Generating Unit Tests from Formal Proofs

Christian Engel and Reiner Hähnle

Department of Computer Science and Engineering, Chalmers University of Technology
412 96 Göteborg, Sweden
engelc@ira.uka.de, reiner@chalmers.se

Abstract. We present a new automatic test generation method for JAVA CARD based on attempts at formal verification of the implementation under test (IUT). Self-contained unit tests in JUnit format are generated automatically. The advantages of the approach are: (i) it exploits the full information available in the IUT and in its formal model giving very good hybrid coverage; (ii) a non-trivial formal model of the IUT is unnecessary; (iii) it is adaptable to the skills that users may possess in formal methods.

Keywords: model-based testing, program verification, symbolic execution, test coverage, theorem proving, unit testing, white-box testing.

1 Introduction

We present a new automatic test case generation (ATCG) method for object-oriented software based on formal verification technology, accordingly called *verification-based test generation* (VBT). It combines features from white and black box test generation methods. VBT uses the full information contained in a formal specification *and* the underlying implementation under test (IUT). The main advantages over model-based test generation are: a detailed formal model of the IUT is not needed, in fact, test cases can be generated even from trivial specifications; in addition, it is possible to generate test cases that exhibit bugs contained only in the code and not in the specification. Such errors cannot reliably be detected with model-based test generation. As test generation is based on *systematic* attempts to verify the IUT against its specification, the resulting test cases satisfy rather strong *hybrid*, i.e., model-based as well as code-based coverage criteria.

Like other test generation approaches we concentrate on creating self-contained unit test cases including fixtures and test oracles. The intended application domain are safety- and security-critical JAVA and JAVA CARD programs running on small embedded devices such as smart cards or mobile phones. Unit testing is an essential technique for ensuring quality of industrial software. Writing unit tests by hand is labour intensive and leaves significant uncertainties concerning the quality of the produced tests in terms of achieved test coverage and of correctness of the test oracle relative to the specification of the tested code. To remedy this situation, various ATCG approaches have been suggested. The most common are specification- or model-based test generation (commonly referred to as black box techniques) [1,4,5,9,10,11] and white box approaches [8,26,27,28] that are based on code-driven state exploration by symbolic execution. A

B. Meyer and Y. Gurevich (Eds.): TAP 2007, LNCS 4454, pp. 169–188, 2007.
© Springer-Verlag Berlin Heidelberg 2007

detailed comparison of our method to these and to other ATCG approaches is done in Section 8.

Our own approach contains features from both white and black box techniques, but adds a new ingredient: the starting point of the VBT process is a systematic attempt, based on symbolic execution, to formally verify correctness of a given JAVA CARD program `p` relative to a precondition *pre* and postcondition *post*. In our concrete setting we use the KeY verification system [2] that provides appropriate (attempted) proofs based on symbolic execution of the target program `p`, interleaved with first-order simplification. It returns tree-shaped proof objects where the nodes can be interpreted as symbolic execution states and the branches as symbolic execution paths through `p`. The presentation in this paper is based on the implementation in the KeY system, but in principle, it would be possible to use any other JAVA verification system that works with symbolic execution, for example, the KIV system [25].

The proof object on which test case generation is based does not need to constitute a complete proof, for example, loops may have been approximated by executing them symbolically a fixed number of times. This has the advantage that the proof construction phase and, therefore, test generation is fully automatic. The information contained in a proof is used to extract test data from the path conditions that characterize certain symbolic execution paths (or all of them, depending on the desired test coverage). From the postcondition *post* test oracles are generated.

A complete functional specification of the implementation under test `p` is not required, because test generation is based on symbolic execution of the *code* `p`, while the *specification pre*, *post* is only needed to synthesize the test oracle. Meaningful test cases are obtained already for trivial specifications. For example, the precondition *pre* could just express that object references in `p` are non-null and the postcondition *post* is merely *true*. For the generation of test oracles, somewhat more extensive specifications are required, even though they can be far from complete (as will be shown below).

For specification our implementation supports the popular high-level Java Modeling Language (JML) [23] whose compatibility with JAVA syntax reduces the extent of formal methods knowledge that JAVA developers need to come up with formal specifications. For example, it enables the programmer to write the postcondition as a JAVA expression using query methods. In essence, the test oracle is then directly provided as a JAVA method.

In summary, the main advantages of the VBT methodology are: (i) it is fully automatic, but since coverage and quality of tests can be improved by more complete proofs it is adaptable to the skill of users; (ii) test generation is possible already with a trivial specification—again, since test oracles and relevance of generated test cases are improved by fuller specifications, the method is adaptable to the skill of users; (iii) rather strong hybrid code- and model-based coverage criteria are met; (iv) test generation and verification happen in a uniform framework and tool; (v) the full JAVA CARD programming language is covered.

In the next section we provide background on test coverage, JAVA CARD, formal verification, program logics, and symbolic execution. In Sect. 3 we outline a basic version of our method guided by an example. In Sect. 4 we extend it to code with unbounded loops and recursion. We also discuss when various code-based coverage

criteria are reached. In Sect. 5 we present some measures that ensure high automation of our method and in Sect. 6 we show that the tests obtained from our approach satisfy further coverage criteria [29]. In Sect. 7 we report on some experimental results, followed by related and future work in Section 8. Due to space limitations generated test cases and proofs of theorems cannot be reproduced here. On the web page `i12www.ira.uka.de/~engelc/testCases/` proofs of theorems as well as full specifications of the examples in JML and generated test cases can be found.

2 Background

Test Coverage Criteria. To make the paper self-contained we define some standard notions of test coverage [29]. Recall that in general an implementation under test (IUT) has an infinite number of execution paths, but only a finite number of branches.

Definition 1. *A formula* φ *is a* path condition *for an execution path* p *through the IUT iff* p *is executed whenever* φ *holds before the execution of the IUT.*

A feasible execution path *is an execution path that has a satisfiable path condition. A branch or statement in the IUT is called* feasible *if it is contained in at least one feasible execution path. We say a branch (a path) is* covered *by a test case iff it is executed when running the test case.*

A test set for a given IUT satisfies the feasible branch (path) coverage criterion *iff each feasible branch (path) is covered by it.*

JAVA CARD. The JAVA CARD programming language is a dialect of JAVA characterized by the absence of a number of language features (mainly floating point types, multiple threads, dynamic class loading) and the presence of others (persistent vs. transient memory, atomic transactions). Most language features, however, are available in JAVA CARD just like in JAVA. While JAVA CARD achieves unprecedented independence from the hardware platform, one can state that the gap between the behaviour of programs in desktop simulators and after actual deployment on the target hardware is a serious concern for JAVA CARD software developers. In practice, *all* JAVA CARD developer workspaces are equipped not only with emulators, but with various JAVA CARD hardware platforms, and for good reasons: in principle, it is possible to test JAVA CARD applications with the help of a simulator in a standard JAVA environment on a desktop. There are also emulators that mimic the behaviour of smart card hardware. The simulated and the actual behaviour, however, differs considerably. This is due to ambiguities [20] in the JAVA CARD language definition, but also because simulators and emulators do not implement all JAVA CARD aspects or the implementation on the device is faulty. As a consequence, even if JAVA CARD code has been formally verified it is essential to test it, because correct execution cannot be assumed.

