The Symbolic Execution Debugger:
a Productivity Tool for Java Based on Eclipse and KeY

Martin Hentschel, Richard Bubel, and Reiner Hähnle

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
hentschel|bubel|haehnle@cs.tu-darmstadt.de

Abstract. We present the Symbolic Execution Debugger (SED), an extension of the Eclipse debug platform for interactive symbolic execution. Being based on symbolic execution, its functionality goes beyond that of traditional interactive debuggers. For instance, debugging can start directly at any method or statement and all program execution paths are explored simultaneously. To support program comprehension, execution paths as well as intermediate states are visualized.
Using KeY as underlying symbolic execution engine, SED supports sequential JAVA programs and the inspection of verification proofs.

Keywords: Symbolic Execution, Debugging, Program Execution Visualization

1 Introduction

This updated and extended version of [7] presents the Symbolic Execution Debugger[1] (SED), a language independent extension of the Eclipse debug platform for symbolic execution. Symbolic execution [3,4,10,11] is a program analysis technique based on the interpretation of a program with symbolic values. This makes it possible to explore *all* concrete execution paths (up to a finite depth). We describe an SED implementation that uses KeY [2] as the underlying symbolic execution engine, supporting sequential JAVA without floats, garbage collection and dynamic class loading. Our main contributions are the SED platform, interactive symbolic execution of JAVA and visualization of program behavior including unbounded loops and method calls.

The SED supports traditional debugger functionality like step-wise execution or breakpoints, and enhances it as follows: Debugging can begin at any method or any other statement in a program, no fixture is required. The initial state can be specified partially or not at all. During symbolic execution all feasible execution paths are discovered, thus it is not necessary to set up a concrete initial program state leading to an execution where a targeted bug occurs. At any time each intermediate state can be inspected using the SED. Intermediate states

[1] The website www.key-project.org/eclipse/SED contains an Installation and User Guide (including instructions on how to use API classes), as well as a screencast and theoretical foundations.

tend to be small and simple, because symbolic execution can be started close to the suspected location of a bug and the symbolic states contain only program variables accessed during execution. This makes it easy for the bug hunter to comprehend intermediate states and the actions performed on them to find the origin of a bug. Heisenbugs [5], a class of program errors that disappear while debugging, are avoided as the behavior of a program is correctly reflected in its symbolic execution. Besides debugging the SED platform allows to visualize and explore results of static analysis based on symbolic execution.

2 Symbolic Execution

Symbolic execution (SE) means to execute a program with symbolic values in lieu of concrete values. We explain SE and how it is used interactively in the SED by example: method `eq` shown in the listing in Fig. 1 compares the given `Number` instance with the current one.

For a JAVA method to be executed it must be called explicitly. For instance, the expression `new Number().eq(new Number());` invokes `eq` on a fresh instance with a different instance as argument. This results in a single execution path: first the guard in line 5 is evaluated to true, as fields of integer type are initialized with 0 by default. Finally, true is returned as result. To inspect another execution path the method has to be called in a different state.

Let us execute method `eq` symbolically, i.e., without a concrete argument, but a reference to a symbolic value n which can represent any object or `null`. In our SE tree notation we use different icons to underscore the semantics of nodes. As Fig. 1 shows, the root is a *Start Node* representing the initial state and the program fragment (any method or any block of statements) to execute. Here a call to `eq` is represented by its *Method Call* child node.

The `if`-guard, represented as a *Branch Statement* node, splits execution when the field `value` is accessed on the symbolic object n. Because nothing is known about n, it could be `null`. The *Branch Condition* children nodes show the condition under which each path is taken. On the left, where n is not `null`, the comparison in the `if`-guard splits execution again. If both values are the same, the `return` statement is executed, indicated by a *Statement* node. Now the symbolic path of the method is fully executed and returns true in the *Method Return* child node. This SE path ends in the *Termination* node. The branch where the values are different looks similar, but false is returned instead. In the rightmost branch the parameter n has the value `null` and SE ends with an uncaught `NullPointerException`, visualized as an *Exceptional Termination* node.[2]

In contrast to concrete execution, SE does not require fixture code and discovers all feasible execution paths (up to its execution depth). Each SE path through an SE tree may represent infinitely many concrete executions and is characterized by its path condition (the conjunction of all branch conditions on

