White-box Testing by Combining Deduction-based Specification Extraction and Black-box Testing

Bernhard Beckert and Christoph Gladisch

University of Koblenz-Landau, Dept. of Computer Science
`beckert@uni-koblenz.de`, `gladisch@uni-koblenz.de`

Abstract. We propose to use deductive program verification systems to generate specifications for given programs and to then use these specifications as input for black-box testing tools. In this way, (1) the black-box testing method can make use of information about the program's structure that is contained in the specification, and (2) we get a separation of concerns and a clear interface between program analysis on the one hand and test-case generation and execution on the other hand, which allows the combination of a wide range of tools.
The method for specification extraction using a program verification calculus described in this paper has been successfully implemented in the KeY program verification system.

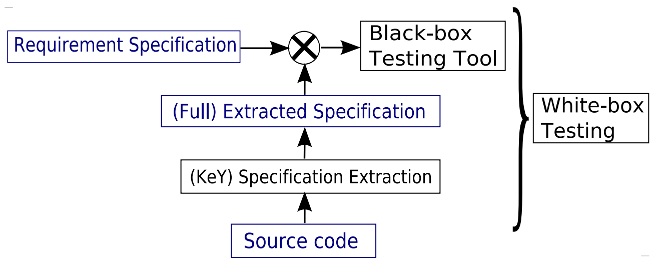
1 Introduction

Overview. We propose to use deductive program verification systems to generate specifications for given programs and to then use these specifications as input for black-box testing tools (e.g. [17, 13, 14]). Thus, (1) the black-box testing method can make use of information about the program's structure that is contained in the specification, and (2) we separate concerns and get a clear interface between program analysis on the one hand and test-case generation and execution on the other hand, which allows the combination of a wide range of tools.

To achieve goal (1), the structure of the extracted specification must reflect the structure of the analysed program. That is easy to achieve using the symbolic execution rules that are an inherit part of verification calculi while excluding simplification rules that would replace parts of the resulting specification by (simpler) logically equivalent formulae with less structural information.

Using deductive techniques, it is easy to fine-tune the specification generation process, yielding different levels of testing coverage. More scalability of quality assurance in the software development process can be achieved.

Background. The work reported in this paper has been carried out as part of the KeY project [1, 3] (`www.key-project.org`). The goal of this project is to develop a tool supporting formal specification and verification of Java Card programs within a commercial platform for UML-based software development.



Requirement Specification
Black-box Testing Tool
(Full) Extracted Specification
White-box Testing
(KeY) Specification Extraction
Source code

Fig. 1. Overview of our approach.

The method for specification extraction using a program verification calculus described in this paper has been successfully implemented in KeY. In the following, we use the KeY verification system, its verification calculus (dynamic logic), and target language (Java Card) to describe our approach. However, the ideas we present in this paper are independent of particular programming languages and verification calculi, and are easily adapted to other tools.

Another method combining testing and verification has been implemented in KeY that uses similar techniques but is different from our method in that it does not use specification generation as an intermediate step [8].

2 The General Approach

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 only either one of the two tasks: verification and testing. They do not offer interfaces providing information that could be used for the other task.

The solution that we propose (see Figure 1), is to use *specifications* that are extracted from programs for interfacing. Generated specifications are in the middle between program analysis and deductive verification (using symbolic execution) on one side and test-case generation on the other side. On both sides, there are tools that can produce specifications resp. take them as input for test-case generation (tools that do not immediately offer required interface can be extended with little effort).

As explained in detail in the following section, deductive verification mechanism that use rules for symbolically executing programs can be used for specification generation. These derived specifications can be weak (i.e., partial) or strong (i.e., complete) depending on the methods used and the complexity of the program. Derivation of a strong—or even the strongest—specification can be done automatically in some cases but may require user interaction (if a complicated invariant has to be provided). For testing purposes, the strongest specification is usually not required, so that automatic specification generation is sufficient.

The extracted specification consists of pre- and post-conditions expressed in classical first-order or higher-order logic. Since that is the basis for virtually all specification languages—such as the Object Constraint Language (OCL), Z, the Java Modeling Language (JML), or Spec#—simple syntactic changes are sufficient to generate the appropriate input for a particular testing tool.