Formal verification. Our approach to automatic test generation is based on a white box analysis of the richest possible program model: the target source code together with a formal programming language semantics. Such representations are realized in formal software verification systems. In these systems a program logic (for example, Hoare

logic or dynamic logic—see below) allows to express properties of programs which then can be formally proven from a set of logical inference rules that capture the axiomatic semantics of the target language. State-of-the-art program verification systems are able to prove security and correctness properties of industrial JAVA CARD software [2,21,25]. The implementation described in this paper is based on the verification tool KeY [2].

Dynamic Logic for JAVA CARD. In our method the target program and its specification are both modeled in a version of a dynamic logic (DL) [18] calculus called JAVA DL [2]. Dynamic logic is a program logic that generalizes Hoare logic. It can be seen as a modal logic with modalities $\langle \mathtt{p} \rangle\, \varphi$ and $[\mathtt{p}]\, \varphi$ for every program p with an arbitrary formula φ in its scope (which in turn may contain modalities). JAVA DL formulas are interpreted over first-order Kripke structures $\mathcal{K} = (\mathcal{S}, \rho)$, where $\mathcal{S}$ is a set of first-order structures including interpretations of the identifiers occurring in programs, and ρ is a function that assigns to a program p its operational semantics as a transition relation $\rho(\mathtt{p}) \subseteq \mathcal{S} \times \mathcal{S}$: if p is a legal JAVA CARD program and started in state $s \in \mathcal{S}$, then $(s, s') \in \rho(\mathtt{p})$ iff p terminates normally (i.e., not abruptly) in final state s'. The formula $\langle \mathtt{p} \rangle\, \varphi$ holds in $s \in \mathcal{S}$ iff p terminates normally and in the final state after termination φ holds. In other words, p is *totally correct* with respect to postcondition φ. Dually, $[\mathtt{p}]\, \varphi$ expresses *partial correctness*: if p terminates normally, then in the final state φ holds.

State Updates. In JAVA (as in other object-oriented programming languages), different object type variables may refer to the same object. This phenomenon, called aliasing, causes difficulties for the handling of assignments in a calculus for JAVA DL. For example, whether or not the formula $\mathtt{o1.f} \doteq \mathtt{1}$ holds after (symbolic) execution of the assignment `o2.f = 2;`, depends on whether o1 and o2 refer to the same object. Therefore, JAVA assignments cannot be symbolically executed by syntactic substitution. In the JAVA DL calculus another solution is used, based on the notion of (state) *updates*.

Definition 2. Atomic updates *are of the form* $\mathtt{loc} := \mathtt{val}$*, where* val *is a logical term without side effects and* loc *is either a program variable or a simple field access or an array access. Updates may appear in front of any formula or term, where they are surrounded by curly brackets for easy parsing. The semantics of* $\{\mathtt{loc} := \mathtt{val}\}\varphi$ *is the same as that of* $\langle \mathtt{loc=val;} \rangle\, \varphi$.

The idea with updates $\mathcal{U}$ is that during symbolic execution they represent the current computation state in which a program formula $\mathcal{U}\langle \mathtt{p} \rangle\, \varphi$ is executed. They are continously simplified during symbolic execution, but their application to modal formulas in their scope is delayed until the program has been completely executed.

Sequent Calculus. As it is usual for program logics, the axiomatization of JAVA DL is based on a sequent calculus. The central notion is that of a sequent, which is an expression of the form $\Gamma \Rightarrow \Delta$, where Γ and Δ are finite sets of formulas. The sequent $\Gamma \Rightarrow \Delta$ is valid, if and only if the conjunction of the formulas in Γ implies the disjunction of the formulas in Δ in all states of any JAVA DL-Kripke structure. The rules of a sequent calculus are denoted with schematic reasoning patterns that characterize validity of formulas occurring in the conclusion of a rule. Their general format is:

$$\frac{\Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta}$$

Soundness of the calculus requires that for each rule the validity of the sequents above the line imply the validity of the sequent below the line. There is a sequent rule for each top-level operator both for left and right sides of the sequent arrow. For example,

$$\frac{\Gamma \Rightarrow \varphi, \Delta \quad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \varphi \wedge \psi, \Delta} \text{ AND-RIGHT}$$

is a rule that characterizes conjunction on the right and is named accordingly. Here φ and ψ (Γ and Δ) are schematic variables that can be instantiated with any (set of) formula(s). Rules with an empty set of premisses are admissible. They are called *axioms* and their premiss is labelled with $*$. A typical axiom has the conclusion $\Gamma \Rightarrow t \doteq t, \Delta$.

Rules are read bottom-up: the bottom sequent is the sequent on which the rule is applied. The sequents on top are the results of the rule-application. Thus, when proving validity of a formula φ the proof starts with the sequent $\Rightarrow \varphi$ (the empty set of formulas on the left is omitted). Partial proofs in a sequent calculus take the shape of trees whose nodes are labelled with sequents. In *complete proofs* all leaves are labelled with $*$.

Symbolic Execution. The programs occurring in JAVA DL formulas are executable JAVA code. Rules for program formulas operate on sequents of the form $\Gamma \Rightarrow \mathcal{U}\langle \pi\,\mathtt{p}\,\omega \rangle\, \varphi, \Delta$, where $\mathcal{U}$ is an update containing the current state of symbolic execution and $\mathtt{p}$ is a single JAVA statement called the *first active statement*. The prefix π consists of an arbitrary number of opening braces, try-blocks and method frames (the stack trace), and ω is the whole rest of the program. Each rule for a program formula specifies how to execute symbolically one particular JAVA expression or statement, possibly with additional restrictions. *Symbolic* execution entails that locations have no concrete but symbolic values and the effect of the execution of a statement is described by logical means with symbolic values. When a loop or a recursive method call is encountered, it is in general necessary to perform induction or supply a suitable invariant.

JAVA DL extends other variants of DL used for theoretical investigations or verification purposes, because it handles such phenomena as side effects, aliasing, object types, exceptions, and finite integer types. Since deduction in the JAVA DL calculus is based on symbolic program execution and simple program transformations, it is close to a programmer's understanding of JAVA.

In a symbolic setting, the code branch that the control flow takes after evaluation of a conditional statement cannot always be determined as it depends on symbolic values. In general, a case distinction has to be introduced in the proof. One may also view this as symbolic execution branching into different execution paths. Each symbolic execution path is governed by a branch condition that is syntactically added to the left side of sequents during evaluation of conditional statements. Thus, the branching conditions accumulate in the sequent during symbolic execution and the current path condition for each execution path (including incomplete execution paths that have not yet terminated) is always contained in the leaves of proof trees during any phase of symbolic evaluation. This is illustrated in Fig. 1.

