The KeY System 1.0 (Deduction Component)

Bernhard Beckert, Martin Giese, Reiner Hähnle, Vladimir Klebanov,
Philipp Rümmer, Steffen Schlager, and Peter H. Schmitt

www.key-project.org

Abstract. The KeY system is a development of the ongoing KeY project, whose aim is to integrate formal specification and deductive verification into the industrial software engineering processes. The deductive component of the KeY system is a novel interactive/automated prover for first-order Dynamic Logic for Java. The KeY prover features a user-friendly graphical interface, a backtracking-free free-variable sequent calculus, a simple and powerful theory formalization language called "taclets," solution procedures for linear and non-linear integer arithmetic, external theorem prover integration, and facilities for proof reuse, among other aspects. The system is publicly available.

Introduction. The KeY system is the main software product of the KeY project, a joint effort between the University of Karlsruhe, Chalmers University of Technology in Göteborg, and the University of Koblenz. The KeY system is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. At the core of the system is a deductive verification component, which also can be used as a stand-alone prover. It employs a free-variable sequent calculus for first-order Dynamic Logic for Java. The calculus is proof-confluent, i.e., no backtracking is necessary during proof search.

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

In this paper we concentrate on the description of the KeY prover and the reasoning techniques it employs. The prover consists of ca. 124,000 lines[1] of Java code. The standard rule base consists of 1,725 rules that are written in about 15,000 lines of KeY's "taclet" rule description language. About 1,300 of these formalize the semantics of the Java programming language. The system has been created by 14 implementors since 1999, who spent a total of about

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

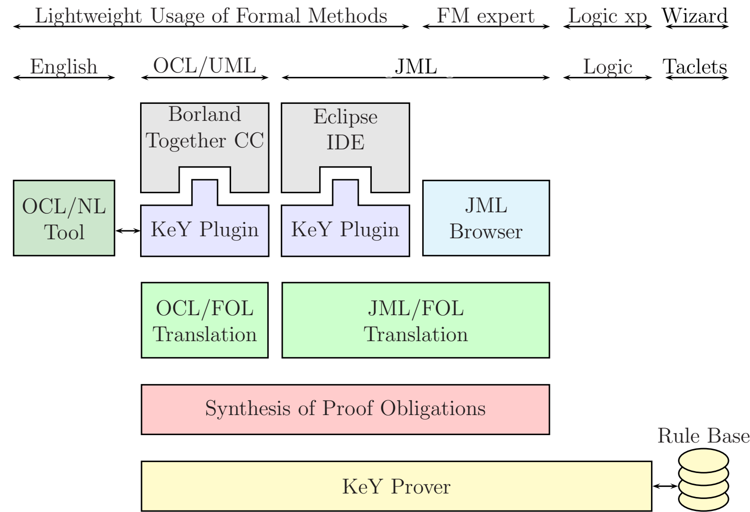

Fig. 1. Architecture and interfaces of the KeY system

30 person years. Recently, version 1.0 of the KeY system has been released in connection with the KeY book [2]. The KeY tool is available under GPL and can be downloaded from `www.key-project.org`.

The KeY Program Verification System. The architecture of the KeY system is shown in Fig. 1. Optional plugins to the popular Eclipse IDE and to the Borland Together CASE tool suite have been developed to lower the entry hurdle for users with no or little training in formal methods. KeY supports several languages for specifying properties of object-oriented models. Many people working with UML or model-driven development have familiarity with the specification language OCL (Object Constraint Language), a part of UML 2.0. Another supported specification language, which enjoys popularity among Java developers, is JML (Java Modeling Language). KeY can also translate OCL expressions to natural language (English and German).

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

The system is not a classical verification condition generator (VCG), but a theorem prover for program logic that combines a variety of automated reasoning techniques. The KeY prover is distinguished from most other deductive verification systems in that symbolic execution of programs, first-order reason-

ing, arithmetic simplification, external decision procedures, and symbolic state simplification are interleaved. For loop- and recursion-free programs, symbolic execution typically is performed in a fully automated manner.

Syntax and Semantics of the KeY Logic. The foundation of the KeY logic is a typed first-order predicate logic with subtyping. This foundation is extended with parameterized modal operators $\langle p \rangle$ and $[p]$, where p can be any sequence of legal Java Card statements. The resulting multi-modal program logic is called Java Card Dynamic Logic or, for short, Java Card DL [2, Chapt. 3].

As is typical for dynamic logic, Java Card DL integrates programs and formulas within a single language. The modal operators refer to the final state of program p and can be placed in front of any formula. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds, while $[p]\phi$ does not demand termination and expresses that *if* p terminates, then ϕ holds in the final state. For example, "when started in a state where `x` is zero, `x++;` terminates in a state where `x` is one" can be expressed as $\mathtt{x} \doteq 0 \mathrel{->} \langle \mathtt{x++;} \rangle (\mathtt{x} \doteq 1)$. The states used to interpret formulas are first-order structures sharing a common universe.

