Integrating Verification and Testing of Object-Oriented Software

Christian Engel, Christoph Gladisch, Vladimir Klebanov, and Philipp Rümmer

www.key-project.org

Abstract. Formal methods can only gain widespread use in industrial software development if they are integrated into software development techniques, tools, and languages used in practice. A symbiosis of software testing and verification techniques is a highly desired goal, but at the current state of the art most available tools are dedicated to just one of the two tasks: verification or testing. We use the KeY verification system (developed by the tutorial presenters) to demonstrate our approach in combining both.

1 What KeY Is

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

Recently, version 1.0 of the KeY system has been released in connection with the KeY book [2]. The KeY system is written in JAVA and runs on all common architectures. It is available under GPL and can be downloaded from www.key-project.org.

1.1 Towards Integration of Formal Methods

Formal methods can only gain widespread use in industrial software development if they are integrated into software development techniques, tools, and languages used in practice. KeY integrates with (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 offers then the *extended functionality* to generate proof obligations from selected parts of specifications and verify them with the KeY prover. The core of the KeY system, the KeY verification component, can also be used as a stand-alone prover, though.

The KeY project is constantly working on techniques to increase the returns of using formal methods in the industrial setting. Recent efforts in this area concentrate on applying verification technology to traditional software processes. These have resulted in development of such approaches as symbolic debugging

and verification-based testing. The latter is the central topic of this tutorial with Section 3 explaining why and how to utilise synergies between verification and testing.

1.2 Full Coverage of a Real-world Programming 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 [5] because of its importance for security- and safety-critical applications.

For specification, KeY supports both the OMG standard Object Constraint Language (OCL) [20] and the Java Modeling Language (JML) [16], which is increasingly used in the industry. In addition, KeY features a syntax-directed editor for OCL that can render OCL expressions in several natural languages while they are being edited.

The KeY prover and its calculus [2] 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 [15] and JML[1]) 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.

To a certain degree, KeY allows to customise the assumed semantics of JAVA CARD. For instance, the user can choose between different semantics of the primitive JAVA integer types. 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 incomplete for programs that depend on JAVA's finite ring semantics), and a faithful semantics of JAVA integers (sound and complete, but difficult to verify).

2 Foundations of KeY

2.1 The 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 employs a logic called JAVA CARD DL, which is an instance of *Dynamic Logic* (DL) [12]. DL, like Hoare Logic [14], has the advantage of transparency with respect to the program to be verified. This means, 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"

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

directly on the JAVA CARD source code. This transparency is extremely helpful for proving problems that require a certain amount of human interaction.

DL itself 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. Under program we understand a sequence of JAVA CARD statements.

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.[2]

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 $\{\psi\}\ p\ \{\phi\}$.

The following is an example of a JAVA CARD DL formula:

$$\texttt{o1.f} < \texttt{o2.f} \rightarrow \langle \texttt{int t=o1.f; o1.f=o2.f; o2.f=t;}\rangle\ \texttt{o2.f} < \texttt{o1.f}$$

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, correctness of program transformations, or the validity of assignable clauses. Also, a pre- or postcondition 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 [2].

2.2 Verification As Symbolic Execution

The actual verification process in KeY can be viewed as *symbolic execution* of source code. Unbounded loops and recursion are either handled by induction over data structures occurring in the verification target or by specifying loop invariants and variants. Symbolic execution plus induction as a verification paradigm was originally suggested for informal usage by Burstall [4]. The idea to use Dynamic Logic as a basis for mechanising symbolic execution was first realised in the Karlsruhe Interactive Verifier (KIV) tool [13]. Symbolic execution is very well suited for interactive verification, because proof progress corresponds to

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

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) perform substitutions on formulae to record state changes of a program. In the KeY approach to symbolic execution, the application of substitutions is *delayed* as much as possible; instead, the state change effect of a program is made *syntactically explicit* and accumulated in a construct called *updates*. Only when symbolic execution has completed are updates turned into substitutions. For more details about updates we refer to [2].

The second foundation of symbolic execution, next to updates, is *local 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.

2.3 Automated Proof Search

For automated proof search, a number of predefined strategies are available in KeY, which are optimised, for example, for symbolically executing 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 automated search for quantifier instantiations uses meta variables 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].

KeY also offers an SMT-LIB backend[3] for proving near-propositional proof goals with external decision procedures.