Example 1 (Rule for the **if***-statement).* When an **if** statement is symbolically executed a case distinction whether the guard is true or not has to be made. This is reflected by a split in the proof tree.

$$\frac{\Gamma, \texttt{c} \doteq TRUE \Rightarrow \langle \pi\, \texttt{\{p\}}\, \omega \rangle\, \psi, \Delta \quad \Gamma, \texttt{c} \doteq FALSE \Rightarrow \langle \pi\, \omega \rangle\, \psi, \Delta}{\Gamma \Rightarrow \langle \pi\, \textbf{if}\, \texttt{(c) \{p\};}\, \omega \rangle\, \psi, \Delta} \ \textsc{IfThenSplit}$$

The conditional **if**`(c) {p}` is the first active statement in the modality, where c is a boolean side-effect free expression. The updates before the program formulas are not explicitly written. The left premiss represents the case that the expression `c` holds, thus we find $\texttt{c} \doteq TRUE$ on the left side which becomes part of the path condition on the corresponding execution path. As `c` holds, the body of the **if**-statement is executed, therefore, the program formula $\langle \pi\, \textbf{if}\, \texttt{(c) \{p\};}\, \omega \rangle\, \psi$ is transformed into $\langle \pi\, \texttt{\{p\}}\, \omega \rangle\, \psi$, where symbolic execution continues. The right premiss represents the case that `!c` holds thus we find $\texttt{c} \doteq FALSE$ on the left side of the sequent. In this case the body of the if statement is not executed and we get the new program formula $\langle \pi\, \omega \rangle\, \psi$.

3 Overview of Verification-Based Test Generation

Verification-based testing (VBT) is motivated by the insight that a formal analysis of a specification and/or the corresponding code, as performed in a formal proof attempt, yields enough information to produce test cases. In our view a full description of a software system consists of *both, implementation and specification*. In order to detect as many errors as possible, it is essential to analyse and compare two levels of modeling.

Several ideas from other test generation methods are as well found in VBT, for example, to synthesize a test oracle from a formal specification (the postcondition) or to use reasoning technologies such as deduction, constraint solving and symbolic execution to achieve a high automation of the test generation process.

We walk through our test generation method guided by an example. It will demonstrate the automatic creation of self-contained unit tests for an implementation under test (IUT) containing only a finite number of feasible execution paths (the general case is handled in Sect. 4). In this case we obtain a test set satisfying the rather strong feasible execution path coverage criterion (Def. 1). The reason for this can be found in the soundness of the JAVA DL sequent calculus: if a certain path p with path condition φ in a code fragment c would not figure in a complete proof then also a complete proof for the invalid formula $\varphi \rightarrow \langle \texttt{c} \rangle\, false$ could be constructed which would imply that the calculus is unsound.

Example 2. Method `conditionalSwap` swaps the values of the field `value` of two objects `x` and `y` of type `NaturalNumberWrapper` provided that `x.value >= y.value`. Its behaviour is specified using JML [23] as shown below.

```
public class NaturalNumberWrapper{

    private /*@spec_public@*/ int value;

    //@ public invariant value > 0;
```

```
    /*@ public normal_behavior
      @   requires x!=null && y!=null;
      @   ensures \old(x.value) >= \old(y.value) ?
      @     (\old(x.value)==y.value && \old(y.value)==x.value) :
      @     (\old(x.value)==x.value && \old(y.value)==y.value);
      @*/
    public static void conditionalSwap(NaturalNumberWrapper x,
                                       NaturalNumberWrapper y){
        if(x.value >= y.value){
            swap(x, y);
        }
    }

    public static void swap(NaturalNumberWrapper x,
                            NaturalNumberWrapper y){
        y.value += x.value;
        x.value = y.value - x.value;
        y.value -= x.value;
    }
}
```

Clearly, there are two feasible execution paths characterized by the path condition $\texttt{x.value} \geq \texttt{y.value}$, resp., by $\texttt{x.value} < \texttt{y.value}$ that are induced by the guard of the conditional occurring in `conditionalSwap`, see also Fig. 1.

Extraction of the IUT. KeY's JML front end automatically translates [15] JML specifications to JAVA DL formulas that constitute a proof obligation (PO) for the KeY verifier. For the method `conditionalSwap` and its JML specification the PO is

$$\forall x'.\forall y'.\{\texttt{x} := x', \texttt{y} := y'\}((inv \wedge pre) \rightarrow \langle\texttt{conditionalSwap(x,y);}\rangle\Phi), \tag{1}$$

where Φ is a first-order formula representing the post condition, *inv* is the formula $\forall z.z.\texttt{value} > 0$ representing the class invariant for class `NaturalNumberWrapper` and $pre := (\texttt{x} \neq \textbf{null} \wedge \texttt{y} \neq \textbf{null})$ the precondition of `conditionalSwap` defined by the JML specification. After quantifier elimination by skolemization and pushing in updates (where $\mathcal{S}$ abbreviates $\{\texttt{x} := c_{x'}, \texttt{x} := c_{x'}\}$) we obtain:

$$\mathcal{S}(inv \wedge pre) \rightarrow \mathcal{S}\langle\texttt{conditionalSwap(x,y);}\rangle\Phi. \tag{2}$$

This formula is the root node in the partial proof tree depicted in Fig. 1 (the left part of the implication is abbreviated with Γ). From this formula we extract the IUT

$$\texttt{x = } c_{x'}\texttt{; y = } c_{y'}\texttt{; conditionalSwap(x,y);}$$

and the postcondition Φ. Later we generate a test oracle from Φ. The node in the proof tree used for extracting the IUT (2) is called *code node*.

Extraction of Path Conditions from the Proof Tree.

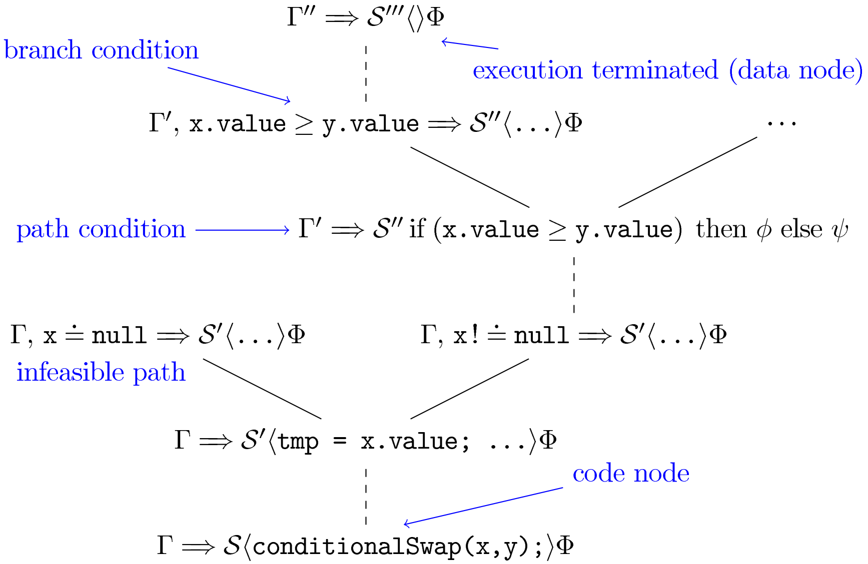

Fig. 1. Partial proof tree for the example in the text

Definition 3. *A proof tree in which each branch is either closed or ends with a leaf that contains no code fragments anymore (indicating termination of symbolic execution on that branch) is called* fully executed.

