

The KeY Platform for Verification and Analysis of Java Programs

Wolfgang Ahrendt¹, Bernhard Beckert², Daniel Bruns^{2(✉)}, Richard Bubel³,
Christoph Gladisch², Sarah Grebing², Reiner Hähnle³, Martin Hentschel³,
Mihai Herda², Vladimir Klebanov², Wojciech Mostowski⁴, Christoph Scheben²,
Peter H. Schmitt², and Mattias Ulbrich²

¹ Chalmers University of Technology, Gothenburg, Sweden

² Karlsruhe Institute of Technology, Karlsruhe, Germany
bruns@kit.edu

³ Technische Universität Darmstadt, Darmstadt, Germany

⁴ University of Twente, Enschede, The Netherlands

Abstract. The KeY system offers a platform of software analysis tools for sequential Java. Foremost, this includes full functional verification against contracts written in the Java Modeling Language. But the approach is general enough to provide a basis for other methods and purposes: (i) complementary validation techniques to formal verification such as testing and debugging, (ii) methods that reduce the complexity of verification such as modularization and abstract interpretation, (iii) analyses of non-functional properties such as information flow security, and (iv) sound program transformation and code generation. We show that deductive technology that has been developed for full functional verification can be used as a basis and framework for other purposes than pure functional verification. We use the current release of the KeY system as an example to explain and prove this claim.

1 Overview

Motivation. Over the last decades the reach and power of verification methods and tools has increased considerably, and there has been tremendous progress in the verification of real world systems. The basic technologies of deductive program verification have matured. State of the art verification systems can prove functional correctness at the source code level for programs written in industrial languages such as Java.

The authors gratefully acknowledge support by the German National Science Foundation (DFG) under projects <http://www.key-project.org/DeduSec/> DeduSec and <http://www.se.tu-darmstadt.de/research/projects/albia/> ALBIA both within <http://www.spp-rs3.de/SPP> 1496 “Reliably Secure Software Systems – RS³” and under project IMPROVE within SPP 1593 “Design For Future – Managed Software Evolution”, as well as by the European Research Council (ERC) grant 258405 for the <http://fmt.cs.utwente.nl/research/projects/VerCors/> VerCors project.

While for many years the term *formal verification* had been almost synonymous with *functional verification*, in the last decade it became more and more clear that full functional verification is an elusive goal for almost all application scenarios. Ironically, this happened through the advances of verification technology: with the advent of verifiers that fully cover and precisely model industrial languages and that can handle realistic systems, it finally became obvious just how difficult and time-consuming the specification and verification of real systems is. Because of this, ‘simpler’ verification scenarios are often used in practice, relaxing the claim to universality of the verified properties.

Using deductive verification as core technology. In this paper, we show that deductive technology that has been developed for full functional verification (of Java programs) can be used as a basis and framework for other methods and purposes than pure functional verification:

- complimentary validation techniques such as testing and debugging,
- methods tackling the complexity of verification such as modularization and abstract interpretation,
- analyses of non-functional properties such as information flow security,
- sound program transformation and code generation.

We claim that for such an extended usage scenario, much of the work that went into the development of deductive verification systems can be reused. This includes the program logics and verification calculi that capture the semantics of the target programming language as well as the specification language, proof search mechanisms, user interfaces of semi-automatic verification systems that support proof construction and understanding proof states, interfaces to SMT (satisfiability modulo theories) solvers, as well as data structures for programs, specifications, and formulas, and associated parsers and pretty printers.

The KeY system. We use the current release of the KeY system [1] (KeY 2.2) to explain and prove the claim that deductive verifications methodology can serve as a platform for various verification and analysis methods, though other examples of this phenomenon, such as Boogie [2] and Why [3] verification frameworks, can be given. The KeY system is developed by the KeY project, a joint effort between the Karlsruhe Institute of Technology, Technical University of Darmstadt, and Chalmers University of Technology in Gothenburg, ongoing since 1999. KeY is free/libre/open source software and can be downloaded from <http://key-project.org/download/>.

Contents of this paper. In the following two sections, we describe the core technology for functional verification implemented in KeY: its program logic for Java and its sequent calculus, that provides symbolic execution for Java (Sect. 2), and its user interface (Sect. 3). Java is not a modular language. The specification of Java programs must support an appropriate mechanism that permits to decompose the verification target into components of manageable size that can

be verified separately. In Sect. 4 we show that such a technology can be seamlessly integrated into the core verification technology. In Sect. 5 we describe how symbolic execution calculi can be reused for abstract interpretation based verification. Verification methods such as symbolic execution can be used to generate tests from the specification and the source code (glass box testing) or only the specification (black box testing) of the verification target. As shown in Sect. 6, using reasoning techniques, one can generate tests that exercise particular program paths, satisfy various code coverage criteria, or cover all disjunctive case distinctions in the specification. A further use of verification for bug finding is to enhance the debugging process by using verification technology that is based on symbolic execution to implement symbolic debuggers (Sect. 7). Symbolic debugging covers all possible execution paths, and there is no need to initialize input values. Besides functional verification, the verification of non-functional properties, e.g., security properties, is of growing importance. In Sect. 8, we show that information flow properties can be verified by using functional verification methods as a basis. Finally, in Sect. 9, we show that a deductive calculus is a good basis for covering additional mechanisms of the programming language in a modular way (no need to build a new calculus or system). This is exemplified with Java Card’s transaction mechanism that is not part of standard Java.