When the structure-preserving extraction of a specification is combined with a black-box testing tool that analyses the specification's structure, the result is effectively a white-box testing method (as illustrated by Figure 1). The black-box testing tool generates test cases for every pair of pre- and post-conditions in the specification and, thus, generates tests for every execution path that has been obtained by symbolically executing the program. 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.

3 A Deductive Program Verification Calculus

Dynamic Logic. The program logic we consider in this paper is an instance of dynamic logic [11]. This instance, called Java Card DL, is the logical basis of the KeY system's software verification component [2]. Dynamic logic is a multi-modal logic with a modality $[p]$ for every program p of the considered programming language. The formula $[p]\phi$ expresses that, if the program p terminates in a state s, then ϕ holds in s. A formula $\psi \rightarrow [p]\phi$ expresses that, for every state s_1 satisfying pre-condition ψ, if a run of the program p starting in s_1 terminates in s_2, then the post-condition ϕ holds in s_2. For deterministic programs, there is exactly one such world s_2 (if p terminates) or there is no such world (if p does not terminate). The formula $\psi \rightarrow \langle p \rangle \phi$ is thus equivalent to the Hoare triple $\{\psi\}p\{\phi\}$. In contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators.

State Updates. We allow *updates* of the form $\{x := t\}$ resp. $\{o.a := t\}$ to be attached to formulas, where x is a program variable, o is a term denoting an object with attribute a, and t is a term. The semantics of an update is that the formula that it is attached to is to be evaluated after changing the state accordingly, i.e., $\{x := t\}\phi$ has the same semantics as $\langle \texttt{x = t;} \rangle \phi$. We also allow parallel updates of the form $\{u_1 \,||\, u_2\}$. Updates can be seen as a language for describing program transitions. The KeY system has a powerful update simplification mechanism that transforms sequences of updates into a single parallel update.

Program Verification by Symbolic Execution. The Java Card DL calculus used for program verification in the KeY system is a sequent calculus that works by reducing the question of a formula's validity to the question of the validity of several simpler formulae. Since Java Card DL formulae contain programs, the Java Card DL calculus has rules that reduce the meaning of programs to the meaning of simpler programs, which corresponds to a *symbolic execution* [12]. For example, to find out whether the sequent ("$\Longrightarrow$" is the sequent symbol)

$$\Longrightarrow \langle \texttt{o.next.prev=o;} \rangle \texttt{o.next.prev} \doteq \texttt{o}$$

is valid, we symbolically execute the Java code in the diamond modality. At first, the calculus rules transform it into an equivalent but longer (albeit in a sense simpler) sequence of statements:

$$\Longrightarrow \langle \texttt{ListEl v; v=o.next; v.prev=o;} \rangle \texttt{o.next.prev} \doteq \texttt{o} \ .$$

This way, we have reduced the reasoning about a complex expression to reasoning about several simpler expressions (unfolding).

Now, when analysing the first of the simpler assignments (after removing the variable declaration), one has to consider the possibility that evaluating the expression `o.next` may produce a side effect if `o` is `null` (in that case an exception is thrown). However, it is not possible to unfold `o.next` any further. Something else has to be done, namely a case distinction. This results in the following two new goals:

$$\neg(\texttt{o} \doteq \texttt{null}) \Longrightarrow \{\texttt{v} := \texttt{o.next}\}\langle \texttt{v.prev=o;} \rangle \texttt{o.next.prev} \doteq \texttt{o}$$
$$\texttt{o} \doteq \texttt{null} \Longrightarrow \langle \texttt{throw new NullPointerException();} \rangle \texttt{o.next.prev} \doteq \texttt{o}$$

Thus, we can state the essence of symbolic execution: the Java code in the formulae is step-wise unfolded and replaced by case distinctions and syntactic updates. Loops and recursion are handled using invariants and induction.