A fully executed proof tree for the PO (1) is constructed automatically by the KeY system. It is partially shown in Fig. 1. Recall from Sect. 2 that branches in the proof tree can be identified with execution paths through the IUT. Since each node contains a path condition that leads to the current point of symbolic program execution, we are interested in exactly those nodes that contain an empty program (signifying termination of symbolic execution). These nodes are the leaves of open branches in a fully executed proof tree and referred to as *data nodes* from now on. They have the form

$$\Gamma \Rightarrow \mathcal{U}\langle\rangle\,\Phi, \Delta, \tag{3}$$

where Γ and Δ are sets of first-order formulas and $\mathcal{U}$ is a sequence of updates representing the effect of the symbolic execution of the IUT on the branch of the proof tree whose path condition is, therefore, given by:

$$\bigwedge_{\gamma\in\Gamma}\gamma \wedge \bigwedge_{\delta\in\Delta}\neg\delta. \tag{4}$$

It is important to realize that closed branches where symbolic execution did not terminate need not be considered, since those branches must have been closed because of an unsatisfiable path condition and, therefore, cannot be reached. In Fig. 1, for example, the node labelled “infeasible path” originates from the null pointer check performed

each time when an attribute on an object reference is accessed, here $\mathtt{x} \doteq \mathbf{null}$. Symbolic execution of this node leads to a new proof goal of the form

$$\Gamma, c_{x'} \doteq \mathbf{null} \Rightarrow \mathcal{S}' \langle \pi\, \mathbf{throw\ \ new}\ \mathtt{NullPointerException();}\ \omega \rangle\, \Phi. \tag{5}$$

It represents the case that a null pointer exception is thrown. It can be closed immediately, because the formula $c_{x'} ! \doteq \mathbf{null}$ is contained in the precondition Γ (originating from the requires clause of the JML contract). In the fully executed proof tree we find the following data nodes:

$$c_{y'} \doteq c_{x'},\, inv_1 \Rightarrow\ \mathcal{U}_1 \langle\rangle\, \Phi,\, pre_1 \tag{6}$$

$$c_{y'}\mathtt{.value} \leq c_{x'}\mathtt{.value},\, inv_2 \Rightarrow\ \mathcal{U}_2 \langle\rangle\, \Phi,\, pre_2,\, c_{y'} \doteq c_{x'} \tag{7}$$

$$c_{y'}\mathtt{.value} \geq c_{x'}\mathtt{.value} + 1,\, inv_2 \Rightarrow\ \mathcal{U}_3 \langle\rangle\, \Phi,\, pre_2 \tag{8}$$

Here, inv_1 is $\{inv,\, c_{x'}\mathtt{.value} \geq 1\}$ and inv_2 is $\{inv,\, c_{x'}\mathtt{.value} \geq 1,\, c_{y'}.value \geq 1\}$. They are derived from the invariant of the JML specification. Formula pre_1 stands for $c_{x'} \doteq \mathbf{null}$ and pre_2 for $\{c_{x'} \doteq \mathbf{null},\, c_{y'} \doteq \mathbf{null}\}$. They stem from the precondition of the JML method contract. In (4) we defined path conditions in such a way that, in addition to branching conditions, they may contain constraints like $pre_{\{1,2\}}$ and $inv_{\{1,2\}}$ stemming from formulas present in the code node. The formulas $c_{y'}\mathtt{.value} \geq c_{x'}\mathtt{.value} + 1$ and $c_{y'}\mathtt{.value} \leq c_{x'}\mathtt{.value}$ are introduced by a case distinction performed when the **if** statement occurring in `conditionalSwap` is symbolically executed (see Fig. 1). The formula $c_{y'} \doteq c_{x'}$ occurring in data node (6) on the left and in data node (7) on the right side are introduced by another case distinction caused by an alias analysis when the assignments in `swap` are symbolically executed. This case distinction is needed for distinguishing whether `x` and `y` are referencing the same object. From the data nodes the path conditions are obtained via (4).

Generation of Integer Test Data. For creating suitable test data for each execution path we have to find first-order models of the corresponding path condition formulas. For integer types concrete interpretations are currently found by applying *Cogent* [12] or *Simplify* [14] to the formula

$$\bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \bigwedge_{\delta \in \Delta} \delta,$$

i.e., the negation of the path condition (4). If the path condition is satisfiable and the decision procedure manages to deliver a counter example for its negation the integer type test data are derived from the returned counter example. While *Simplify's* integer arithmetic is unbounded, Cogent, as a decision procedure for *C* expressions, uses bounded 32-bit arithmetic. Thus for getting meaningful results one has to restrict the arithmetic operations allowed in the specification to 32-bit Java **int** operations and the permitted **int** literals to values expressible in 32-bit **int** arithmetic (i.e. discrete values in the interval $[-2^{31}, 2^{31}-1]$). This is possible in KeY.

In contrast to *Cogent*, *Simplify* does not necessarily return a concrete counter example. In general the counter examples provided by *Simplify* have the form $\bigwedge_{\pi \in \Pi} \pi$, where π is an atomic formula of the form $p(t_1,t_2)$ with top level predicate $p \in \{<, \leq, >, \geq, \doteq\}$. If for each $p(t_1,t_2) \in \Pi$ t_1 represents a Java location and t_2 an integer literal we have

found a concrete counter example. Otherwise, *Simplify* is applied recursively to the refined formula $\neg(t_1 \doteq t_2 \wedge \bigwedge_{\pi \in \Pi} \pi)$, where t_1 and t_2 are chosen in such a way that one of the following conditions holds:

- $t_1 \leq t_2 \in \Pi$,
- $t_1 \geq t_2 \in \Pi$,
- $t_3 < t_2 \in \Pi$ with $t_1 := t_3 + 1$,
- $t_1 > t_3 \in \Pi$ with $t_2 := t_3 + 1$ or
- t_1 occurs in Π and t_2 is an arbitrary integer literal iff no inequations occur in Π and thus none of the previous conditions can be met.

The procedure is repeated until Simplify returns a concrete counter example. This relatively naive approach is sufficient in practice, because path conditions are easily satisfiable for non-pathological programs.

Generation of Reference Type Test Data. As a first step the set R of all terms that occur in the path condition and whose type is non-primitive is grouped into equivalence classes $R_{/\sim}$ where $a \sim b$ iff $\bigwedge_{\gamma \in \Gamma} \gamma \wedge \bigwedge_{\delta \in \Delta} \neg\delta \models a \doteq b$ (In JAVA DL $a \doteq b$ means object identity). For each of these equivalence classes C test data are chosen to be either (i) **null** iff **null** $\in C$ or (ii) an object of type t where t is the minimal static type of the terms $t \in C$ if t is not an array type or (iii) an array of length n otherwise, where n is the concrete value found for a term $a.length$ with $a \in C$ during the integer type test data generation phase. If no such term a exists an arbitrary value n is chosen.