2 A Prover Performing Symbolic Execution

The core of the KeY system consists of a theorem prover for a program logic that combines a variety of automated reasoning techniques. The KeY prover differs from many other deductive verification systems in that symbolic execution of programs, first order reasoning, arithmetic simplification, external decision procedures, and symbolic state simplification are interleaved.¹ For loop and recursion free programs, symbolic execution is performed in a fully automated manner.

The program logic supported by KeY is *Dynamic Logic* (DL) [5], a first order multi-modal logic. DL extends first order logic (FOL) with two families of modal operators: $\langle p \rangle$ (‘diamond’) and $[p]$ (‘box’) where p is a program fragment. 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.² Typically, ϕ is a FOL formula; in this case, $\langle p \rangle \phi$ corresponds to the weakest precondition of p w.r.t. ϕ . Another frequent pattern of DL is $\phi \rightarrow [p]\psi$, which corresponds to $\{\phi\}p\{\psi\}$ in Hoare logic [6]. DL is closed under all logical connectives. For instance, the formula $\exists v. ([p](x \doteq v) \wedge [q](x \doteq v))$ states that the final value of x is the same, whether we execute p or q .

To enable formal arguments about soundness and completeness, the KeY prover employs a *sequent calculus* for reasoning about Java DL formulas. Each

¹ The prover closest to KeY in this regard is KIV [4].

² This formulation assumes a deterministic programming language, like sequential Java in the context of KeY.

proof node is a sequent of the form $\Gamma \Rightarrow \Delta$, where Γ and Δ are sets of formulas, with the intuitive meaning that the conjunction of the assumptions Γ implies at least one of the formulas in Δ .

A proof in KeY consists of logical rule applications on DL sequents, using a proof strategy called *symbolic execution*. It is exactly this principle which makes the KeY prover an excellent basis for the various techniques described in this paper. We exemplify the principle of KeY style symbolic execution: consider the sequent (1), with precondition $x > y$ and postcondition $y > x$. The program in the modality swaps the values stored in x and y , using arithmetic.

$$x > y \Rightarrow \langle x=x+y; y=x-y; x=x-y \rangle y > x \quad (1)$$

To prove this formula, KeY symbolically executes one statement at a time, turning Java code into a compact representation of its effect.⁴ This representation is called *update*, essentially an explicit substitution, to be applied at some later point. In our example, symbolic execution of $x=x+y; y=x-y; x=x-y$; results in

$$x > y \Rightarrow \{x := y \parallel y := x\} \langle \rangle y > x \quad (2)$$

The expression $x := y \parallel y := x$ is an update. The symbol \parallel indicates its *parallel* nature. Once the modality is empty, it is discarded, and the accumulated update is applied to the postcondition $y > x$, leading to the proof goal $x > y \Rightarrow x > y$, that can be closed immediately. The update application has swapped x and y , translating the condition on the intermediate state into a condition on the initial state. The interleaving of collecting and applying updates very much facilitates *forward* symbolic execution. This is exploited not only for giving the proving process an intuitive direction, but also as a basis for realizing the other features of the KeY platform outlined in this paper.

To reason efficiently in a rich program logic for a target language like Java, a large number of sequent calculus rules are needed (over 1500 in the standard configuration). To implement these efficiently and to permit external validation of the rules, we use so-called *taclets*, described in [1, Chap. 4]. Unbounded loops cannot be handled by symbolic execution alone. KeY has invariant and induction rules for this purpose [1, Sect. 3.6], see also Sect. 5 below. Method calls can be handled either by inlining the method body or by replacing a method invocation by the method’s specification, see Sect. 4 for an example.

3 User Interface

The KeY verification system uses a graphical user interface (GUI) that is designed to make the interactive construction of formal verification proofs intuitive and efficient. During formal verification a vast amount of technical information is generated. The GUI helps the proof engineer to access relevant information.