The type system of the KeY logic is designed to match the Java type system but can be used for other purposes as well. The logic includes *type casts* (changing the static type of a term) and *type predicates* (checking the dynamic type of a term) in order to reason about inheritance and polymorphism [2, Chapter 2]. The type hierarchy contains the types such as boolean, the root reference type Object, and the type Null, which is a subtype of all reference types. It contains a set of user-defined types, which are usually used to represent the interfaces and classes of a given Java Card program. Finally, it contains several integer types, including both the range-limited types of Java and the infinite integer type $\mathbb{Z}$.

Beside built-in symbols (such as type-cast functions, equality, and operations on integers), user-defined functions and predicates can be added to the signature. They can be either *rigid* or *non-rigid*. Intuitively, rigid symbols have the same meaning in all program states (e.g., the addition on integers), whereas the meaning of non-rigid symbols may differ from state to state.

Finally, there is another kind of modal operators called *updates*. They can be seen as a language for describing program transitions. There are simple function updates corresponding to assignments in an imperative programming language, which in turn can be composed sequentially and used to form parallel or quantified updates. Updates play a central role in KeY: the verification calculus transforms Java Card programs into updates. KeY contains a powerful and efficient mechanism for simplifying updates and applying them to formulas.

Rule Formalization and Application. The user can easily interleave the automated proof search implemented in KeY and interactive rule application. For interactive rule application, the KeY prover has an easy to use graphical user interface that is built around the idea of direct manipulation (Fig. 2). To apply a rule, the user first selects a *focus of application* by highlighting a (sub-)formula or a (sub-)term in the goal sequent. The prover then offers a choice of rules

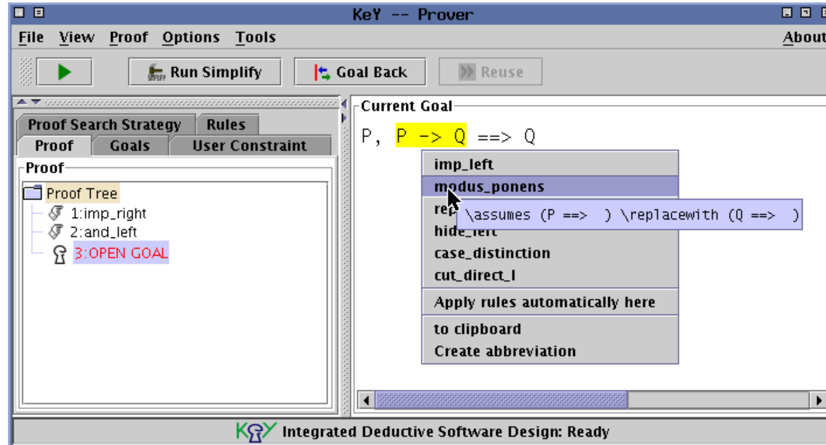

Fig. 2. Screenshot of the KeY prover user interface

applicable at this focus. This choice remains manageable even for very large rule bases. Rule schema variable instantiations are mostly inferred by matching.

Another simple way to apply rules and give instantiations is by drag and drop. If the user drags an equation onto a term the system will try to rewrite the term with the equation. If the user drags a term onto a quantifier the system will try to instantiate the quantifier with this term.

The interaction style is closely related to the way rules are formalized in the KeY prover. There are no hard-coded rules; all rules are defined in the *taclet language* instead. Besides the conventional declarative semantics, taclets have a clear operational semantics, as the following example shows—a "modus ponens" rule in textbook notation (left) and as a taclet (right):

$$\frac{\phi, \psi, \Gamma \vdash \Delta}{\phi, \phi \to \psi, \Gamma \vdash \Delta}$$

```
\find (p -> q ==>)          // implication in antecedent
\assumes (p ==>)            // side condition
\replacewith(q ==>)         // action on found focus
\heuristics(simplify)       // strategy information
```

The `find` clause specifies the potential application focus. The taclet will be offered to the user on selecting a matching focus and if the formula mentioned in the `assumes` clause is present in the sequent. The action clauses `replacewith` and `add` allow modifying (or deleting) the formula in focus, as well as adding additional formulas (not present here). The `heuristics` clause provides priority information to the parameterized automated proof search strategy.

The taclet language is quickly mastered and makes the rule base easy to maintain and extend. Taclets can be proven correct against a set of base taclets. A full account of the taclet language is given in [2].

