suffer from slow convergence and from scaling issues. Why not try to import successful techniques from formal verification, such as contracts or symbolic values, to machine learning? Clearly, here lie vast research opportunities.

1.5.4 Code Reviews

Code inspections and code reviews [Fagan, 1976] are popular and important software quality assurance measures [Sommerville, 2015]. One of their downsides is that they are very time-intensive. Tools such as the SED (see Chapter 11) can and should be further developed into Code Review Assistants that efficiently guide through all possible behaviors, animate execution paths and data structures and can find potential problems or code smells not merely based on metrics and syntactic analyses, but based on deductive verification technology. A recent experimental user case study [Hentschel et al., 2016] showed that this is a promising path.

1.5.5 Integration

When writing an overview article on deductive verification [Beckert and Hähnle, 2014], we realized to which large extent the verification community suffers from a fragmentation of tools. There are well over one hundred verification tools currently available with widely varying scopes, theoretical bases, and usage scenarios. Only a few subcommunities organize competitions or systematic tool comparisons [Klebanov et al., 2011, Beyer, 2015]. In most cases, larger case studies are not publicly available. With the exception of SMT solvers that are integrated via the SMT-LIB standard (smt-lib.org) [Barrett et al., 2010], virtually no generally accepted interface languages or APIs exist. Even programs annotated in JML cannot be readily exchanged, because of slightly different interpretation of the semantics of some JML constructs in different tools and because of different coverage of Java.

Even though it is natural to combine, for instance, symbolic execution with invariant generation and termination analysis, this is exceedingly time consuming in practice. It is important to work on exchange standards that would allow, for example,

to transfer symbolic program states or invariants at a certain point of execution in a semantically sound manner.

Acknowledgments and Disclaimer

I would like to thank Richard Bubel and Martin Hentschel for helping to clarify the KeY architecture discussed in Section 1.4 and for drafting the first version of Figure 1.1. Daniel Grahl, Martin Hentschel, Peter Schmitt, and Shmuel Tyszberowicz all proofread this chapter and gave valuable feedback. More generally, this chapter could not have been written without the continuing efforts of the whole KeY team. Nevertheless, the opinions and judgments expressed here are the author's and do not necessarily reflect those of everyone else involved in KeY.