Figure 1 shows KeY’s GUI with a loaded and partly performed proof task. Problem files containing Java code and specifications as well as (partial) proofs

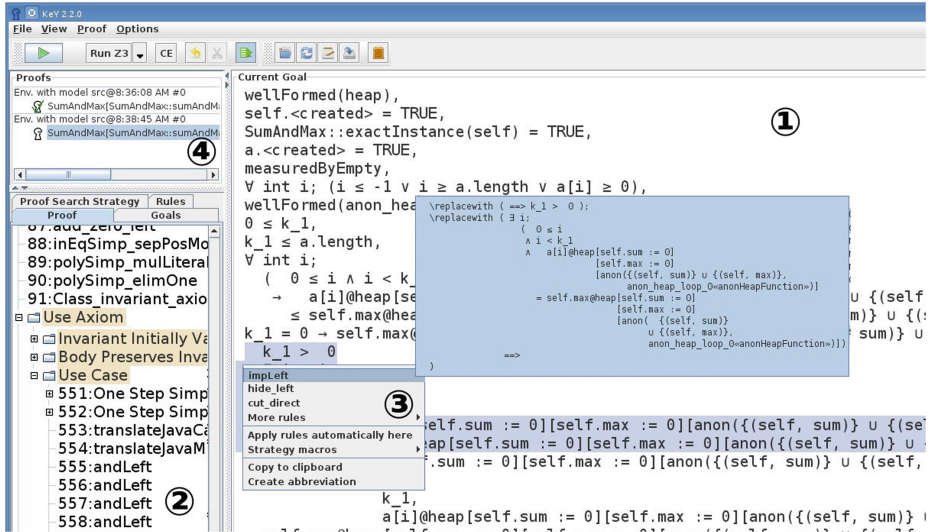






Fig. 1. The main window of the KeY GUI

can be loaded into the KeY system by selecting ‘File → Load’ or using the button . A new ‘Proof Management’ window appears and the user can choose the proof obligation to verify. Complete and partly finished proofs are also listed. If code or specifications that were used in a proof have changed, this is indicated as well.

After choosing a proof obligation, the corresponding sequent formula is constructed and shown in the right pane of the main window, see Fig. 1 (1). Then the user can start the proof construction. In the upper left pane of the main window the opened proofs are listed. Clicking allows to switch between different proofs. The icon next to the proofs indicates whether a proof is complete () or it contains open goals (). Below is a pane with different tabs to display the open goals, the proof tree, strategy settings for the proof search and information about the calculus rules. In Fig. 1 (2) the proof tree pane is shown. Each node of the proof tree is annotated either with the applied proof rule (e.g., `implLeft`) or with information about the proof step (case distinction, invariant, etc.). A right click on the proof tree or a node in it produces a context menu with possible actions on proof trees. The ability to prune, collapse, or unfold (parts of) the proof tree are indispensable for navigation and understanding in larger proofs. The user may also annotate proof nodes with free textual comments.

On the right pane (‘Current Goal’) of the of the main window the sequent of the selected proof node is shown. Pointing and right-clicking on parts of the sequent produces a context menu with a list of applicable proof rules for the highlighted formula, see Fig. 1 (3). Hovering over the rules shows a tooltip with the result that each rule gives when applied.

Hovering over operators in the ‘Current Goal’ pane, the subexpressions connected with that operator are highlighted. To search for formulas or proof nodes there are two mechanisms, one that searches for (sub)formulas in the sequent view and one that searches for proof nodes in the proof tree. These searches are useful when trying to understand how a formula has changed during the proof process. All buttons and menu entries as well as strategy settings have related tooltips, which briefly describe their functionality.

KeY provides a counter example generator (button ) that transforms a proof goal to an SMT formula over bitvector types and feeds it to an SMT solver. If a counterexample is found, it is presented to the user for inspection. This feature can help the user in cases where it is unclear whether the current proof goal is valid or there is a flaw in the specification or code.

User interaction during proof search. The KeY prover attempts to automate proof search as much as possible, but it also supports user interaction to guide the proof process. Proofs for valid formulas are often, but not always found automatically. For example, if complex quantifier instantiations are required, proof search may fail. When automated proof search fails, the user can apply inference rules interactively in a stepwise manner, as described above. The problem is that after a failed automated verification attempt, the user may be confronted with an intermediate proof object that is difficult to understand, because the automatic proof strategy tends to produce normal forms that are hard to read. This led us to pursue a semi-automated proof style where the user does not apply every step manually, but interacts with the automated strategy only at certain points of interest. KeY provides composite interaction steps, so-called *strategy macros*, that combine the application of basic deduction steps to achieve a specific purpose. The available strategy macros in KeY include: *Finish symbolic execution* symbolically executes Java programs in modalities. *Propositional expansion* applies only propositional rules. *Close provable goals* closes any open goal that is automatically provable, but does not touch goals where no automatic proof is found. The strategy macro *Autopilot* applies these three substrategies in this order. It divides the proof obligation into small subcases and thus guides the user to those points of the specification for which the automated proof failed.

4 Modular Specification and Verification

A crucial goal for any formal verification system is the ability to modularize a larger target program into manageable subtasks. In the context of KeY this concerns the program written in Java and the Java Modeling Language (JML) [7] for its specification. We have extended JML with concepts that support abstraction and modular verification to a language called JML* [8]. In the recent verification events in which KeY participated (VSTTE 2010, FoVeOOS 2011, VSTTE 2012, and FM 2012; cf. [9]), these concepts have proven to be effective.

Abstraction in JML specifications is provided through model fields, model methods, and ghost fields. When specifying object-oriented code modularly, it is generally important that an abstraction of the state of an object exists and can be used in other parts of the specification. This way, details of the object’s implementation need not be revealed, which lets the verification both scale better and become more modular. In KeY we follow this general principle.