Test Oracle. The test oracle is generated by transforming the postcondition Φ of the IUT into loops iterating over boolean JAVA expressions. Quantified subformulas are only allowed to occur in Φ if they match one of the following patterns

$$\forall x.\,(a \prec_1 x \wedge x \prec_2 b \rightarrow \Psi) \qquad\qquad \exists x.\,(a \prec_1 x \wedge x \prec_2 b \wedge \Psi),$$

where x has an integer type and $\prec_1, \prec_2 \in \{<, \leq\}$. This restriction essentially confines postconditions within the *guarded fragment* of first-order logic [19]. Guarded quantified formulas can be evaluated by a loop iterating over the range given by the bounded guard predicate $a \prec_1 x \wedge x \prec_2 b$. The postcondition of Example 2 is quantifier-free and can be trivially turned into boolean JAVA expression.

Example 3 (Sort). The following approximate specification of a sorting algorithm has a post condition containing quantified expressions obeying the above restrictions:

```
/*@ public normal_behavior
  @  ensures a!=null ==>
  @          (\forall int i; 0<=i && i<a.length-1; a[i]<=a[i+1])
  @           &&
  @          (\forall int i; 0<=i && i<a.length;
  @           (\exists int j; 0<=j && j<a.length; \old(a[i])==a[j])
  @          );
  @*/
public static void sort(int[] a) { ... }
```

From the first quantified JML subexpression

```
\forall int i; 0<=i && i<a.length-1; a[i]<=a[i+1]
```

the following JAVA oracle is computed:

```
boolean result = true;
for (int _i0 = (0); _i0 <= ((-2) + _old2_a.length); _i0++) {
  result = result && TestBubbleSort0.subformula1(_i0,_old2_a,buffer);
}
buffer.append(...);
return result;
```

Here, `TestBubbleSort0.subformula1` is a wrapper method for the oracle created from the subexpression `a[i]<=a[i+1]` (`a` and `i` are renamed to `_old2_a` and `_i0`). It has parameters `_i0, _old2_a` needed to evaluate `a[i]<=a[i+1]` and the `StringBuffer` variable `buffer` is used for logging results of the evaluation of subexpressions. This provides valuable information when a test run fails.

In the case of the trivial postcondition *true* the test oracle always succeeds. In this case the resulting unit tests can only fail if the execution hangs or throws an uncaught exception. Even in this case we obtain meaningful and important tests, because uncaught exceptions are the cause of many serious errors.

Generation of Unit Tests. The generated tests are in JUnit (`www.junit.org`) format. For every feasible execution path found a separate test method is created. In this way erroneous execution paths can easily be identified after failed tests.

For the example above three test methods are created corresponding to the path conditions of data nodes (6)–(8). Each test method contains a different test case for each first-order model obtained from the path condition.

The test case generated from (6) reports a failure when executed. Analysis of the reason why the postcondition is not satisfied exhibits a bug: whenever the arguments `x` and `y` of `swap` point to the same object the result state is $\texttt{x.value} \doteq \texttt{y.value} \doteq 0$ irrespective of the initial value. The branching condition $c_{y'} \doteq c_{x'}$ in (6) covers exactly this case.

This kind of bug is not easy to discover with model-based test generation, because the case distinction on whether the arguments are identical objects is an implementation issue and does not occur in the specification naturally.

Modification of the Implementation under Test. In contrast to some other white box test generation methods [27,28] the program logic JAVA DL is capable of handling symbolic reference type values. In order to provide a fixture for reference type test data a modified version of the IUT is included in the generated unit test. The modifications consist of supplying default constructors and *get* and *set* methods for **private** and **protected** fields so that the test fixture can create objects with the properties determined by the found models of the path conditions. The new methods are uniquely named and do not change the semantics of the IUT. This means in the case of the example that methods for accessing and modifying the field `value` are added to the class `NaturalNumberWrapper`. In general, also **final** modifiers are removed from final instance fields and from final static fields that are not initialized with a compile time constant.

4 Unbounded Number of Execution Paths and Test Coverage

An IUT containing loops or recursive method calls may give rise to an infinite number of feasible execution paths if its specification imposes no upper bound on the number of iterations of the loop or the recursion depth. In this case it is obviously not possible to find a finite proof tree that covers every feasible execution path. We can see two approaches to deal with this situation:

1. Unwind the loop (unfold the recursive method call) a fixed number of times. This strategy creates only a partial proof. One uses only those execution paths on which symbolic execution has terminated.
2. Replace non-linear constructs such as loops and method calls by suitable specifications, i.e., an invariant in the case of loops and a contract in the case of method calls. This allows to obtain complete proof trees.

The first approach does not try to produce fully executed proofs. Proof attempts are simply stopped after a given resource bound has been reached and the information obtained so far is exploited. The second approach formally constructs a fully executed and possibly even complete proof tree, but this proof tree either has gaps (where a contract is used) or relies on an invariant supplied by the user. Which of the two approaches is appropriate depends on the specific IUT and the targeted coverage as pointed out below.

4.1 Partial Proofs

To unwind a loop of the form **while**`(b) {p};` means to syntactically replace it with

`l1:` **if**`(b) { l2: p';` **while**`(b) {p} };`,

where `p'` is obtained from `p` by replacing the occurring *break* and *continue* statements in an appropriate way by local jumps to the fresh labels `l1` and `l2`.

By unwinding we may successively explore the potentially unlimited number of terminating execution paths that lead through the loop statement. Since every feasible branch contained in the loop body is taken[1] on some finite execution path we can obtain branch coverage if we only unwind the loop often enough. In practice we can usually not guarantee this except for the case that for every branch in the IUT a containing feasible execution path has been found and thus every branch is feasible. However, this can in general not be assumed since for realistic programs with array or attribute accesses branches containing raised `NullPointer`- or `ArrayIndexOutOfBounds`-exceptions are usually infeasible if the program is correct, and there is no way to determine which branches are feasible by means of unwinding alone.

The advantage of this approach is that it is highly automatic and requires no additional input such as loop invariants. It also yields high code coverage in most cases, because it turns out that very often all feasible execution paths through the loop are already feasible in the first iteration of the loop and, hence, can be discovered by unwinding the loop merely once. Similar considerations as for unwinding of loops can be made for the symbolic execution of method bodies of recursive method calls.

[1] Otherwise the branch would not be feasible by definition. JAVA does not allow infinite loops, so each branch must occur on at least one finite execution path.

4.2 Complete Proofs

By supplying suitable loop invariants it is possible to find complete proofs even for code that contains unbounded loops. In this case one obtains branch coverage for the resulting test cases provided that (i) every loop is symbolically executed under its invariant and (ii) symbolic execution terminates on every branch of the proof tree.