4 Specifications Extraction in the Simple Case

In the following, we describe the automatic specification extraction that has been implemented in the KeY system. The main idea is: (1) We use symbolic execution to construct an update and a path condition for every execution path of the program (as described in the previous section). Consider, for example, the simple program `x=x+2;x=x+3;` that has only one path (and no condition). The resulting update is $\{\texttt{x} := \texttt{x+5}\}$. (2) To construct a specification, the updates are then applied to a (trivial) post-condition Φ consisting of equalities of the form $x' \doteq \texttt{x}$ (for every location `x` that may be changed by the program). The new variable x' represents the post-value of `x`. In the example, applying the update results in the equality $x' \doteq \texttt{x+5}$, which specifies the program.

We demonstrate this idea in more detail using the following program that switches the values of the variables `x` and `y`:

—— Java ——

```java
d=myMath.abs(x-y);
if (x<y) { x=x+d; y=y-d; }
else     { x=x-d; y=y+d; }
```

—— Java ——

The specification extraction is realised by the construction of a proof tree for the proof obligation $\Gamma \Longrightarrow \langle\alpha\rangle\Phi$, where α is the above JAVA program and Φ is the (trivial) post-condition $x' \doteq \mathtt{x} \wedge y' \doteq \mathtt{y}$ (where x', y' represent the post-values of `x` resp. `y`). In general, Φ contains an equation for every location that the program may change (these locations may be given by the user or found automatically by analysing the program). If an approximation is used and not all locations occur in Φ that are actually changed, the constructed specification lacks information about locations not mentioned in Φ but is correct.

The premiss Γ may contain lemmas and specifications of library functions. In our example, it consists of the method contract for the method `abs`:

$$(\mathtt{x} \geq 0 \rightarrow \langle\mathtt{res=abs(x)}\rangle\mathtt{res} \doteq \mathtt{x}) \ \wedge \ (\mathtt{x} < 0 \rightarrow \langle\mathtt{res=abs(x)}\rangle\mathtt{res} \doteq -\mathtt{x}) \ .$$

Using this contract and applying some simplification steps to perform the case distinction between $\mathtt{x} \geq 0$ and $\mathtt{x} < 0$, we obtain the following partial proof tree:

$$\dfrac{\dfrac{\dfrac{(B_1) \qquad (B_2)}{\begin{array}{l}\Gamma', \mathtt{x} \geq \mathtt{y} \Longrightarrow \\ \quad \{\mathtt{d} := \mathtt{x} - \mathtt{y}\}\langle\mathtt{if}\ \ldots\rangle\Phi\end{array}} \qquad \dfrac{(B_3) \qquad (B_4)}{\begin{array}{l}\Gamma', \mathtt{x} < \mathtt{y} \Longrightarrow \\ \quad \{\mathtt{d} := -(\mathtt{x} - \mathtt{y})\}\langle\mathtt{if}\ \ldots\rangle\Phi\end{array}} \qquad (B_5)}{\cdots}}{\Gamma \Longrightarrow \langle\mathtt{d=myMath.abs(x-y);\ if}\ \ldots\rangle\Phi}$$

where Γ' contains the additional condition $\neg(\mathtt{myMath} \doteq \mathtt{null})$ and the subtree (B_5) considers the case of a null pointer dereferencing in line 2 of the source code. The subtrees (B_1) and (B_2) examine the case $\mathtt{x} \geq \mathtt{y}$. Moreover, (B_1) analyses the execution path where the condition in the if-statement is true. Thus, (B_1) is the following closed proof branch:

$$\dfrac{\dfrac{*}{\ldots, \mathtt{x} \geq \mathtt{y}, \mathtt{x} < \mathtt{y} \Longrightarrow \ldots}}{\ldots, \mathtt{x} \geq \mathtt{y}, \{\mathtt{d} := \mathtt{x} - \mathtt{y}\}(\mathtt{x} < \mathtt{y}) \Longrightarrow \{\mathtt{d} := \mathtt{x} - \mathtt{y}\}\langle\mathtt{x=x+d;\ y=y-d}\rangle\Phi} \qquad (B_1)$$

(B_1) and (B_4) are closed by a contradiction on the left side of the sequent as they correspond to execution paths that, in fact, are infeasible.