While similar in appearance to fields in Java, *model fields* are declared for specification and verification only, which allows the specifier to use JML* features beyond Java expressions for their definition (including quantification). The value of a model field is computed from the system state to which it is coupled through a **represents** clause that is understood as a logical axiom. *Ghost fields*, too, are only visible during verification. Unlike model fields, however, their value does not depend on the state but is part of it (like a Java field). Both abstraction techniques (ghost and model) can be used within the same specification in KeY.

Going beyond original JML, specification-only program elements in JML* allow the use of *abstract data types* (ADTs). When reasoning about concrete, mutable data structures, e.g., linked lists or trees, we are usually only interested in properties regarding the payload within these structures. ADTs provide an abstraction from the implementation, concealing details about the linked data structure that resides on the heap. JML* provides the two built-in ADTs `\seq` (finite sequences) and `\locset` (sets of memory locations; see below).

Modularity of verification is provided through the concept of design by contract [10]. Every method implementation is verified against its contract. Since method invocations are abstracted by their contracts, contracts proved correct remain valid even in case that new code has been added to the program. Method contracts do not only contain pre- and postconditions (to describe their intended behavior), but also *frame conditions* (to describe what must be *preserved*). In JML*, a frame is a set of heap locations to which a method may write at most.

```

1 public interface List {
2   /**@ public model instance \seq conts;
3   /**@ accessible contents: footp;
4
5   /**@ public model instance \locset footp;
6   /**@ accessible footp: footp;
7
8   /**@ public normal_behavior
9   @ ensures \result == conts.length;
10  @ accessible footp;          @*/
11  public /*@pure@*/ int size();
12
13  /**@ public normal_behavior
14  @ ensures conts == \seq_concat(
15  @   \old(conts), \seq_singleton(x));
16  @ ensures \new_elems_fresh(footp);
17  @ assignable footp;          @*/
18  public void append(int x);  }

```

Fig. 2. List specified with model fields

In some cases, the locations of a frame are known beforehand and can be simply enumerated (static framing). For rich heap data structures, however, there is a need to describe all locations that ‘belong to’ the data structure; a set that may change during a run of the program. Such a set of locations is called a *footprint*. The technique to specify frames and footprints in JML* and to reason about them in KeY is the *dynamic frames* approach [8,11], that introduces a type `\locset` for location sets. Frames and footprints can be given in an abstract, possibly recursive, manner.

For instance, the footprint of a linked list is the union of its head node’s locations (for its local data) and the footprint of its tail (if not `null`).

Figure 2 shows an example of a list interface, that is specified using two model fields: `conts` for its contents and `footp` for its footprint. The mutator `append()` *changes only* the footprint of this instance (L. 17, `assignable` clause), while the pure method `size()` *depends only on* this footprint (L. 10, `accessible` clause). If for two lists `a` and `b` the footprints, `a.footp` and `b.footp` denote disjoint sets, we can conclude—without any knowledge about implementation—that a call to `a.append()` does not have an influence to the result of `b.size()`. Otherwise, `a` and `b` could be *aliased deeply*, for instance, `a` could be a tail of `b`. The predicate in L. 16 expresses that only fresh object references may be added to the footprint ensuring that sets are still disjoint after the method call if they were before.

Other verification frameworks have similar concepts of modularization: Dafny [12] uses less fine-grained dynamic frames, ghost state, and pure functions, and allows for user-defined ADTs. An alternative approach to the framing challenge is separation logic. VeriFast [13] allows the modular specification of Java and C code based on separation logic together with user defined ADTs and lemmas. VCC [14] uses ownership to deal with that challenge.

5 Abstract Interpretation

Achieving a high degree of automation is still a challenge in program verification. The nature of user interactions is either direct with the underlying theorem prover (cf. Sect. 3) or it is implicit in the need to provide specifications such as method contracts, loop invariants or induction hypotheses. SMT solvers and automated theorem provers have improved considerably during the previous decade such that writing and finding specifications is now the main bottleneck for program verification. In this section, we briefly sketch our approach to achieve higher automation by generating loop invariants automatically, using abstract interpretation techniques [15]. More details on the approach are given in [16] which, however, was only implemented recently and is available at <http://www.se.tu-darmstadt.de/research/projects/albia/download/>.

In a nutshell, our approach works as follows: the verification process starts as usual with a DL formula that represents a proof obligation, for instance, that a method `m()` satisfies its contract. The automated proof search executes `m()` symbolically. As Java DL models the semantics of sequential Java faithfully and precisely, we do not lose any precision until we reach a loop. In general, the user would now have to provide a loop invariant either annotated in the source code as a JML loop invariant specification or entered interactively when the loop is encountered during the proof. Instead, we use abstraction to avoid the need for a user-supplied invariant. But unlike in abstract interpretation, we avoid to abstract the symbolically executed program. Instead, we abstract only part of the symbolic state when the loop is encountered, namely that part which is possibly modified by the loop. The (automatically proven) soundness condition is that the abstract symbolic state represents at least all concrete states that are

reachable by exiting the loop. For the part of the symbolic state that has not been abstracted, no precision is lost.

