KeY: A Formal Method for Object-Oriented Systems

Wolfgang Ahrendt[1], Bernhard Beckert[2], Reiner Hähnle[1], and Peter H. Schmitt[3]

[1] Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
ahrendt|reiner@chalmers.se
[2] Department of Computer Science
University of Koblenz
beckert@uni-koblenz.de
[3] Department of Theoretical Computer Science
University of Karlsruhe
pschmitt@ira.uka.de

Abstract. This paper gives an overview of the KeY approach and highlights the main features of the KeY system. KeY is an approach (and a system) for the deductive verification of object-oriented software. It aims for integrating design, implementation, formal specification and formal verification as seamlessly as possible. The intention is to provide a platform that allows close collaboration of conventional and formal software development methods.

1 Introduction

The KeY Approach and System. This paper gives an overview of the KeY approach and highlights the main features of the KeY system.

KeY is an approach (and a system) for the deductive verification of object-oriented (OO) software. It aims for integrating design, implementation, formal specification and formal verification as seamlessly as possible. The intention is to provide a platform that allows close collaboration of conventional and formal software development methods.

Recently, version 1.0 of the KeY system has been released in connection with the KeY book [3]. The KeY system is written in Java and runs on all usual architectures. It is available under GPL and can be downloaded from `www.key-project.org`.

Towards an Integration of Formal Methods in Software Engineering. KeY is primarily not a stand-alone tool, but a plugin to (currently two) well-known CASE tools: Borland Together and the Eclipse IDE. Users can develop a whole software project, comprised of specifications as well as implementations, entirely within either of the mentioned CASE tools. The KeY plugin then offers the *extended functionality* to generate proof obligations from selected parts of specifications and verify them with the KeY prover. The KeY verification component, being the core of the KeY system, can also be used as a stand-alone prover.

KeY supports the OMG standard Object Constraint Language (OCL) [26] for specification as well as the Java Modeling Language (JML) [19], which is increasingly used in industrial contexts [5]. Translation of specifications from OCL and JML into logic, as well as the synthesis of a variety of proof obligations, is completely automatic. The same is true, to a large extent, for proof search. In addition, KeY features a syntax-directed editor for OCL that can render OCL expressions in several natural languages while they are being edited. It is even possible to translate OCL expressions automatically into fragments of English and German. This means that KeY provides a common tool and conceptual base for developers and formal methods specialists. The architecture and interfaces of KeY are depicted in Fig. 1.

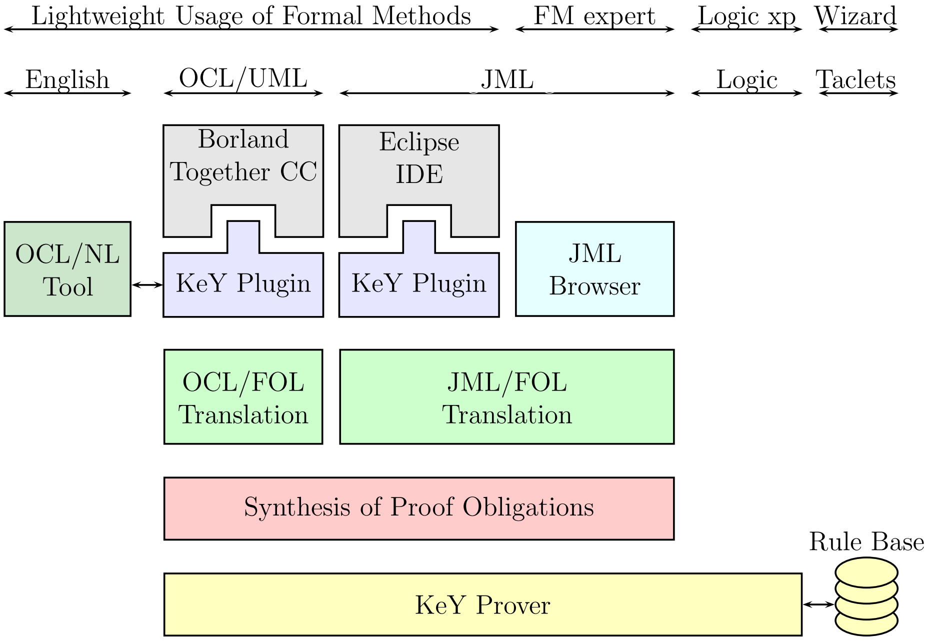


Fig. 1. Architecture and interfaces of the KeY system.