2.4 User-friendly Graphical User Interface

Despite a high degree of automation (see Sect. 2.3), 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). In the case of human-guided

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

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. These tasks are supported by dynamic context menus and drag-and-drop.

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.

3 Integrating Verification and Testing

3.1 Why Integrate

Although deductive verification can achieve a level of reliability of programs that goes beyond most other analysis techniques, there are reasons to augment fully-symbolic reasoning about programs with execution of concrete tests. We distinguish two classes of reasons.

The first class involves failing or inapplicable verification. In many common cases it is impossible to apply verification successfully: be it because no full formal specification is available, because verification is too costly, or simply because the program at hand proves to be incorrect. Moreover, once a verified program is (even slightly) changed, existing correctness proofs become invalid and have to be repeated. We will show how verification technology can be applied also in such situations by generating test-cases based on symbolic execution of programs [10, 1], and by turning proof search into a systematic bug search [18].

The second class is due to principal shortcomings of formal verification. Symbolic reasoning about programs on the source code level does not take all phenomena into account that can occur during the actual program execution. It happens routinely that a JAVA CARD application works perfectly on the desktop emulator, but behaves erroneously once deployed on the card. This is typically because the card does not provide a JAVA CARD virtual machine that fully complies with the semantic model used for the verification. As it is simply too complex to formally specify and verify compilers, protocols, smart card operating systems, virtual machine implementations, etc., testing is essential even if a complete proof has been found. The KeY system can automatically generate test-cases from proofs and thus simplifies testing after verification.

3.2 Generating Test Cases From Proofs

The KeY tool integrates all necessary steps for generating comprehensive JUnit tests for white-box testing. The major steps are (1) computation of path conditions with verification technology, (2) generation of concrete test data by constraint solving, and (3) generation of test oracles. The KeY tool can also be combined with existing black-box tools by outsourcing the second and third steps and achieving synergy effects between the tools.

In the following, we assume that we have a program under test p (PUT) and its specification ϕ, which can be a contract (i.e., a pre- and a postcondition) or

an invariant. Even very simple specifications yield useful test cases: A specification of total correctness with a postcondition *true* is sufficient to generate tests detecting uncaught exceptions.

3.3 Test-Case Generation by Bounded Symbolic Execution

The proof obligation resulting from the program p and the property ϕ is input into the KeY system, which symbolically executes p for up to a fixed number of steps. This produces a bounded symbolic execution tree, from which feasible execution paths and branches with the corresponding path and branch conditions are easily extracted. With the help of external arithmetics decision procedures like Simplify [9] or Cogent [8], concrete models for these path conditions are computed. These serve as test inputs for p. The property ϕ is translated into a test oracle. Thus, we obtain a test case for every feasible execution path of p (below the bound). The output of the process is a complete JUnit test case suite that requires no further modifications.

Let us consider a simple example program:

```
/*@ public normal_behavior
  @   ensures (
  @     \forall int i; 0<=i && i<arr.length; arr[i]<=\result);
  @*/
public int getMax(int[] arr){
  int max = arr[0];
  for(int i=1; i<arr.length; i++){
    if(arr[i]<max) max = arr[i];
  }
  return max;
}
```

The JML specification requires that `getMax()` terminates normally (without raising an exception) and the returned result is greater or equal than each of `arr`'s entries. This postcondition is translated by KeY into a test oracle, with universal quantifiers mimicked by loops:

```
private boolean oracle(int result, int[] arr){
  boolean b = true
  for(int i=0; i<arr.length; i++){
    b = b && arr[i]<=result;
  }
  return b;
}
```

Since we made no assumptions on `arr.length`, the number of loop iterations is bounded only by `Integer.MAX_VALUE`. Since the number of feasible execution paths through the loop is $2^{\texttt{arr.length}}$ (due to the **`if`** statement), it is technically not possible to create a test satisfying full feasible path coverage.

Nonetheless, we still get useful test cases if we symbolically execute the code by unrolling the **`for`** loop a bounded number of times, for instance just once.

These tests already catch both implementation bugs contained in `getMax()`. We now describe this in more detail.