We illustrate the process with a small example: the loop in Fig. 3 increases the program variable `i` until `n` is reached. For ease of presentation we choose exemplarily a trivial abstract domain for integers, namely, the sign domain $\{0, \leq, \geq, <, >, \top\}$ which classifies integers into zero, non-positive, non-negative, negative, positive, or any integer.

```
i = 0;
while (i < n)
    i = i + 1;
```

Fig. 3. Loop example for symbolic state abstraction

On reaching the loop, the symbolic state looks as follows: $n: n_0, i: 0$ where n_0 is a symbolic value representing an unknown but concrete value. Note, both values are not abstract and no precision has been lost until this point. Abstraction of the symbolic state begins by unwinding the loop and analyzing which values have been changed, that is, one compares the state before entering the loop with the state after the first loop iteration. The only changed value is that of `i`. The most precise abstract value that we can give to `i` and that is valid before and after executing the body is \geq . Unwinding the loop once more and re-computing the abstract value for `i` gives no change. We found a fixed point and the abstracted symbolic state is $n: n_0, i: \geq$, which is used to continue symbolic execution after the loop.

In contrast to approaches like CEGAR [17], which use a counterexample guided refinement loop approach (i.e., a coarse abstraction is stepwise refined in case of a spurious counterexample), we start with a fully precise modeling and loose precision only when needed and only for a localized (and often small) part of the symbolic state. As is true for all abstraction based approaches, we loose some precision, and thus completeness, in exchange for higher automation. However, the trade off is more than justifiable when targeting specific program properties like secure information flow (see Sect. 8), absence of null pointer exceptions, etc.

Combining deductive verification and abstract interpretation has also been pursued by Leino and Logozzo [18]. They use a theorem prover from within an abstract interpretation system to compute loop invariants on demand. However, the abstract interpretation system and the theorem prover remain separate systems. Deep integration of abstract interpretation into deductive verification based on dynamic logic has also been proposed by [8] using the technique of predicate abstraction [19].

6 Test Case Generation

Even though the area of deductive verification made tremendous progress and provided powerful tools, deductive methods still require expert level knowledge. As a lightweight technique, KeY offers a verification based test case generation facility [20,21], where deductive verification is used as a base technology. From source code augmented with JML specifications, KeY generates proof obligations in dynamic logic. During verification with the prover, the proof branches over the necessary case distinctions, largely triggered by boolean decisions in the source

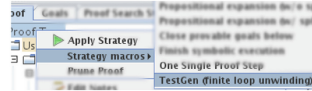
```

1 final class List { /*@ nullable */ public List nxt;
2
3 public /*@ pure nullable */ List get(int i){
4     return i==0?this:(nxt==null || i<0)?null:nxt.get(i-1);
5 }
6
7 /*@ public normal_behaviour
8   requires a.length>0 && l!=null;
9   ensures (forall int i:0<=i&&i<a.length;a[i]=l.get(i));*/
10 public void L2A(/*@nullable */List l, List[] a){
11     for(int i=0; i<a.length; i++){ a[i]=l;
12         if(l==null) break; l=l.nxt;
13     } }

```

Fig. 4. Method L2A violates its contract

Step 1. Create a proof tree




Step 2. Press the  button




Fig. 5. Test generation steps

code, as explained in Sect. 2. On each proof branch, a certain path through the program is executed symbolically. KeY TestGen uses the same machinery for a different purpose, namely generating JUnit test cases. The idea is to let the prover construct an unfinished proof tree (with a bounded number of loop unwindings), then to read off a *path constraint* from each branch, i.e., a constraint on the input parameters and initial state for this path. We generate concrete test input data satisfying each of these constraints, thereby achieving strong code coverage criteria, in particular MCDC (Modified Condition/Decision Criterion), by construction.

In addition to the source code, KeY’s test generation facility requires formal specifications, for two purposes. First, specifications are needed to complete the test cases with *oracles* to check the test’s pass/fail status. The second role of specifications is to allow symbolic execution of method calls within the code under test. The prover can use the specification, rather than implementation, of called methods to continue symbolic execution. In particular, frequently used library methods need to be specified.

As an example, Fig. 4 shows a class `List` (representing a list node). Method `get` returns the `i`-th list node starting from `this` and following the `nxt` field. The intended behavior of method `L2A` is to copy list elements starting from `l` into the array `a`—as many as fit into the array. The user may not see the mistake in the code and spend valuable time with failed verification attempts. However, the problem can be quickly detected using KeY’s test generation functionality.