The specification parts corresponding to feasible execution paths are contained in open proof branches such as (B_2), which considers the case where the condition of the if-statement is false:

$$\dfrac{\dfrac{\dfrac{\dfrac{\ldots, \mathtt{x} \geq \mathtt{y} \Longrightarrow x' = \mathtt{y} \wedge y' = \mathtt{x}}{\ldots, \mathtt{x} \geq \mathtt{y} \Longrightarrow \{\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}\}\Phi}}{\ldots, \mathtt{x} \geq \mathtt{y} \Longrightarrow \{\mathtt{d} := \mathtt{x} - \mathtt{y} \,||\, \mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{y} - (\mathtt{x} - \mathtt{y})\}\Phi}}{\ldots, \mathtt{x} \geq \mathtt{y}, \neg(\mathtt{x} < \mathtt{y}) \Longrightarrow \{\mathtt{d} := \mathtt{x} - \mathtt{y} \,||\, \mathtt{x} := \mathtt{x} + (\mathtt{x} - \mathtt{y})\}\langle\mathtt{y=y-d}\rangle\Phi}}{\ldots, \mathtt{x} \geq \mathtt{y}, \{\mathtt{d} := \mathtt{x} - \mathtt{y}\}\neg(\mathtt{x} < \mathtt{y}) \Longrightarrow \{\mathtt{d} := \mathtt{x} - \mathtt{y}\}\langle\mathtt{x=x+d;\ y=y-d}\rangle\Phi} \qquad (B_2)$$

The open goal in (B_2) yields the pre-/post-conditions $\neg(\mathtt{myMath} \doteq \mathtt{null}) \wedge \mathtt{x} \geq \mathtt{y}$ and $x' = \mathtt{y} \wedge y' = \mathtt{x}$. The branch (B_3) is similar and yields the pre-/post-condition pair $\neg(\mathtt{myMath} \doteq \mathtt{null}) \wedge \mathtt{x} < \mathtt{y}$ and $x' = \mathtt{y} \wedge y' = \mathtt{x}$.

These two pairs could be simplified into one but that would remove structure from the specification. If the intended coverage criterion is path coverage, the black-box technique must be provided a distinct specification for each path.

Branch (B_5) handles the case that a `NullPointerException` is thrown at line 2 of the source code. We obtain the open goal

$$\texttt{myMath} \doteq \texttt{null} \Longrightarrow \langle\texttt{throw e;}\rangle\Phi$$

which expresses that an exception is thrown if $\texttt{myMath} \doteq \texttt{null}$.

The generated specification can now be translated into input languages for testing tools. In the example, we use the Java Modeling Language (JML):

—— Java + JML ——

```
/*@      requires REQ;
  @      ensures  ENS;
  @ also requires x>=y && myMath != null && REQ;
  @      ensures  y=\old(x) && x=\old(y) && ENS;
  @ also requires x<y  && myMath != null && REQ;
  @      ensures  y=\old(x) && x=\old(y) && ENS;
  @ also requires myMath == null         && REQ;
  @      signals  (NullPointerException e) true && ENS; @*/
 public swap() throws NullPointerException  { d=myMath... }
```

—— Java + JML ——

`REQ`/`ENS` is the `requires`/`ensures` pair from the original requirement specificaiton. A conjunctive cross-product has to be made from both specifications.

The generation of the test cases, the computation of the preamble, and the execution of the test suite is then performed by a black-box testing tool like UTJML [7] or JMLTT [4]. Using the structure of the specification, the test data $\{1, 0, M\}$, $\{0, 1, M\}$, $\{1, 0, null\}$ may be generated for the program variables $\{\texttt{x}, \texttt{y}, \texttt{myMath}\}$ (where M refers to an appropriate object). In this way all execution paths are excercised.

If the symbolically executed source code does not contain loops or recursive methods, then the set of the extracted pre-/post-conditions *can* ensure path coverage and even stronger coverage criteria. Which criterion is satisfied by the extracted specification depends on properties of the used method contracts and in particular on the subset of calculus rules that are actually used in the construction of a proof tree. Certain rules could simplify the program or the formulas too much with the effect that structural properties of the program are lost. The relation between the used verification calculus rules and coverage criteria is one of our current research topics.

Integer overflow checks and the creation and initialization of objects and classes, which has been ignored here due to space limitations, are covered by the KeY Java Card DL calculus.