In order to understand why this is the case, let us first look at the invariant rule. We do not use the standard invariant rule, but one that has been optimized for usage in imperative programming languages [3]. In the rule below I represents the invariant. The point where the rule deviates from the usual invariant rules is the update set $\mathcal{V}$. It represents all locations that can possibly be changed in the loop body `q`, the so-called *modifier set*, by assigning fresh constants to all critical locations. The first premiss states as usual that the invariant holds in the current state $\mathcal{U}$. In the standard rule the second premiss (invariance property) must be shown for arbitrary states which often requires to strengthen invariants. It turns out to be sound to show the invariance property in the state $\mathcal{UV}$ which contains all locations from $\mathcal{U}$ that are not modified in the loop body. This is exploited in the rule below. The guard `b` is free of side effects.

$$\frac{\Gamma \Rightarrow \mathcal{U}I, \Delta \quad \Gamma, \mathcal{UV}(I \wedge \texttt{b}) \Rightarrow \mathcal{UV}[\texttt{q}]I, \Delta \quad \Gamma, \mathcal{UV}(I \wedge \neg\texttt{b}) \Rightarrow \mathcal{UV}[\pi\,\omega]\Phi, \Delta}{\Gamma \Rightarrow \mathcal{U}[\pi\,\textbf{while}\texttt{(b) \{q\}}\,\omega]\Phi, \Delta}$$

We will argue that if we apply the loop invariant rule to every subgoal containing the formula $[\pi\,\textbf{while}\texttt{(b) \{q\}}\,\omega]\Phi$ (that is, in every partial execution path reaching the loop statement **while**`(b) {q}`) we can achieve full feasible branch coverage of both `q` and of $\pi\,\omega$, hence of the whole loop.

Let p be the symbolic execution path corresponding to the proof branch from the root to node $\Gamma \Rightarrow \mathcal{U}[\pi\,\textbf{while}\texttt{(b)\{q\}}\,\omega]\Phi, \Delta$ with path condition φ_p obtained with (4). The subgoal from the leftmost premiss in the invariant rule is irrelevant for finding execution paths since no further symbolic execution takes place. Let *inv* be the second subgoal obtained from the middle premiss. It is valid if the invariant is preserved by the execution of the loop body. All code branches occurring in the body `q` that are occurring on any feasible execution path, of which p is a prefix and whose path condition implies φ_p are also feasible when the symbolic execution of `q` starts in the state defined by the node *inv* and are thus contained in the proof subtree starting with *inv*. This is owed to the soundness of the applied loop invariant rule [3]. If there were a feasible branch `br` in `q` that is not explored by symbolic execution of `q`, then the loop invariant rule would not be sound, because the invariant is possibly not preserved by some feasible execution path through `q` that contains `br`. Thus the proof subtree starting with node *inv* covers all branches in `q` that are feasible under the precondition φ_p. The subgoal *post* from the rightmost premiss in the invariant rule represents the situation after the loop has been executed and it contains the code $\pi\,\omega$. It can be argued in a similar way.

The usage of loop invariants in symbolic execution not only ensures branch coverage, but it is also more efficient than finite unwinding, because typically less code is symbolically executed. Nevertheless, even a symbolic execution tree covering all feasible branches is not always sufficient to generate tests that satisfy the feasible branch coverage criterion. The problem are the fresh constants introduced in updates $\mathcal{V}$ of the invariant rule representing the new values of the locations in the modifier set of

the loop. These constants might become part of branching conditions and, hence, path conditions. If branching conditions containing these new constants cannot be expressed with the help of terms whose values were already known in the prestate of the IUT, that is with the help of terms occurring in the code node, it becomes impossible to tell how such conditions evaluate during a run with the chosen test data and which of the associated branches is therefore covered by the test case.

Example 4. To illustrate the effect of loop unwinding during exploration of execution paths by symbolic execution we look at an implementation of the bubble sort algorithm that was specified in Example 3.

```
1  public static void sort(int[] a) {
2      if (a == null) { return; }
3      boolean sorted = false;
4      int help;
5      while (!sorted) {
6          sorted = true;
7          for (int i = 0; i < (a.length - 1); i++) {
8              if (a[i] > a[i + 1]) {
9                  help = a[i];
10                 a[i] = a[i + 1];
11                 a[i + 1] = help;
12                 sorted = false;
13 }}}}
```

We list the case distinctions that occur and the path conditions that are obtained during the symbolic execution of this code. The evaluation of the conditional statement in line 2 leads to the first case distinction.

- $a \doteq null$**:** Symbolic execution terminates after the execution of the **return** statement leading to path condition $a \doteq null$.
- $a\,! \doteq null$**:** Symbolic execution continues in line 3 and $a\,! \doteq null$ is added to the left-hand side of the sequent, i.e., to the current path condition. The next case distinction is encountered when the **while** loop is reached whose symbolic execution by unwinding needs to distinguish whether `sorted` $\doteq TRUE$ holds.
  - `sorted` $\doteq TRUE$**:** Since `sorted` has been initialized with **false** this branch leads to an infeasible path condition and it can be closed immediately.
  - `sorted` $\doteq FALSE$**:** Since `sorted` has the value **false** at this point of the program execution this branch condition is equivalent to *true* and thus does not change the current path condition (still being $a\,! \doteq null$). When executing the first iteration of the **for** loop, which is unwound in the same manner as the **while** loop, a case distinction on the expression `i < (a.length - 1)` is made. Whether this expression can be evaluated without raising an exception depends on whether $a \doteq null$ holds.
    - $a \doteq null$**:** The path condition is $a\,! \doteq null$. This code branch is infeasible and the corresponding branch in the proof tree closable.
    - $a\,! \doteq null$**:** This branch condition is implied by the path condition. The evaluation of the guard `0 < (a.length - 1)` terminates without raising an exception, but gives rise to a further case distinction:

$0 \geq (a.length - 1)$**:** The for loop is not executed. Since the guard of the **while** loop does not hold in its next iteration, symbolic execution terminates on this proof branch without introducing a new branch condition. The path condition obtained is $a\,! \doteq null \wedge 0 \geq (a.length - 1)$ from which $a\,! \doteq null \wedge (0 \doteq a.length \vee 1 \doteq a.length)$ can be derived, because the length of an array cannot be negative (this knowledge is provided by JAVA DL calculus rules).

$0 < (a.length - 1)$**:** The path condition is now $a\,! \doteq null \wedge 1 < a.length$. The execution of the first iteration of the **for** loop starts which makes a case distinction on the guard of the conditional in line 8 necessary.

The exploration of execution paths through the outer and inner loop can be continued for arbitrarily many loop iterations depending on the desired coverage or the number of desired test cases. Path conditions are continously simplified during this process, enabling us to avoid symbolic execution of infeasible paths by closing the corresponding proof branch. This is all done fully automatically.

Approximating Method Calls. For the purpose of generating unit tests it is often not desirable to take into account the implementation of all methods called in the IUT. Using modifier sets (and the method's postcondition) one can approximate symbolic execution of a method call. The idea is the same as for invariants and the above arguments apply. Of course, branch coverage is not obtained for the method body then.

5 Increasing Automation

Pruning of the Proof Tree. The subtrees below data nodes in proofs have no significance for the creation of unit tests, because they contain no symbolic execution steps. Thus, when the verification system is run with the purpose of test case generation, we prune any proof steps below data nodes. This prevents proof trees from becoming closed, but increases efficiency. The same applies to other nodes containing no code fragments such as the subgoal from the leftmost premiss of the invariant rule.