The first step is to create a proof tree. For example, to execute all program paths with a bound on loop unwindings, the user may choose the strategy macro ‘TestGen’ (Fig. 5, Step 1). By pressing the  button (Step 2), a test suite is generated which constructs different method arguments and creates various list shapes by initializing the `nxt` field, such that every case distinction in the proof tree (and hence in the program) is satisfied and executed. To detect the fault in method `L2A`, a test case is needed that executes the loop at least two times, i.e. `a.length ≥ 2`. To fix the method `L2A`, Line 11 must be replaced with `if(l!=null)l=l.nxt`; which ensures that the rest of the array is initialized with `null` if the end of the list is reached.

Besides generating test cases in order to find out why a proof cannot be closed, we can generate them out of a closed proof tree. In this case a test suite

covering all feasible paths is created. This suite can be used for regression testing the software.

The usefulness of combining proofs and tests has been recognized in the last decade, leading to the conference series Tests and Proofs. A recent extension of a deductive verification tool with test generation capabilities is based on Frama-C [22]. A set of popular test generation tools that are based on symbolic execution and its variants is described in [23].

7 Debugging and Visualization

The Symbolic Execution Debugger (SED) [24, 25] is an Eclipse extension that executes Java methods symbolically. It is implemented on top of KeY and offers interactive execution control just like a traditional debugger, including stepwise execution and suspension at breakpoints.

Symbolic execution makes it possible to explore *all* concrete execution paths of a program (up to a finite depth) in the symbolic states of a single symbolic execution run. The result is a *symbolic execution tree* (SET). In this sense, performing a proof in KeY realizes a sound, fully automatic, general purpose symbolic execution engine for Java. A specific proof search strategy guarantees that symbolic execution reflects the actual evaluation sequence defined by Java semantics. JML specifications are not required, but can be used during symbolic execution. Specifically, loop invariants ensure finite symbolic execution trees in presence of loops; method contracts permit to handle methods for which the source code is not available and guarantee finite symbolic execution trees in presence of recursive method calls.

Debugging by symbolic execution is interesting for various reasons. Most importantly, symbolic execution can start at any method or any other statement in a program, no fixture is required. The initial state can be specified partially or not at all. As all execution paths are covered, it is not necessary to set up a concrete initial program state leading to an execution where a targeted bug occurs. Because symbolic execution can be started ‘close’ to the suspected location of a bug and the symbolic states contain only program variables accessed during execution, the intermediate states of symbolic execution tend to be small and simple. This makes it easy for the bug hunter to comprehend intermediate states and the actions performed on them to find the origin of a bug. Finally, the intended behavior of a program is correctly reflected in its symbolic execution, which, therefore, will not cause a program error that disappears while debugging. The underlying reason is that classical debuggers interact and influence the execution of the analyzed program.

Figure 6 shows a debugging session where method `eq()` is inspected. Its full SET is displayed in the view on the left. Different icons emphasize the role of each node. The root is a start node representing the program fragment under execution. After the call to `eq()` the if-guard `this.value == n.value` is evaluated, which involves an access to the instance variable `value` of parameter `n`. As we know nothing about `n`—it might well be `null`—symbolic execution branches. The tree

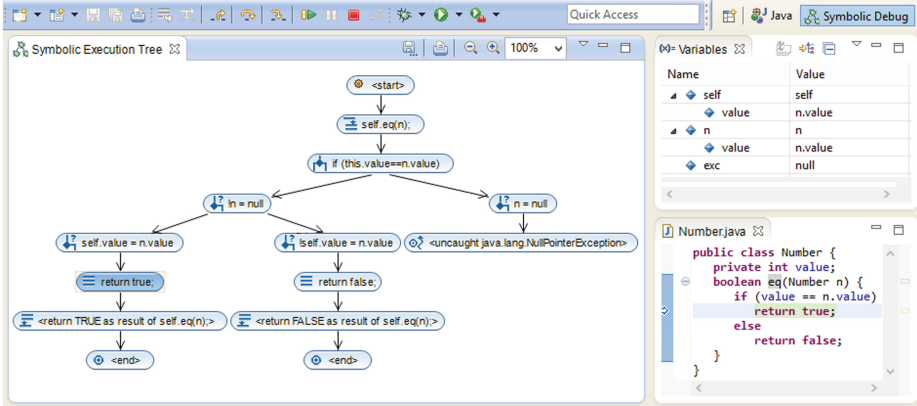


Fig. 6. Symbolic Execution Debugger: debugging method `eq` symbolically

branches are labelled with the condition under which each path is taken: if `n` is `null` then execution terminates with an uncaught `NullPointerException`, which may or may not be intended behavior. In the latter case, it directly points to a bug. Otherwise, the guard can be evaluated to either true or false. In each case a return statement is executed next.

The symbolic program state of a selected node is shown in the *Variables* view. The `value` fields of instance variables `self` and `n` have an identical symbolic value. So either `self.value` and `n.value` have the same value or `self` and `n` refer to the same object. Aliasing is a common source of bugs and SED helps to find them by visualizing all nonisomorphic memory layouts fulfilling a symbolic state.

EFFIGY [26] was the first system that allowed to interactively execute a program symbolically in the context of debugging. It did not support specifications or visualization. Behavior trees [27] are an abstract visual notation to specify the behavior of software systems. These are derived from a requirements analysis rather than from source code and they do not represent symbolic states.