[2] The instantiation of the thrown exception is not visualized since we do not include execution of JAVA API methods for simplicity.

```
1  public class Number {
2     private int value;
3
4     public boolean eq(Number n) {
5        if (value == n.value) { return true; }
6        else { return false; }
7     }
8
9     // ...
10 }
```

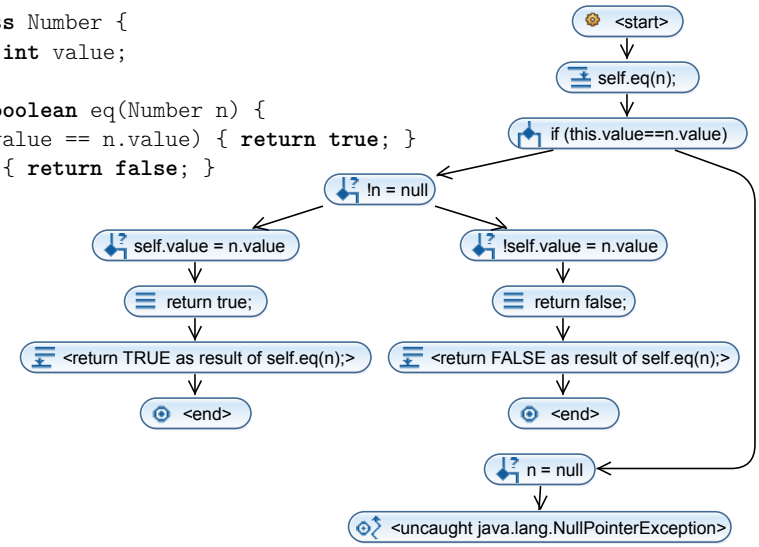


Fig. 1: Source code of class `Number` and SE tree of method `eq`

it). SE may not terminate in presence of loops and recursive methods which can be avoided by applying loop invariants or method contracts, see Section 4.

3 Basic Usage of the Symbolic Execution Debugger

The SED is realized as an Eclipse plugin. SE of a selected method or selected statements in a method can be started via the Eclipse context menu item *Debug As, Symbolic Execution Debugger (SED)*. The user is then offered to switch to the *Symbolic Debug* perspective, which provides all relevant views for interactive symbolic execution (see Fig. 2).

The *Debug* view allows, as usual, to switch between debug sessions and to control program execution. Instead of the current stack trace of active threads, the view shows the traversed SE tree. An alternative and more sophisticated visualization of the SE tree is shown in the *Symbolic Execution Tree* view. To ease navigation within large SE trees a thumbnail view called *Symbolic Execution Tree (Thumbnail)* is provided. The SE tree of the screenshot (Fig. 2) is identical to the tree in Fig. 1. The additional frames (blue rectangles) displayed in view *Symbolic Execution Tree* represent the bounds of code blocks. Such frames can be independently collapsed and expanded to abstract away from the inner structure of code blocks, thus achieving a cleaner representation of the overall code structure by providing only as much detail as required for the task at hand. A collapsed frame contains only one branch condition node per path (namely the conjunction of all branch condition of that particular path), displaying the constraint under which the end of the corresponding code block is reached.