Obviously, the pruned part of a proof tree might not have been closable. It is easy, for example, to specify a too strong invariant that is preserved by the loop body, but simply does not hold at the beginning of the loop. This is checked in the first premiss of the invariant rule and if that part of the proof is not explored, then the application of the invariant rule simply becomes unsound. For test case generation this means that we might lose coverage of those branches that are feasible under the given precondition but not under the assumed invariant. We found that in practice this happens rarely and it is outweighed by the advantage of improved automation and speed. If branch coverage is important, the user can enforce full exploration of trees.

Automatic Instantiation of Quantifiers. In order to prove subgoals that contain quantified formulas it is in general necessary to provide suitable terms for instantiation of quantifiers. Owing to the undecidability of first-order logic, it is not possible to restrict these instances in a finite way. First-order quantifiers with variables ranging over the

integers are instantiated during proof search by external theorem provers such as Simplify [14]. This leaves first-order quantifiers over object reference types. It would do to ask the user to instantiate them interactively, but we found that the following brute force method works well in practice: object type quantifiers occurring in open proof goals are automatically instantiated with all symbolic object references that occurred so far during symbolic execution of the IUT and that are known to be not **null**.

6 Additional Coverage Criteria

As pointed out in Section 4, complete proof trees satisfy the feasible path coverage criterion if they are constructed by finite unwinding of all feasible execution paths and neither loop invariant rules nor approximation of methods are used. This holds even for incomplete proof trees constructed in this manner, where each open branch contains a data node indicating complete symbolic evaluation of every feasible execution path. In addition, such proof trees meet a variant of the multiple condition coverage (MCC) criterion [29] of the precondition.

Definition 4 (Minimal Partial Interpretation). *A* partial interpretation *is a mapping s from first-order formulas that contain no unbound variables into $\{true, false, \perp\}$ satisfying $s(a \wedge b) = \min(s(a), s(b))$ and $s(a \vee b) = \max(s(a), s(b))$ under the total order $false < \perp < true$ as well as $s(\neg \perp) = \perp$.*

Let $\Phi[a_1, \ldots, a_n]$ be a first-order formula, where $a_1, \ldots, a_n$ are exactly those atomic or quantified subformulas in Φ that contain no unbound variables. We call a partial interpretation s minimal *relatively to $\Phi[a_1, \ldots, a_n]$ if the following conditions hold:*

- $s(\Phi[a_1, \ldots, a_n]) = true$ *or* $s(\Phi[a_1, \ldots, a_n]) = false$
- $s_i(\Phi[a_1, \ldots, a_n]) = \perp$ *for all* $1 \leq i \leq n$ *such that* $s(a_i) \neq \perp$*, where*

$$s_i(q) = \begin{cases} s(q), & if\ q \neq a_i \\ \perp, & if\ q = a_i \end{cases}.$$

The idea behind minimal partial interpretations is that they fix the interpretation of just enough subformulas of Φ in order to determine its truth value. In order to cover all possible interpretations of a first-order formula it is, therefore, sufficient to cover merely those combinations of subformulas that are fixed by at least one of its minimal partial interpretations. Since we base our variant of multiple condition coverage on minimal partial interpretations (instead of complete interpretations) it results in less test cases while still ensuring full logical coverage of a condition.

Definition 5 (MCC). *Let $\Phi[a_1, \ldots, a_n]$ be the precondition of the IUT in a proof tree T. We say T meets the MCC_p (MCC_b) criterion iff it contains for every minimal interpretation s such that $s(\Phi[a_1, \ldots, a_n]) = true$ every execution path (branch) that is feasible under the precondition*

$$\bigwedge_{a:\, s(a)=true \text{ and } a \in \{a_1,\ldots,a_n\}} a \quad \wedge \bigwedge_{b:\, s(b)=false \text{ and } b \in \{a_1,\ldots,a_n\}} \neg b.$$

Theorem 1. *Test cases generated from complete proofs satisfy the* MCC_b *criterion implying full feasible branch coverage. If, in addition, proofs have been constructed without using loop invariant rules then test cases satisfy* MCC_p *which implies full feasible path coverage.*

The proof is by a straightforward induction over the syntactic structure of the precondition of the IUT. It is contained in the long version of this paper.

As explained in Sect. 4.2, whether a test with the same coverage as the proof tree actually can be constructed depends on the concrete form of invariants and contracts which may introduce fresh constants from modifier sets in the path conditions.

7 Evaluation

In order to evaluate our approach we first injected a number of typical errors into some standard algorithms: the `median` of three integers, the `insert` method of binary search trees (BST), a shift-add multiplier, and bubblesort. In each case we were able to detect the bugs with our automatically generated test cases. The specifications of `insert` and `sort` were incomplete and would be easy to create for a non-expert. The results are summarized in the table below (BC/PC = branch/path coverage obtained):

Method	Specification	Proof	BC	PC	covered paths
`conditionalSwap`	precise	no	yes	yes	3
`median`	precise	yes	yes	yes	6
BST, `insert`	lightweight	no	yes	∞	65
Shift-add multiplier	lightweight	no	yes	no	16
Bubblesort, `sort`	approximate	no	yes	∞	42
dto., fixed length (4)	approximate	yes	yes	yes	24

We also briefly compared our results with two model-based test generation tools (unfortunately, no code-based test generation tools were made available): ESC/Java2 [11] and UTJML [9]. None of the two tools is able to detect all bugs. This is not surprising, because none of them satisfies code-based coverage criteria. ESC/Java2 produces occasional spurious warnings and UTJML, which is in an early development stage, cannot cope with more complex methods such as `sort`. Details on the comparison are in [16].

Finally, we started to evaluate our method with an industrial application. The smart cart vendor association GlobalPlatform (`www.globalplatform.org`) provides a hardware-, operating system-, and vendor-neutral card specification [17] for JAVA CARD applications. An implementation for this specification is currently being made by IBM Deutschland Entwicklung GmbH. In order to validate vendor-specific implementations against a reference it is necessary to provide test cases with good code coverage. Based on the card specification [17] we wrote a lightweight JML specification for a part of the card life cycle management and used our tool to automatically create test cases for the `process` method of the applet and for `setAppletLifeCycle`. The method calls to the JAVA CARD API were approximated with a JML-based specification provided by W. Mostowski at `www.cs.ru.nl/~woj/software/software.html`.

The methods do not contain loops or recursive calls, so we could achieve execution path coverage (modulo JAVA CARD API calls). We produced several dozen test cases which are able to detect a number of typical coding and specification errors.

8 Conclusion, Related and Future Work

We presented a new method for automatic test case generation based on possibly incomplete, but automated attempts at formal verification of the IUT. We are able to generate self-contained unit tests in JUnit format. The implementation is based on the verification system KeY [2] and supports the JAVA CARD programming language. The approach exploits the full information available in the IUT and it is adaptable to the formal methods skill of users. In particular, a detailed formal specification of the IUT is not required. Depending on the completeness of the underlying proof attempts the method guarantees strong hybrid coverage criteria.