2 Full Coverage of a Real World Language

To ensure acceptance among practitioners it is essential to support an industrially relevant programming language as the verification target. We chose Java Card source code [7] because of its importance for security- and safety-critical applications. We refrained from using a home-spun sublanguage of Java, because it is unrealistic to assume that applications are written in it.

The KeY prover and its calculus [3, Chapt. 3] support the full Java Card 2.2.1 language. This includes all object-oriented features, Java Card's transaction mechanism, the (finite) Java integer types, abrupt termination (local jumps and exceptions) and even a formal specification (both in OCL [18] and JML[4]) of the essential parts of the Java Card API. In addition, some Java features that are not part of Java Card are supported as well: multi-dimensional arrays, Java class initialisation semantics, `char` and `String` types. In short, if you have a sequential Java program without dynamic class loading and floating point types, then it is (in principle) possible to verify it with KeY.

3 Beyond Hoare Logic

KeY is a *deductive verification* system, meaning that its core is a theorem prover, which proves formulae of a suitable logic. Different deductive verification approaches vary in the choice of the used logic. The KeY approach uses *Dynamic Logic* (DL) [14], which (like Hoare Logic [16]) is transparent with respect to the programs that are subject to verification. Programs are neither abstracted away into a less expressive formalism such as finite-state machines nor are they embedded into a general purpose higher-order logic. Instead, the logic and the calculus "work" directly on the source code. This transparency is extremely helpful for proving problems that require a certain amount of human interaction.

DL is a particular kind of *modal logic*. Different parts of a formula are evaluated in different worlds (states), which vary in the interpretation of functions and predicates. DL differs, however, from standard modal logic in that the modalities are "indexed" with pieces of program code, describing how to reach one world (state) from the other. Syntactically, DL extends full first-order logic with two additional (mix-fix) operators: $\langle\,.\,\rangle\,.$ (diamond) and $[\,.\,]\,.$ (box). In both cases, the first argument is a *program*, whereas the second argument is another DL formula. A formula $\langle p\rangle\varphi$ is true in a state s if execution of p terminates normally when started in s and results in a state where φ is true. As for the other operator, a formula $[p]\varphi$ is true in a state s if execution of p, when started in s, does *either* not terminate normally *or* results in a state where φ is true.[5]

DL is closed under all its connectives. For instance, in a DL formula $\langle p\rangle\varphi$, the post-condition φ may by any DL formula again, like in $\langle p\rangle\langle q\rangle\psi$. Also, arbitrary connectives can enclose box or diamond as in the following formula which states equivalence of p and q w.r.t. the "output", the program variable `x`.

$$\forall\, val. \quad (\ \langle p\rangle\ \mathtt{x} = val \quad\leftrightarrow\quad \langle q\rangle\ \mathtt{x} = val\ ) \tag{1}$$

[4] See `http://www.cs.ru.nl/~woj/software/software.html`.

[5] These descriptions have to be generalised when non-deterministic programs are considered, which is not the case here.

A frequent pattern of DL formulae is $\varphi \rightarrow \langle p \rangle \psi$, stating that the program p, when started from a state where φ is true, terminates with ψ being true afterwards. The formula $\varphi \rightarrow [p]\psi$, on the other hand, does not claim termination, and has exactly the same meaning as the Hoare triple $\{\psi\}\ p\ \{\phi\}$.

Unlike most other variants of DL, KeY DL comprises programs from a real language, namely Java Card. Concretely, p is a sequence of (zero, one, or more) Java Card statements. Accordingly, the logic is called Java Card DL. The following is an example of a Java Card DL formula:

$$\texttt{o1.f} < \texttt{o2.f} \rightarrow \langle \texttt{int t=o1.f; o1.f=o2.f; o2.f=t;} \rangle\, \texttt{o2.f} < \texttt{o1.f} \qquad (2)$$

It says that, when started in any state where the integer field `f` of `o1` has a smaller value than `o2.f`, the statement sequence "`int t=o1.f; o1.f=o2.f; o2.f=t;`" terminates, and afterwards `o2.f` is smaller than `o1.f`.