8 Information Flow Analysis

Programs with publicly accessible interfaces (like web applications) are increasingly used to process confidential data. This raises the importance of information flow control within such applications: confidential information must not leak to public outputs. *Information flow* is the degree to which the initial values of variables containing confidential data (‘high’ variables) interfere with the final values of publicly observable (‘low’) variables. Formal techniques for information flow analysis and control are concerned with showing that information flow is absent or limited in a program. A survey is available in [28], though many advances have occurred since its publication. Three approaches for analyzing information flow have been implemented using KeY.

The first approach is based on *self-composition* [29,30], which is appealing because it is semantically precise, supports semantic declassification (i.e., accepting that specific parts or properties of confidential information does become public), and can be realized on top of a software verification systems like KeY in a direct manner. The implementation in KeY [31] (which can be seen as a direct formulation of information flow in Java DL) symbolically executes a program p twice with equal symbolic values for the low variables but possibly different values for high variables. Absence of information flow is shown by proving that the symbolic values for the low variables are equal in the respective final states.

Beyond this basic idea, the approach and its implementation feature several optional refinements. First, it is possible to execute p symbolically only once and combine the obtained verification conditions. Second, if p is decomposable into individual parts, each without information flow, then these parts can be considered independently reducing the number of code paths that need to be reasoned about. Third, the analysis supports not only primitive types but also object references as secret and publicly observable values [32]. Finally, modular contracts used for functional verification (Sect. 4) can be used when proving absence of information flow as well.

Specifying information flow policies that programs must adhere to happens with an extension of JML [33]. The language allows a convenient and fine-grained specification of declassification and erasure by assigning security levels (high/low) to terms instead of variables and fields.

Another approach is to track information flow with ghost states [34]. It aims at a higher degree of automation and higher efficiency by trading precision. Declassification is supported. The approach can be combined with abstract interpretation (Sect. 5) and thus holds the potential for increasing automation by inference of suitable invariants. In this approach, we complement each program variable with a ghost variable that overapproximates the set of locations the actual variable depends on. When a program variable is assigned a new value t , its corresponding ghost variable is automatically updated too. The new value of the ghost variable is the union of the dependencies of all variables occurring in t (plus implicit dependencies caused by control flow). A program is secure if the set of calculated dependencies is a subset of those allowed by the specification.

The third approach combines KeY with external tools for projection computation and model counting in a tool chain for *quantitative* information flow analysis of imperative programs [35]. The user does not specify what information is acceptable to declassify, but instead the tool chain computes a number of information-theoretical measures (e.g., Shannon entropy or min-entropy) reflecting the amount of confidential information in bit disclosed by the program.

9 Verification of Java Card Applets

One of the specific strengths of KeY is its complete support for verification of programs written in Java Card [36], a dialect of Java for smart cards. This includes support for all features specific to the Java Card platform. These are

```

a[0] = 0;
JCSytem.beginTransaction();
Util.arrayFillNonAtomic(a,0,1,1);
a[0] = 2;
JCSytem.abortTransaction();
/*@ assert a[0] == 1;

a[0] = 0;
JCSytem.beginTransaction();
a[0] = 2;
Util.arrayFillNonAtomic(a,0,1,1);
JCSytem.abortTransaction();
/*@ assert a[0] == 0;

```

Fig. 7. Two programs exhibiting subtleties of the Java Card memory management

the memory model that distinguishes between persistent and transient data as well as a transaction mechanism that ensures atomic updates of the persistent memory of the device [37, Chap. 7]. Each Java Card device is equipped with two types of memory: (i) persistent memory that keeps its contents between card power-ups (i.e., sessions), (ii) transient (scratch pad) memory that is reset on every power-up. Consequently, the semantics of a primitive assignment to an array element³ depends on the kind of memory that the array is allocated to. Moreover, the transaction mechanism allows to group several assignments into atomic blocks and to collectively undo several assignments in one system API call.

On top of this, there is a specific interplay between special system API calls and regular assignments that involve the same persistent data. We illustrate this with the two programs in Fig. 7 that are both correct relative to their stated `assert` annotations. The call to the `arrayFillNonAtomic` method assigns value 1 to the array element `a[0]`. In principle, it should bypass any rollback effects of `abortTransaction` (which is what indeed happens in the program on the left), however, an earlier regular assignment to `a[0]` inside the same transaction disables this bypass effect of `arrayFillNonAtomic`.

A correct treatment of situations like the one in Fig. 7 in the underlying program logic may be expected to be difficult. Indeed, previous formalizations of Java Card were quite complex [38, 39]. In KeY 2.2 we use the explicit heap model to our advantage: with an additional heap variable the Java Card memory model is formalized in a completely modular manner. This is achieved by adding a handful of carefully crafted rules for entering and exiting transactions, and assigning array elements in transaction contexts [40]. The introduction of an additional heap variable also involves a slight, yet fully transparent, extension of the JML* specification language to enable sound and complete modular verification also of programs involving Java Card transactions.