Related Work. The most common ATCG methodolology is specification- or model-based test generation [1,4,5,9,10,11]. Here, test cases are generated from a formal specification or model of the IUT which itself is not required or taken into account. Consistency of the test oracle with the specification is guaranteed. The drawback is that the information contained in the IUT is not analysed, therefore, no code coverage guarantees can be given. Test cases such as the one that exhibited an implementation error at the end of Section 3 are easy to miss in model-based approaches. Another problem is that a detailed formal model of the underlying system is required in order to create relevant test cases. Such models often do not exist or are too expensive to create.

More recently, white box ATCG approaches appeared [8,26,27,28] that are based on code-driven state exploration by symbolic execution. Often, they support only a limited subset of the target language features. Symbolic execution performed by the relatively advanced system Symstra [28], for instance, does not yet feature symbolic values that have a reference type. Closest among this family of ATCG approaches to ours regarding scope and performance is [26] where, however, verification cannot be combined with testing and the target language is restricted to CIL bytecode.

A different starting point is used in the systems TestEra and Korat [6,22], where systematically all non-isomorhpic inputs up to a fixed bound are generated that pass a feasibility filter based on method preconditions. A uniform framework for verification and testing has been formalised in HOL/Isabelle for a toy target language in [7], but the test generation process is not automatic. Independently of the present work, a very similar method than ours has been developed [24] based on the Bogor verification tool. This is very recent work and yet unpublished, so a detailed comparison has to wait.

Future Work. We obtained promising results on non-trivial programs but a more thorough evaluation and comparison to other automatic test generation methods is required, in particular, to model-based [1,10] and state exploration-based [27,28] approaches. We also plan to generate comprehensive test cases for a GlobalPlatform reference implementation (Section 7) in collaboration with IBM and the GlobalPlatform Association.

The syntactic form of postconditions is currently restricted to first-order formulas with finite guards in order to achieve full automation when computing test oracles. Using advanced first-order theorem proving technology, this can probably be generalized.

Incomplete proofs constructed by finite unwinding of unbounded loops to a fixed bound are not guaranteed to satisfy feasible branch coverage, however, as stated in Sect. 4.2, the obtained path conditions are easier to turn into test cases as in complete

proofs that involve loop invariants, due to the absence of fresh constants related to modifier sets. It would be interesting to combine the information from both approaches.

As stated in Section 4.1, by *sufficient* finite unwinding it is always possible to obtain feasible code branch coverage of the generated test data, because each feasible code branch is executed after a finite number of execution steps. Even though it is not possible to compute the number of unwinding steps uniformly for each program, one could implement an incomplete check whether a *given* proof tree enjoys branch coverage by relating the statements in feasible paths of the proof tree to code branches. As argued in Section 4.1, branch coverage tends to happen early, so this would be a useful test.

In order to approximate execution path coverage, arguably a data-driven approach to unwinding is more useful than the naive code-driven one we are currently using. Data-driven unwinding has been realized in Kiasan [13], where it is called k-bounding.

Acknowledgments. We thank Klaus Peter Gungl from IBM Deutschland Entwicklung GmbH for letting us have the source code of their GP Card Spec implementation.

References

1. Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Vacelet, N., Utting, M.: BZ-TT: A tool-set for test generation from Z and B using contraint logic programming. In: Hierons, R., Jerron, T. (eds.) Formal Approaches to Testing of Software, FATES 2002 workshop of CONCUR'02, August 2002, pp. 105–120. INRIA Report (2002)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Beckert, B., Schlager, S., Schmitt, P.H.: An improved rule for while loops in deductive program verification. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 315–329. Springer, Heidelberg (2005)
4. Bouquet, F., Dadeau, F., Legeard, B., Utting, M.: JML-Testing-Tools: a Symbolic Animator for JML Specifications using CLP. In: Halbwachs, N., Zuck, L. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 551–556. Springer, Heidelberg (2005)
5. Bourdonov, I.B., Kossatchev, A., Kuliamin, V.V., Petrenko, A.: UnitesK test suite architecture. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 77–88. Springer, Heidelberg (2002)
6. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: Frankl, P.G. (ed.) Proc. ACM Intl. Symp. Software Testing and Analysis, July 2002. Software Engineering Notes, vol. 27, 4, pp. 123–133. ACM Press, New York (2002)
7. Brucker, A.D., Wolff, B.: Interactive testing with HOL-TestGen. In: Grieskamp, W., Weise, C. (eds.) FATES 2005. LNCS, vol. 3997, pp. 87–102. Springer, Heidelberg (2006)
8. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Testing concurrent object-oriented systems with Spec Explorer. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 542–547. Springer, Heidelberg (2005)
9. Cheon, Y., Kim, M., Perumandla, A.: A complete automation of unit testing for Java programs. In: Arabnia, H.R., Reza, H. (eds.) Proc. Intl. Conf. on Software Engineering Research and Practice, Las Vegas, USA, vol. 1, pp. 290–295. CSREA Press (2005)
10. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 231–255. Springer, Heidelberg (2002)

11. Cok, D.R., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
12. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 296–300. Springer, Heidelberg (2005)
13. Deng, X., Lee, J., Robby,: Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In: Proc. 21st IEEE/ASM Intl. Conference on Automated Software Engineering, Tokyo, Japan, pp. 157–166. IEEE Computer Society Press, Los Alamitos (2006)
14. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
15. Engel, C.: A Translation from JML to JavaDL. Studienarbeit, University of Karlsruhe (2005)
16. Engel, C.: Verification based test case generation. Master's thesis, Department of Computer Science, University of Karlsruhe (August 2006)
17. GlobalPlatform, Foster City, USA. GlobalPlatform Card Specification, version 2.2 edn. (March 2006)
18. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
19. Hladik, J.: Implementation and optimization of a tableau algorithm for the guarded fragment. In: Egly, U., Fermüller, C.G. (eds.) TABLEAUX 2002. LNCS (LNAI), vol. 2381, pp. 145–159. Springer, Heidelberg (2002)
20. Hubbers, E., Poll, E.: Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. Dept. of Computer Science NIII-R0438, Radboud University Nijmegen (2004)
21. Jacobs, B., Marché, C., Rauch, N.: Formal verification of a commercial smart card applet with multiple tools. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 241–257. Springer, Heidelberg (2004)
22. Khurshid, S., Marinov, D.: TestEra: Specification-based testing of Java programs using SAT. Automated Software Engineering 11(4), 403–434 (2004)
23. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual, Draft revision 1.193 (May 2006)
24. Robby: Bogor/Kiasan: Combining symbolic execution, model checking, and theorem proving. Presentation at European Science Foundation Exploratory Workshop on Challenges in Program Verification, University of Nijmegen (October 2006)
25. Stenzel, K.: Verification of Java Card Programs. PhD thesis, Fakultät für angewandte Informatik, University of Augsburg (2005)
26. Tillmann, N., Schulte, W.: Parameterized unit tests. In: Wermelinger, M., Gall, H. (eds.) Proc. 10th European Software Engineering Conference/13th ACM Intl. Symp. on Found. of Software Engineering, Lisbon, Portugal, pp. 253–262. ACM Press, New York (2005)
27. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, pp. 97–107. ACM Press, New York, USA (2004)
28. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 365–381. Springer, Heidelberg (2005)
29. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. 29(4), 366–427 (1997)