The main advantage of DL over Hoare logic is increased expressiveness: one can express not merely program correctness, but also security properties [8, 20], correctness of program transformations, or the validity of assignable clauses. Also, a pre- or post-condition can contain programs themselves, for instance to express that a linked structure is acyclic. A full account of Java Card DL is found in the KeY book [3, Chapt. 3].

KeY interfaces with OCL as well as JML specifications, by translating them (and the specified Java code) into *proof obligations* in Java Card DL. Following Fig. 1 from the right to the left, we have essentially four scenarios, varying in the origin of the DL proof obligations (POs):

(i) *Hand-crafted* POs, to be loaded from `.key` files.
(ii) *Automatically generated* POs
  (a) from JML-augmented Java source files, using
    – the JML browser of the KeY stand-alone system.
    – Eclipse with the KeY plug-in.
  (b) from OCL-augmented UML diagrams and Java source files, using Borland Together with KeY extensions.

4 Symbolic Execution

The actual verification process in KeY can be viewed as *symbolic execution* of source code. Unbounded loops and recursion are handled by induction over data structures occurring in the verification target. Alternatively, partial correctness of loops can also be shown by a rule that uses invariants. Symbolic execution plus induction as a verification paradigm was originally suggested for informal usage by Burstall [6]. The idea to use dynamic logic as a basis for mechanising symbolic execution was first realized in the Karlsruhe Interactive Verifier (KIV) tool [15]. Symbolic execution is extremely suitable for interactive verification, because proof progress corresponds to program execution, which makes it easy to interpret intermediate stages in a proof and failed proof attempts.

Most program logics (e.g., Hoare Logic, wp-calculus) have in common that the state change effected by a program translates, at some point, into substitutions applied to formulae. In the KeY approach to symbolic execution, the application of substitutions is *delayed* as much as possible; instead of using substitutions, the state change effect of a program is made *syntactically explicit* and accumulated in a construct called *updates*. The role of updates is to record the effects of (a certain path in) the execution of a program. Only when symbolic execution has completed are updates turned into substitutions. We omitted updates so far in the discussion of DL and introduce them by example now.

For instance, when proving formula (2), the prover will after some steps construct the following *sequent* as an intermediate goal (slightly adjusted for presentation):[6]

$$\texttt{o1.f} < \texttt{o2.f} \;\vdash\; \{\texttt{t:=o1.f}\}\langle\texttt{o1.f=o2.f; o2.f=t;}\rangle\ \texttt{o2.f} < \texttt{o1.f} \tag{3}$$

The expression “`t:=o1.f`” is an update, which in this case represents the effect of the symbolically executed initialisation statement. Executing the next Java statement leads to nested (consecutive) updates “`{t:=o1.f}{o1.f:=o2.f}`”, which are merged into one *parallel* update:

$$\texttt{o1.f} < \texttt{o2.f} \;\vdash\; \{\texttt{t:=o1.f || o1.f:=o2.f}\}\langle\texttt{o2.f=t;}\rangle\ \texttt{o2.f} < \texttt{o1.f} \tag{4}$$

Soon after, we have

$$\texttt{o1.f} < \texttt{o2.f} \;\vdash\; \{\texttt{t:=o1.f || o1.f:=o2.f}\}\{\texttt{o2.f:=t}\}\langle\rangle\ \texttt{o2.f} < \texttt{o1.f} \tag{5}$$

This time, the update merging step results in:

$$\texttt{o1.f} < \texttt{o2.f} \;\vdash\; \{\texttt{o1.f:=o2.f || o2.f:=o1.f}\}\langle\rangle\ \texttt{o2.f} < \texttt{o1.f} \tag{6}$$

Two things have happened. First, in a parallelisation step, `t:=o1.f` has been *applied* to `o2.f:=t`. Second, `t:=o1.f` has been *simplified away.* This is justified, because `t` does not appear in the post-condition. Finally, the empty modality $\langle\rangle$ is removed. Only thereafter, the parallel update “meets” the post-condition, and is applied as a substitution, leading to a trivially true sequent.

The second component of symbolic execution, next to updates, is *program transformation.* Java (Card) is a complex language, and the calculus for Java Card DL performs program transformations to resolve all the complex constructs of the language, breaking them down to simple effects that can be moved into updates. For instance, in the case of `try-catch` blocks, symbolic execution proceeds on the “active” statement *inside* the `try` block, until normal or abrupt termination of that block triggers different transformations.