References

1. Ahrendt, W.: Using KeY. In: Beckert, B., Hähnle, R., Schmitt, P.H. (eds.) *Verification of Object-Oriented Software*. LNCS (LNAI), vol. 4334, pp. 409–451. Springer, Heidelberg (2007)

³ Objects other than arrays are not subject to the described mechanisms.

2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., M. Leino, K.R.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Filiâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
4. Stenzel, K.: A formally verified calculus for full Java Card. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 491–505. Springer, Heidelberg (2004)
5. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
6. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580, 583 (1969)
7. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT* **31**(3), 1–38 (2006)
8. Weiß, B.: Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction. Ph.D. Thesis, Karlsruhe Institute of Technology (2011)
9. Bruns, D., Mostowski, W., Ulbrich M.: Implementation-level verification of algorithms with KeY. *Softw. Tools Technol. Transf.* (Springer, Heidelberg) to appear. DOI:[10.1007/s10009-013-0293-y](https://doi.org/10.1007/s10009-013-0293-y)
10. Meyer, B.: Applying “design by contract”. *IEEE Comput.* **25**(10), 40–51 (1992)
11. Kassios, I.T.: The dynamic frames theory. *Form. Asp. Comput.* **23**(3), 267–288 (2011)
12. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
13. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
14. Schulte, W., Songtao, X., Smans, J., Piessens, F.: A glimpse of a verifying C compiler. In: *C/C++ Verification Workshop* (2007)
15. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Fourth ACM Symposium on Principles of Programming Language*, Los Angeles, pp. 238–252. ACM Press, New York (1977)
16. Bubel, R., Hähnle, R., Weiß, B.: Abstract interpretation of symbolic execution with explicit state updates. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 247–277. Springer, Heidelberg (2009)
17. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
18. M. Leino, K.R., Logozzo, F.: Loop invariants on demand. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 119–134. Springer, Heidelberg (2005)
19. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
20. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 169–188. Springer, Heidelberg (2007)

21. Beckert, B., Gladisch, C.: White-box testing by combining deduction-based specification extraction and black-box testing. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 207–216. Springer, Heidelberg (2007)
22. Petiot, G., Kosmatov, N., Giorgetti, A., Julliand, J.: How test generation helps software specification and deductive verification in Frama-C. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 204–211. Springer, Heidelberg (2014)
23. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) ICSE, pp. 1066–1071. ACM (2011)
24. Hentschel, M., Bubel, R., Hähnle, R.: Symbolic execution debugger (SED). In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 255–262. Springer, Heidelberg (2014)
25. Hentschel, M., Hähnle, R., Bubel, R.: Visualizing unbounded symbolic execution. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 82–98. Springer, Heidelberg (2014)
26. King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7), 385–394 (1976)
27. Dromey, R.G.: From requirements to design: Formalizing the key steps. In: 1st International Conference on Software Engineering and Formal Methods, SEFM, IEEE (2003)
28. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE J. Sel. Areas Commun.* **21**(1), 5–19 (2003)
29. Barthe, G., D’Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Proceedings of the 17th IEEE workshop on Computer Security Foundations, CSFW ’04, Washington, USA, pp. 100–115. IEEE CS (2004)
30. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)
31. Scheben, C., Schmitt, P.H.: Efficient Self-composition for weakest precondition calculi. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 579–594. Springer, Heidelberg (2014)
32. Beckert, B., Bruns, D., Klebanov, V., Scheben, C., Schmitt, P.H., Ulbrich, M.: Information flow in object-oriented software. In: Gupta, G., Peña, R. (eds.) Logic-Based Program Synthesis and Transformation, pp.15–32 (2013)
33. Scheben, C., Schmitt, P.H.: Verification of information flow properties of JAVA programs without approximations. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 232–249. Springer, Heidelberg (2012)
34. van Delft, B.: Abstraction, objects and information flow analysis. Master’s Thesis, Institute for Computing and Information Science, Radboud Uni Nijmegen (2011)
35. Klebanov, V.: Precise quantitative information flow analysis: A symbolic approach. *Theor. Comput. Sci.* **538**, 124–139 (2014). (to appear)
36. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer’s Guide. Addison-Wesley, Boston (2000)
37. Oracle: Java Card 3 Platform Runtime Environment Specification, Classic Edition, Version 3.0.4., September 2012
38. Mostowski, W.: Formal reasoning about non-atomic JAVA CARD methods in dynamic logic. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 444–459. Springer, Heidelberg (2006)

39. Marché, C., Rousset, N.: Verification of Java Card applets behavior with respect to transactions and card tears. In: Proceedings of Software Engineering and Formal Methods (SEFM), Pune, India. IEEE CS Press (2006)
40. Mostowski, W.: A case study in formal verification using multiple explicit heaps. In: Beyer, D., Boreale, M. (eds.) FORTE 2013 and FMOODS 2013. LNCS, vol. 7892, pp. 20–34. Springer, Heidelberg (2013)