The evaluation of the first statement `max = arr[0];` induces the following case distinction:

(1) `arr` $\neq$ **`null`** $\wedge$ `arr.length` ≥ 1: The execution proceeds normally. The path condition on this execution path is feasible.
(2) `arr` $=$ **`null`**: A `NullPointerException` is raised, but this branch condition is contradictory to the implicit JML assumption that arguments of a method are not **`null`**, unless declared `nullable`. KeY recognises this infeasibility and does not generate a test case.
(3) `arr.length` < 1: An `ArrayIndexOutOfBoundsException` is raised. This branch condition is feasible, and a test generated for this path detects that the implementation is erroneous, since we required normal termination of `getMax()`.

The symbolic execution now proceeds on the path (1) with unrolling the **`for`** loop once. After the first iteration of the loop we end up with an open proof tree containing 4 different feasible execution paths (of which 2 have not yet terminated) with path conditions:

(4) `arr.length` < 1: This path is identical to the path (3) above.
(5) `arr.length` $= 1$: The loop is never entered, since the loop guard is false.
(6) `arr.length` $> 1 \wedge$ `arr[1]` $<$ `arr[0]`: The guard `arr[i]<max` of the **`if`** statement is true, and `max` is set to `arr[1]`. This violates the postcondition.
(7) `arr.length` $> 1 \wedge$ `arr[1]` $\geq$ `arr[0]`: The guard of the **`if`** statement is false. The postcondition is also possibly violated on this path, namely in case `arr[1]` $\neq$ `arr[0]`.

The four test cases generated from this proof tree exercise `getMax()` on the paths (4)–(7). The test for (4) initialises `arr` with **`new int`**`[0]` and reports an error due to an exception thrown by `getMax()`. The test for (5) succeeds, while the tests for (6) and (7) report failures due to results not accepted by the test oracle.

3.4 Test-Case Generation from Method Specifications and Loop Invariants

An obvious deficiency of bounded symbolic execution is that it only explores a part of all program behaviours. The following example shows the problematic situation.

```
class Bar {
  final int[] arr = new int[16];
  void foo() {
    int max = getMax(arr);
    if(max<0) { A(); }
  }
}
```

We assume a correct implementation of `getMax()`, which is applied to a fixed-length buffer. In order to compute a path condition for the execution of the method `A()` the unwinding bound for symbolic execution of the loop in `getMax()` must be at least 16. This value, in general, is not practicable due to the exponential growth of execution trees. Even worse, the minimal unwinding bound of a loop for executing a certain branch is generally unknown.

An extension of the bounded approach allows generation of test cases based on loop invariants and method specifications in combination with symbolic execution. A loop or a method call is in this case replaced by the invariant resp. the method specification. With this technique we can compute precise conditions for entering a branch, even if the path passes through a loop or a method invocation.

For example the desired path condition for executing the method `A()` in the code above is: $\forall i.\ 0 \leq i \wedge i < 16 \rightarrow \texttt{arr}[i] < 0$. This path condition is computed with our approach by using the branch condition $\texttt{max} < 0$ and the postcondition of `getMax()`.

3.5 White-box testing by Combining Specification Extraction and Black-box testing

Existing black-box testing tools [7, 3, 6, 17] can be augmented by KeY to provide white-box testing capabilities. In this case, the external tool generates the test inputs and the oracle, while KeY provides information about program structure.

This information is extracted from the symbolic execution tree and input into the black-box testing tool as a part of the program specification. We call this process "structure-preserving specification extraction". The whole approach is illustrated in Figure 1.

Depending on the methods used for its extraction, the specification may not cover iterations of loops above a certain limit. However, by combining the extracted specification with a given requirement specification, black-box testing methods can generate tests that exercise random amounts of loop iterations including those not covered by the extracted specification alone. In this way, it is also possible to achieve a combination of code coverage and data coverage criteria from both techniques.

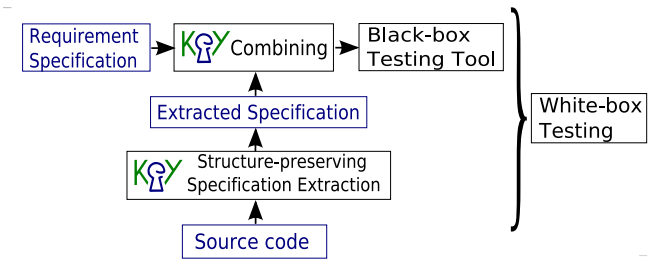

Fig. 1. White-box testing as black-box testing with path extraction.