[6] KeY uses a sequent style calculus, see below. For now, it is sufficient to read the sequent arrow $\vdash$ as an implication.

Loops can be dealt with by using invariants in the traditional Hoare style. Alternatively, the calculus allows to combine *unwinding* with *induction*, which we come to in the following section.

5 KeY is Not Merely a VCG

The KeY system is not merely a verification condition generator (VCG), but a theorem prover for program logic that combines a variety of automated reasoning techniques. The KeY prover differs from most other deductive verification systems in that symbolic execution of programs, first-order reasoning, arithmetic simplification, external decision procedures, and symbolic state simplification are *interleaved*. This interleaving takes place on the level of proof strategies, but also on the level of individual rules.

To illustrate the latter point, we discuss a few rules of our *sequent calculus*. Sequents have the form $\phi_1, \ldots, \phi_n \vdash \phi'_1, \ldots, \phi'_m$, with two (possibly empty) lists of formulae connected by the sequent arrow $\vdash$. The meaning of a sequent is that at least one of the $\phi'_1, \ldots, \phi'_m$ follows from the conjunction of the $\phi_1, \ldots, \phi_n$. An example of a rule in the sequent calculus for Java Card DL is the *induction rule* over natural numbers:

$$\frac{\mathit{\Gamma} \ \vdash \ \phi(0),\ \mathit{\Delta} \qquad \mathit{\Gamma} \ \vdash \ \forall n.(\phi(n) \rightarrow \phi(n+1)),\ \mathit{\Delta}}{\mathit{\Gamma} \ \vdash \ \forall n.\phi(n),\ \mathit{\Delta}} \tag{7}$$

The meaning of a sequent calculus rule is that, in order to prove a sequent matching the conclusion of the rule (here "$\mathit{\Gamma} \ \vdash \ \forall n.\phi(n),\ \mathit{\Delta}$"), it is sufficient to prove all premisses (two in this case). As usual, the rules are actually *rule schemas* and appear properly instantiated in the context of a concrete proof.

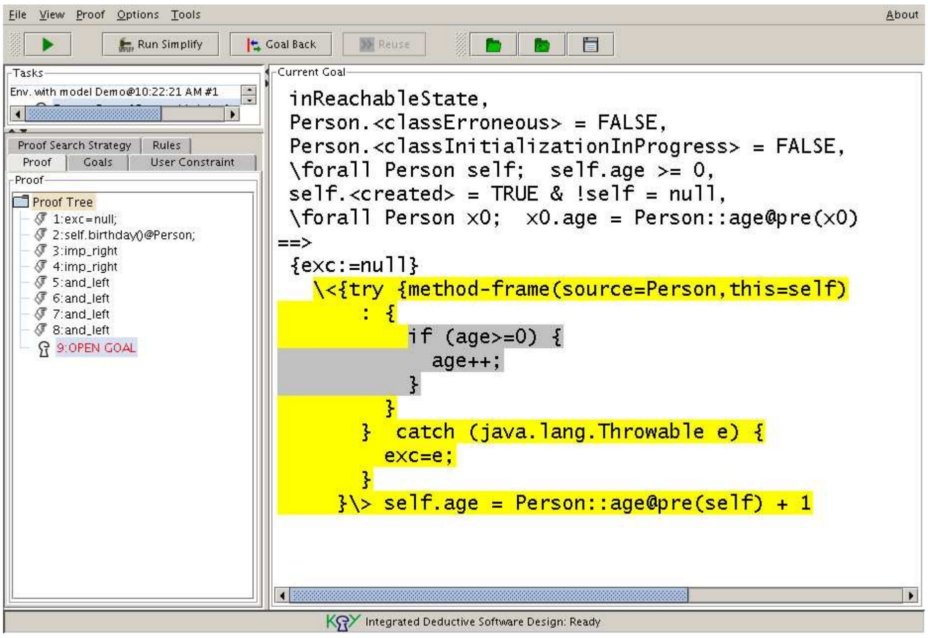
But what has the induction rule (7) to do with Java Card DL, as it looks like a pure first-order rule? The point is that ϕ matches an arbitrary formula in Java Card DL, possibly containing Java Card code (in a modality). And indeed, this rule can be employed for handling loops in ϕ. After applying (7), one proof branch handles the "loop exit" case. In the other branch, the step case is handled, where the loop is unwound once using the *loop unwind* rule:

$$\frac{\mathit{\Gamma} \ \vdash \ \langle \pi \ \texttt{if (}e\texttt{) \{}p\,\texttt{while(}e\texttt{)}\ p\texttt{\}}\ \omega\rangle\phi,\ \mathit{\Delta}}{\mathit{\Gamma} \ \vdash \ \langle \pi \ \texttt{while(}e\texttt{)}\ p\ \omega\rangle\phi,\ \mathit{\Delta}} \tag{8}$$