5 Handling Loops and Recursion

The automatic extraction of a (partial) specification from a program that contains loops is implemented in the KeY tool by unwinding or unfolding. Invariant generation in general requires user interaction but in simple cases invariants can be automatically generated.

Since loops can create arbitrarily long execution paths, it is impossible to create a test set that satisfies a criterion like full path coverage in this case. However, by loop unwinding, a set of partial specifications can be generated that satisfies the path coverage criterion for a bounded amount of total loop iterations. Our experiences show that most loops behave similarly on loop iterations with different bounds. We will refere to the bound by K.

We describe the specification extraction by loop unwinding using the following program P as an example:

Java
```java
k=0; j=0; n=a.length; line.prev=null;
while (k<n) {
  if (j=23) {
    j = 0; oldline = line;
    line = new Line(23);
    oldline.next = line; line.prev = oldline;
  }
  if (j>23 || k>n) throw new Exception();
  line.setCharAt(j,a[k]);  k++; j++;
}
```
Java

This program copies values from the array `a` into dynamically created `Line` objects, where only 23 values can be saved in any one line (we assume that the first `Line` object is created and the array is initialised before P is started). Again due to space restrictions, we cannot present all details in program execution that are considered by the KeY tool.

By unwinding a loop, an if-cascade is created (for nested loops this has to be done recursively). From the program P, we obtain the following if-cascade:

Java
```java
if (k<n){if(j=23){..};if(j>23||k>n)..setCharAt(j,a[k]);k++;j++;
   if(k<n){if(j=23){..};if(j>23||k>n)..setCharAt(j,a[k]);k++;j++;
      ...
     while(k<n){...}
   }
}
```
Java

The specification extraction process results in a proof tree with K sequents of the following form as its leaves, where `body` represents the loop body:

$$\begin{array}{r}
\Gamma,\, \neg(k < n) \Longrightarrow \langle\rangle\Phi \\
\Gamma,\, k < n,\, \neg(k+1 < n) \Longrightarrow \langle \mathit{body}\,;\rangle\Phi \\
\Gamma,\, k < n,\, k+1 < n,\, \neg(k+2 < n) \Longrightarrow \langle \mathit{body}\,; \mathit{body}\,;\rangle\Phi \\
\dots
\end{array}$$

These sequents already show the desired pre-conditions; post-conditions are extracted as follows. From the formula $\langle\rangle\Phi$ we get the post-condition for the case that the loop iterates zero times, which is:

$$k' \doteq 0 \land j' \doteq 0 \land n' \doteq \texttt{a.length} \land a' \doteq \texttt{a} \land \mathit{line}' \doteq \texttt{line} \tag{1}$$

Simplification of the formula $\langle \mathit{body}\rangle\Phi$ results in the post-condition that holds for all execution paths where the loop iterates exactly once. We assume that the following contract is given for the method `setCharAt`:

$$\begin{array}{l}\neg(\texttt{line} \doteq \texttt{null}) \land \langle\texttt{ret=line.getLength()}\rangle(0 < \texttt{x} \land \texttt{x} < \texttt{ret}) \to \\ \quad\langle\texttt{line.setCharAt(x,c); res=line.getCharAt(x)}\rangle(\texttt{res} \doteq \texttt{a}[\texttt{x}])\end{array} \tag{2}$$

Then, the post-condition that is extracted for the case where the loop executes exactly once is the following conjunction:

$$k' \doteq 1 \land j' \doteq 1 \land n' \doteq \texttt{a.length} \land a' \doteq \texttt{a} \land \mathit{line}' \doteq \texttt{line} \land \tag{3}$$

$$\langle\texttt{ret=line.getCharAt(0)}\rangle(\texttt{ret} \doteq \texttt{a}[0]) \tag{4}$$

Conjunction (3) is an updated version of the post-condition (1), and (4) it the result of applying the method contract (2).