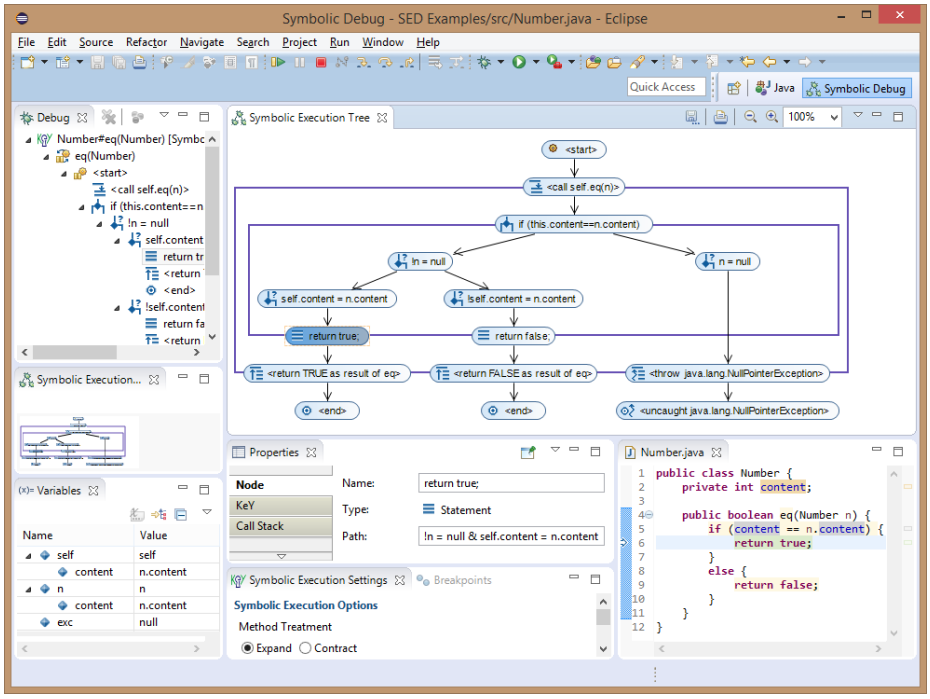


Fig. 2: Symbolic Execution Debugger: interactive symbolic execution

The symbolic program state of a node consists of variables and their symbolic values. It can be inspected in the *Variables* view. The details of a selected variable (e.g. additional constraints) or node (path condition, call stack, etc.) are available in the *Properties* view. The source code line corresponding to the selected SE tree node is highlighted in the editor. Additionally, the editor highlights statements and code members reached during symbolic execution.

The *Symbolic Execution Settings* view lets one customize SE, e.g., one can choose between method inlining and method contract application. Breakpoints suspend the execution and are managed in the *Breakpoints* view.

In Fig. 2 the SE tree node **`return true;`** is selected. In the *Variables* view we can see that the symbolic values of field `value` are identical for the objects referenced by `self` (the current instance) and parameter `n`. This is exactly what is enforced by the path condition. A fallacy and source of bugs is to implicitly assume that `self` and `n` refer to different instances as they are named differently and here also because that an object is passed to itself as a method argument. But the path condition is also satisfied if `n` and `self` reference the same object. The SED helps to detect and locate unintended aliasing by determining and visualizing all possible memory layouts w.r.t. the current path condition.

Selecting context menu item *Visualize Memory Layouts* of an SE tree node creates a visualization of possible memory layouts as a *symbolic object diagram*

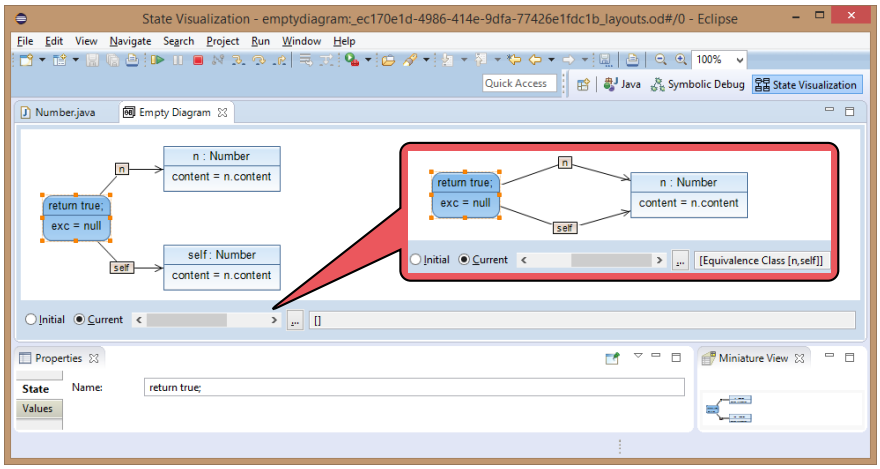


Fig. 3: Symbolic Execution Debugger: different memory layouts

(see Fig. 3). It resembles a UML object diagram and shows the dependencies between objects, the values of object fields and the local variables of the current state.

The root of the symbolic object diagram is visualized as a rounded rectangle and shows all local variables visible at the current node. In Fig. 3, the local variables `n` and `self` refer to objects visualized as rectangles. The content of the instance field `value` is shown in the lower compartment of each object. The local variable `exc` is used by KeY to distinguish among normal and exceptional termination.