This is the interplay of symbolic execution and induction which is best described by the title of Burstall's original paper [6]: "Program proving as hand simulation with a little induction."

6 User-Friendly Graphical User Interface

In spite of a high degree of automation (see Sect. 8), in many cases there are significant, non-trivial tasks left for the user. For that purpose, the KeY system provides a user-friendly graphical user interface (GUI). When proving a property which is too involved to be handled fully automatically, certain rule applications need to be performed in an interactive manner, in dialogue with the system. This is the case when either the automated strategies are exhausted, or else when the user deliberately performs a strategic step (like a case distinction) manually, *before* automated strategies are invoked (again).



Proof Tree
- 1:exc=null;
- 2:self.birthday()@Person;
- 3:imp_right
- 4:imp_right
- 5:and_left
- 6:and_left
- 7:and_left
- 8:and_left
- 9:OPEN GOAL

```
 inReachableState,
 Person.<classErroneous> = FALSE,
 Person.<classInitializationInProgress> = FALSE,
 \forall Person self;  self.age >= 0,
 self.<created> = TRUE & !self = null,
 \forall Person x0;  x0.age = Person::age@pre(x0)
==>
 {exc:=null}
   \<{try {method-frame(source=Person,this=self)
        : {
            if (age>=0) {
              age++;
            }
          }
        }  catch (java.lang.Throwable e) {
          exc=e;
        }
      }\> self.age = Person::age@pre(self) + 1
```

Fig. 2. Screenshot of the GUI of the KeY prover.

In the case of human-guided rule application, the user is asked to solve tasks like: *selecting a proof rule* to be applied, *providing instantiations for* the proof rule's *schema variables*, or *providing instantiations for quantified variables* of the logic. The system, and its advanced GUI, are designed to support these steps well. For instance, the selection of the right rule, out of over 1500(!), is greatly simplified by allowing the user to highlight any subexpression of the proof goal simply by positioning the mouse. In the screenshot of the GUI of the KeY prover displayed in Fig. 2, a `try-catch` statement is highlighted. The first active statement in it, the `if` statement, appears in grey.

A dynamic context menu will offer only those few rules that apply to this expression, in this case, the rule for a statement within a `try-catch` block that is not a `throw`. Furthermore, the menus provide tooltips for each rule. When it comes to interactive variable instantiation, *drag-and-drop* mechanisms greatly simplify and speed-up the usage of the instantiation dialogues, and in some cases even allow to omit explicit rule selection. For example, if the user drags an equation onto a term, the system will try to rewrite the term with the equation. And if the user drags a term onto a quantifier, the system will try to instantiate the quantifier with this term.

Other supported forms of interaction in the context of proof construction are the inspection of proof trees, the pruning of proof branches, stepwise backtracking, and the triggering of proof reuse.

7 A Simple High-Level Rule Language

The implementation of the sequent proof rules in the KeY prover is closely related to the pragmatics of interaction within the GUI as described in the previous section. The rules are written in a high-level language, called the "taclet language." Each rule is represented as one *taclet*. Besides the conventional declarative semantics, taclets have an operational semantics that defines their pragmatics in automatic and interactive proof search. The following example shows a "modus ponens" rule in textbook notation (left) and as a taclet (right):

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

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

This example taclet consists of four clauses. The arrow "==>", appearing in three of them, is the KeY system's representation of the sequent arrow $\vdash$. Within taclets, "==>" is used to indicate whether a matching formula appears on the left- or right-hand side of the sequent.

The `find` clause specifies the potential application focus. The taclet will be offered to the user when selecting a matching focus and if a formula mentioned in the `assumes` clause is present in the sequent. The action clauses `replacewith` and `add` (not present in this example) allow modifying (or replacing) the formula in focus, as well as adding additional formulae. The `heuristics` clause records information for the parameterised automated proof search strategy.