Confluent Calculus. In order to simplify the proof construction, which is typically partly automated and partly interactive, we have developed and employ a

proof confluent sequent calculus. This means that automated proof search does not require backtracking over rule applications, which is advantageous for analyzing failed proof attempts. The automated search for quantifier instantiations uses rigid free variables (called meta variables) like in a free-variable tableau calculus. Instead of backtracking over meta-variable instantiations, instantiations are postponed to the point where the whole proof can be closed, and an incremental global closure check is used. To minimize the confusion of novice users, meta variables are not visible in normal interactive use, if the user provides all required instantiations. Rule applications requiring particular instantiations (unifications) of meta variables are handled by attaching unification constraints to the resulting formulas [2, Sects. 4.3 and 10.2.2]. Equations are handled by ordered rewriting (currently in an incomplete way, which we have not, however, found to be a limiting factor so far).

The taclet language is designed in such a way that the user can only write rules with local effects on sequents, and the handling of meta variables, skolemization, constraints, etc. is taken care of automatically, to reduce the risk of inadvertently introducing rules that are unsound or damage confluence.

Handling Arithmetics. As the theory of integer arithmetic is omnipresent in program verification, KeY directly provides a number of automatic solution and simplification procedures for different fragments of arithmetic. All procedures are formulated in terms of taclets, which have been verified against a small set of base axioms. The implemented methods target both proving (showing that equations are unsolvable) and construction of counterexamples (finding solutions of equations) for ground integer formulas.

The most basic method is a sequent calculus formulation of integer Gaussian elimination, which is a complete method for solving linear equations. As a prerequisite of the procedure, integer expressions are always fully expanded and sorted. Linear inequalities are handled by Fourier-Motzkin variable elimination, which we combine with systematic case distinctions in order to obtain a complete procedure over the integers.

Reasoning in non-linear integer arithmetic is mainly carried out by heuristic cross-multiplication of inequalities, similar to the approach of the ACL2 prover. In order to reduce expressions as far as possible and handle non-linear equations more efficiently, KeY also computes Gröbner bases over the integers.

The KeY system also features a component for easy integration of external automated theorem provers and (semi-)decision procedures. Proof goals are translated into the standardized input format SMT-LIB and discharged by calling any tool that understands this format, such as Yices or CVC Lite. A similar connector for the theorem prover Simplify is also available. The user benefits from the particular abilities of these tools to decide fragments of arithmetics, heuristically instantiate quantifiers, etc.

Applications. The main application of the KeY prover is to support program verification in the KeY system. Among the major achievements in this field so far are the treatment of the Demoney case study (an electronic purse application

provided by Trusted Logic S.A.) and the verification of a Java implementation of the Schorr-Waite graph marking algorithm. This algorithm, originally developed for garbage collectors, has recently become a popular benchmark for program verification tools. Chapters 14 and 15 of the KeY book [2] are devoted to a detailed description of these case studies. A case study [6] performed within the HIJA project has verified with KeY the lateral module of the flight management system, a part of the on-board control software from Thales Avionics.

Lately we have applied the KeY system also to issues of security analysis [3], and in the area of model-based test case generation [1, 4] where, in particular, the prover is used to compute path conditions and to identify infeasible paths. The flexibility of KeY w.r.t. the used logic and calculus further manifests itself in the fact that the prover has been chosen as a reasoning engine for a variety of other purposes. These include the mechanization of a logic for Abstract State Machines [7] and the implementation of a calculus for simplifying OCL constraints [5].

KeY is also very useful for teaching logic, deduction, and formal methods. Its graphical user interface makes KeY easy to use for students. They can step through proofs with different degrees of automation (using the full verification calculus or just the first-order core rules). The authors have been successfully teaching courses for several years using the KeY system. An overview and course materials are available at `www.key-project.org/teaching`.

References

1. B. Beckert and C. Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In Y. Gurevich, editor, *Proceedings, Testing and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
3. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proc. 2nd Int. Conf. on Security in Pervasive Computing*, LNCS 3450, pages 193–209. Springer, 2005.
4. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In Y. Gurevich, editor, *Proceedings, Testing and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.
5. M. Giese and D. Larsson. Simplifying transformations of OCL constraints. In L. Briand and C. Williams, editors, *Proceedings, Model Driven Engineering Languages and Systems (MoDELS), Montego Bay, Jamaica*, LNCS 3713. Springer, 2005.
6. J. J. Hunt, E. Jenn, S. Leriche, P. Schmitt, I. Tonin, and C. Wonnemann. A case study of specification and verification using JML in an avionics application. In M. Rochard-Foy and A. Wellings, editors, *Proc. of the 4th Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. ACM Press, 2006.
7. S. Nanchen, H. Schmid, P. Schmitt, and R. F. Stärk. The ASMKeY prover. Technical Report 436, Department of Computer Science, ETH Zürich, 2004.