3.6 Proving Incorrectness of Programs

All approaches to find program defects that have been described so far make use of the symbolic execution and reasoning capabilities of KeY without actually aiming at the construction of a complete proof. Due to the generality of JAVA CARD DL, however, the problem can also be approached head-on by simply proving the incorrectness of a program [18]. This is done by showing a negated and existentially quantified correctness formula:

$$\exists\, \textit{pre-state}.\ \neg\big(\textit{preconditions} \rightarrow \langle\, \textit{statements}\,\rangle\ \textit{postconditions}\big) \qquad (1)$$

in which *statements* represents the program in question and $\exists\, \textit{pre-state}.$ existentially quantifies all variables, class members and array components that can be read by the program. Formula (1) is true if and only if there is a pre-state in which the preconditions hold, the program fragment does not terminate, or terminates and the postconditions do not hold in the final state. With the help of meta variables (Sect. 2.3) for handling the existential quantifier, symbolic execution and the proof search strategy mechanism, KeY is quite capable to discharge such *disproving obligations* automatically.

Like the discovery of a failing test case, the ability to prove (1) reveals a program defect.[4] The program state for which this happens can be recovered by analysing the proof and extracting the values that were chosen when eliminating the quantifier in (1). Because only symbolic execution of the program is involved, this can even yield descriptions of whole classes of states, in the style of: "the program fails whenever x is greater than y." For the program `getMax()` on page 6, for instance, KeY can automatically find the counterexamples $\texttt{arr.length} = 1$, $\texttt{arr.length} = 2 \wedge \texttt{arr[0]} < \texttt{arr[1]}$ and $\texttt{arr.length} = 3 \wedge \texttt{arr[0]} < \texttt{arr[1]} \wedge \texttt{arr[0]} \leq \texttt{arr[2]}$.

Symbolic incorrectness proofs can also discover defects that are inaccessible to normal testing, for instance the divergence of programs [19]. In order to show that a program does not terminate (for a particular pre-state), it has to be proven that no terminal state is reachable. This can be done by synthesising an invariant that approximates the set of reachable states, and which excludes all terminal states.

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*, volume 4334 of *LNCS*. Springer-Verlag, 2007.
3. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings, International Symposium on Software Testing and Analysis, Roma, Italy*, pages 123–133. ACM, 2002.

[4] But it should be noted that incorrectness proofs cannot detect bugs in other components like the compiler or the runtime environment, in contrast to testing.

4. R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.
5. Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Java Series. Addison-Wesley, June 2000.
6. Y. Cheon, M. Kim, and A. Perumandla. A complete automation of unit testing for Java programs. In *Proceedings, Software Engineering Research and Practice (SERP), Las Vegas, USA*, pages 290–295. CSREA Press, 2005.
7. Y. Cheon and C. E. Rubio-Medrano. Random test data generation for Java classes annotated with JML specifications. In *Software Engineering Research and Practice*, pages 385–391, 2007.
8. B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In K. Etessami and S. K. Rajamani, editors, *Proceedings of CAV 2005*, volume 3576 of *LNCS*, pages 296–300. Springer Verlag, 2005.
9. D. Detlefs, G. Nelson, and J. Saxe. Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Labs, July 2003.
10. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In Y. Gurevich, editor, *Proceedings, Testing and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.
11. M. Giese. Incremental closure of free variable tableaux. In *Proc. Intl. Joint Conf. on Automated Reasoning, Siena, Italy*, volume 2083 of *LNCS*, pages 545–560. Springer-Verlag, 2001.
12. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
13. M. Heisel, W. Reif, and W. 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.
14. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 583, Oct. 1969.
15. D. Larsson and W. Mostowski. Specifying Java Card API in OCL. In P. H. Schmitt, editor, *OCL 2.0 Workshop at UML 2003*, volume 102C of *ENTCS*, pages 3–19. Elsevier, November 2004.
16. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual. Draft Revision 1.200*, Feb. 2007.
17. Parasoft. JTest manual, 2004. http://www.parasoft.com/jtest.
18. P. Rümmer and M. A. Shah. Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In Y. Gurevich and B. Meyer, editors, *TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers*, volume 4454 of *LNCS*, pages 41–60. Springer, 2007.
19. H. Velroyen and P. Rümmer. Non-termination checking for imperative programs. In B. Meyer, editor, *TAP 2008, Prato, Italy*, 2008. To appear. A prototypical implementation is available at `http://www.key-project.org/nonTermination/`.
20. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, Aug. 2003.