Taclets are not only used to represent calculus rules, but also lemmata. The latter can be proven correct against the provided taclets [4]. The taclet language is quickly mastered and makes the rule base easy to maintain and extend. A full account of the taclet language is given in the KeY book [3, Chapt. 4].

8 Automated Proof Search

For automated proof search, a number of predefined strategies are available in KeY using different rule sets that are, for example, optimised to symbolically executing loop-free programs or proving pure first-order formulae.

In order to better interleave interactive and automated proof construction, KeY uses a proof confluent sequent calculus, which means that automated proof search does not require backtracking over rule applications. The taclet language and application mechanism are designed in such a way that the user can write only rules with local effects on sequents, and the handling of meta variables, skolemisation, constraints, etc. is taken care of automatically, to reduce the risk of inadvertently introducing rules which damage confluence.

The automated search for quantifier instantiations uses meta variables[7] that are place-holders for terms. 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. Rule applications requiring particular instantiations (unifications) of meta variables are handled by attaching unification constraints to the resulting formulas [11].

There is a back end to SMT-LIB syntax[8] for proving near-propositional proof goals with external decision procedures.

9 Customisable Verification

The KeY system offers to customise the rule set used for verification. For instance, the user can choose between different semantics of the primitive Java integer types `byte`, `short`, `int`, `long`, and `char`. Options are: the mathematical integers (easy to verify, but not a faithful model of Java and, hence, unsound), mathematical integers with overflow check (sound, reasonably easy to verify, but unable to verify programs that depend on Java's finite ring semantics), and a faithful semantics of Java integers (sound, complete, but difficult to verify). KeY1.0 comes with the mathematical integer semantics chosen as default option, to optimise usability for beginners. However, for a sound treatment of integers, the user should switch to either of the other semantics. Alternatively, one can employ the *proof reuse* feature of KeY, to first construct a proof using the mathematical integer option, and then *replay* the proof with the mathematical overflow semantics selected.

Other examples where one can customise the degree of faithfulness, versus simplicity, are object creation, and null pointer treatment.

10 A Broader Perspective on Verification

One of the most important insights we gained from our work is the realisation that verification technology with symbolic execution can be seen as the base technology of a whole range of applications in software analysis, many of which are more automatic than full verification. In the future we will develop the KeY system into a general software analysis platform where formal verification is only one of many analysis techniques.

[7] This kind of variables are known in the tableau theorem proving community under the name of "free variables" [10].

[8] See `http://combination.cs.uiowa.edu/smtlib/`

For example, in the area of model-based test case generation [2, 9] the prover is used to compute path conditions and to identify infeasible paths. Fully automatic white-box unit test generation for Java Card is possible based on approximative attempts at formal verification of the implementation under test [9]. White-box testing can also be done by combining deduction-based specification extraction and black-box testing, i.e., one generates specifications for given programs and then uses these specifications as input for black-box testing tools [2].

Another usage of verification is in security analysis [8], where the absence and presence of secure information flow including information declassification is shown. Since many security analyses are implemented on the basis of type systems [23] it is promising to try to combine the advantages of type-based and deduction-based methods. In [13] it is shown that dynamic logic can serve as a common framework where such a combination can be realized.

Most of the time, verification attempts are *not* successful, because the specification or the implementation contains bugs. In this case, it is extremely valuable for the user to obtain information from failed proof attempts without having to wade through large proof trees. Generating counter examples for failed proofs, so-called "disproving" of programs, is only started to being explored [22].

It is also possible to cast symbolic program execution to the user interface and the functionality offered by a symbolic source code debugger. One can then set breakpoints, watches, and inspect the intermediate program state. But in contrast to a conventional debugger, such a truly symbolic debugger is based on a symbolic execution tree and can represent not only one program run, but *all* possible program runs [1]. We expect interesting synergies on both sides from combining verification with debugging.

11 Applications

Among the major achievements in program verification using the KeY system are the treatment of the Demoney case study, an electronic purse application provided by Trusted Logic S.A. [3, Chapt. 14] and the verification of a Java implementation of the Schorr-Waite graph marking algorithm [3, Chapt. 15]. This algorithm, originally developed for garbage collectors, has recently become a popular benchmark for program verification tools. As far as we are aware, the KeY study provides the first verification of an executable Java implementation. A case study [17] performed within the HIJA project[9] included formal verification of the lateral module of a flight management system being part of the on-board control software from Thales Avionics. Recently, for the first time an implementation of the Mondex banking card case study [24] was verified with the KeY prover [25].