An invariant inference tool that generates inequalities is able to infere the invariant

$$0 \le k \le n \ \land \ 0 \le j \le n \ \land \ j \le 23$$

automatically. This invariant is not the strongest and therefore it is less accurate on the first K iterations than the specification we have obtained by unrolling. But in contrast to unrolling it contains information about *all* possible iterations of the loop. Using this invariant allows to generate different test cases that exercise the if-statements in the loop body. In order to execute the then-case of the if-statement, the condition $\texttt{j} \doteq 23$ must be true. Applying an invariant rule results in a proof obligation that the invariant is in fact preserved by the loop body. Applying calculus rules to that obligation corresponds to a symbolic execution of the loop body (including the if-statement). Leaves of that part of the proof then result in pre-/post-condition pairs from which test cases with array length 23 can be generated.

The purpose of this example was to show that loop unwinding and the invariant rule are complementary concepts by which different kinds of test sets can be produced that execute different parts of code. The automatic generation of loop invariants will be implemented in the KeY tool and an integration of the dynamic invariants inference tool Daikon is considered (similar to the integration of Daikon into ESC/Java [15]). Relevant work on automatic invariant inference can also be found in [9, 10, 6, 16].

6 Related Work

In the Echo approach [18], a requirement specification is manually refined until an implementation is obtained. The correctness of the manual refinement process is then verified by extracting a specification from the implementation and comparing it to the requirement specification (this does not involve testing).

Synergies between using specifications and testing are also explored in [20]. This approach is similar to our's, because it coincides with the second step in our approach. The difference is, however, in the first step as the extracted specification is obtained by *dynamic* analysis and, therefore, the result is a black-box testing method.

Another related approach is described by Nimmer [15]. It involves, however, dynamic analysis (where we use static analysis) and, in the second step, theorem proving (where we use testing). The Korat system [5] uses symbolic execution to generate test cases (without generating a specification).

Deduction-based specification extraction is similar to test-case generation by symbolic execution (like in Symstra [19])—except that our approach allows to also derive post-conditions.

Our approach is complementary to these methods, and there is no work that more clearly separates static program analysis from test case generation (like it is done in our approach), in order to combine test coverage criteria from the two complementary techniques.

7 Conclusion

We have described how deductive program verification systems can be used to generate specifications, which then can be used as input for black-box testing tools, turning them into white-box testing methods. This approach can be adapted to other symbolic execution methods (e.g., weakest precondition calculi) that allow to extract specifications from programs, provided that the structure of the extracted specification reflects the structure of the program. Since the pre- and post-conditions are extracted a reference implementation can be used as a requirement specification. Furthermore this allows to establish a connection between variables before and after the program execution.

Future work is to investigate the precise relation between test coverage and different simplification rules used for specification extraction.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
2. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.
4. F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-testing-tools: a symbolic animator for JML specifications using CLP. In N. Halbwachs and L. Zuck, editors, *Proceedings, Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Edinburgh, UK*, LNCS 3440, pages 551–556. Springer, 2005.
5. 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.
6. A. Bundy and V. Lombart. Relational rippling: A general approach. In *Proceedings, International Joint Conference on Artificial Intelligence, Montréal, Canada*, pages 175–181. Morgan Kaufmann, 1995.
7. 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.
8. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In Y. Gurevich, editor, *Proceedings, Testing and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007. To appear.
9. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.
10. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings, Principles of Programming Languages (POPL), Portland, USA*, pages 191–202. ACM, 2002.
11. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
12. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
13. N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary coverage criteria for test generation from formal models. In *Proceedings, Software Reliability Engineering, Saint-Melo, France*, pages 139–150. IEEE CS, 2004.
14. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In L.-H. Eriksson and P. A. Lindsay, editors, *Proceedings, Formal Methods Europe (FME), Copenhagen, Denmark*, LNCS 2391. Springer, 2002.
15. J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.
16. J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 229–239, 2002.
17. Parasoft. JTest manual, 2004. http://www.parasoft.com/jtest.
18. E. A. Strunk, X. Yin, and J. C. Knight. Echo: a practical approach to formal verification. In *Proceedings, Formal Methods for Industrial Critical Systems (FMICS), Lisbon, Portugal*, pages 44–53. ACM, 2005.
19. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings, Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Edinburgh, UK*, LNCS 3440, pages 365–381. Springer, 2005.
20. T. Xie and D. Notkin. Exploiting synergy between testing and inferred partial specifications. In *Proceedings, ICSE Workshop on Dynamic Analysis (WODA), Portland, USA*, pages 17–20, 2003.