The flexibility of KeY w.r.t. the used logic and calculus manifests itself in the fact that the prover has been chosen as a reasoning engine for a variety of other logics. These include the mechanisation of a logic for Abstract State

[9] See `http://www.hija.info`.

Machines [21] and the implementation of a calculus for simplifying OCL constraints [12]. A version of the KeY prover that supports the C programming language will be released later this year.

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 using 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. Marcus Baum. A verifying debugger. Master's thesis, Department of Computer Science, Institute for Theoretical Computer Science, to appear, 2007.
2. Bernhard Beckert and Christoph Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In Y. Gurevich, editor, *Proceedings, International Conference on Tests and Proofs (TAP), Zürich, Switzerland*, LNCS. Springer, 2007.
3. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer, 2007.
4. Bernhard Beckert and Vladimir Klebanov. Must program verification systems and calculi be verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, 2006.
5. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
6. Rod M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.
7. Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Java Series. Addison-Wesley, June 2000.
8. Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proceedings, 2nd International Conference on Security in Pervasive Computing*, LNCS 3450, pages 193–209. Springer, 2005.
9. Christian Engel and Reiner Hähnle. Generating unit tests from formal proofs. In Y. Gurevich, editor, *Proceedings, International Conference on Tests and Proofs (TAP), Zürich, Switzerland*, LNCS. Springer, 2007.
10. Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, New York, second edition, 1996.
11. Martin Giese. Incremental closure of free variable tableaux. In *Proceedings, InternationalJoint Conference on Automated Reasoning (IJCAR), Siena, Italy*, number 2083 in LNCS, pages 545–560. Springer, 2001.
12. Martin Giese and Daniel Larsson. Simplifying transformations of OCL constraints. In Lionel Briand and Clay Williams, editors, *Proceedings, Model Driven Engineering Languages and Systems (MoDELS), Montego Bay, Jamaica*, LNCS 3713. Springer, 2005.

13. Reiner Hähnle, Jing Pan, Philipp Rümmer, and Dennis Walter. Integration of a security type system into a program logic. In Ugo Montanari and Don Sanella, editors, *Proc. Trustworthy Global Computing, Lucca, Italy*, LNCS. Springer, 2007.
14. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* MIT Press, 2000.
15. Maritta Heisel, Wolfgang Reif, and Werner Stephan. Program verification by symbolic execution and induction. In K. Morik, editor, *Proceedings, 11th German Workshop on Artificial Intelligence*, volume 152 of *Informatik Fachberichte.* Springer, 1987.
16. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.
17. James J. Hunt, Eric Jenn, Stéphane Leriche, Peter Schmitt, Isabel Tonin, and Claus Wonnemann. A case study of specification and verification using JML in an avionics application. In Marc Rochard-Foy and Andy Wellings, editors, *Proceedings, 4th Workshop on Java Technologies for Real-time and Embedded Systems (JTRES).* ACM Press, 2006.
18. Daniel Larsson and Wojciech Mostowski. Specifying Java Card API in OCL. In Peter H. Schmitt, editor, *OCL 2.0 Workshop at UML 2003*, volume 102C of *ENTCS*, pages 3–19. Elsevier, November 2004.
19. Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual. Draft Revision 1.200*, February 2007.
20. Wojciech Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In Maura Cerioli, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE), Edinburgh*, volume 3442 of *LNCS*, pages 357–371. Springer, 2005.
21. S. Nanchen, H. Schmid, P. Schmitt, and R. F. Stärk. The ASMKeY prover. Technical Report 436, Department of Computer Science, ETH Zürich, 2004.
22. Philipp Rümmer and Muhammad Ali Shah. Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In Y. Gurevich, editor, *Proceedings, International Conference on Tests and Proofs (TAP), Zürich, Switzerland*, LNCS. Springer, 2007.
23. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
24. Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.
25. Isabel Tonin. Verifying the Mondex Case Study: the KeY approach. Technical report, Department of Computer Science, Institute for Theoretical Computer Science, April 2007.
26. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA.* Object Technology Series. Addison-Wesley, Reading/MA, August 2003.