The toolbar (near the origin of the callout) allows to select different possible layouts and to switch between the current and the initial state of each layout. The initial state shows how the memory layout looked before the execution started resulting in the current state. Fig. 3 shows both possible layouts of the selected node **`return true`**; in the current state. The second memory layout (inside the callout) represents the situation, where `n` and `self` are aliased.

4 Usage Scenarios

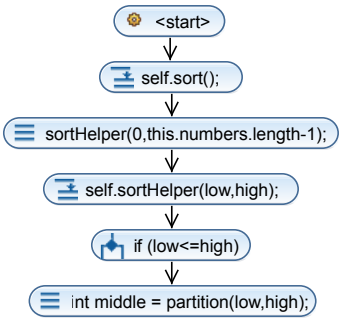
Like a traditional debugger, the SED helps the user to control execution and to comprehend each performed step. It is helpful to focus on a single branch where a buggy state is suspected. (To change the focus to a different branch, no new debugging session or new input values are needed). It is always possible to revisit previous steps, because each node in the SE tree provides the full state.

Finding the Origin of Bugs The explicit rendering of different control flow branches in the SE tree constitutes a major advantage over traditional debuggers. Unexpected or missing expected branches are good candidates for possible

sources of bugs. Fig. 4a shows a buggy part of a Quicksort implementation for sorting array `numbers`. Within a concrete execution of a large application a `StackOverflowError` was thrown. It indicates that method `sortHelper` calls itself infinitely often. Using SED we start debugging close to the suspected location of the bug, namely, at method `sort`. Executing the method stepwise, exhibits execution paths taken when invoking the method in an illegal state. Exploration of such cases can be avoided by providing a precondition which limits the initial symbolic state. In this example, we exclude empty arrays by specifying the precondition `numbers != null && numbers.length >= 1` in the *debug configuration*. After a few steps, the SE tree produced by SED (see Fig. 4b) shows that the **`if`** statement is not branching. This is suspicious and deserves closer attention. Inspecting the **`if`** guard shows that the comparison should have been `low < high` and the source of the bug is found.[3]

```
1  public class QuickSort {
2     private int[] numbers;
3
4     public void sort() {
5        sortHelper(0, numbers.length - 1);
6     }
7
8     private void sortHelper(int low, int high) {
9        if (low <= high) {
10          int middle = partition(low, high);
11          sortHelper(low, middle);
12          sortHelper(middle + 1, high);
13       }
14    }
15
16    private int partition(int low,
17                          int high) {
18       // ...
19    }
20 }
```

(a) Buggy Quicksort implementation (from [6])



(b) SE tree

Fig. 4: Quicksort example

Program and Specification Understanding SE trees show control and data flow at the same time. Thus they can be used to help understanding programs and specifications just by inspecting them. This can be useful during code reviews or in early prototyping phases, where the full implementation is not yet available. It works best, when partial method contracts and invariants are available to achieve

[3] Without the precondition the bug can be observed as well, but a little later.

compact and finite SE trees. However, useful specifications can be much weaker than what would be required for verification. The listing in Fig. 5 shows a buggy implementation of method `indexOf` with a very simple loop invariant written in the Java Modeling Language (JML) [12]. We configured the symbolic execution engine to apply loop invariants instead of unrolling loops, which guarantees a finite SE tree. The resulting SE tree under precondition `a != null` is also shown in Fig. 5. Application of the loop invariant splits execution into two branches. *Body Preserves Invariant* represents all loop iterations and *Use Case* continues execution after the loop (full branch conditions are not shown for brevity).

Even without checking any further details, it is already indicated by the icon crossed out in red that the leftmost branch terminates in a state where the loop invariant is not preserved. Now, closer inspection shows the reason to be that, when the array element is found, the variable `i` is not increased, hence the **`decreasing`** clause (`a.length - i`) of the invariant is violated. The two branches below the *Use Case* branch correspond to the code after the loop has terminated. In one case an element was found, in the other not. Looking at the return node, however, we find that in both cases instead of the `index` computed in the loop, the value of `i` is returned.

```
1  public static int indexOf(int[] a,
2                            int   s) {
3     int index = -1;
4     int i = 0;
5     /*@ loop_invariant i >= 0 && i <= a.length;
6       @ decreasing a.length - i;
7       @ assignable index, i;
8       @*/
9     while (index < 0 && i < a.length) {
10       if (s == a[i]) { index = i; }
11       else { i++; }
12    }
13    return i;
14 }
```

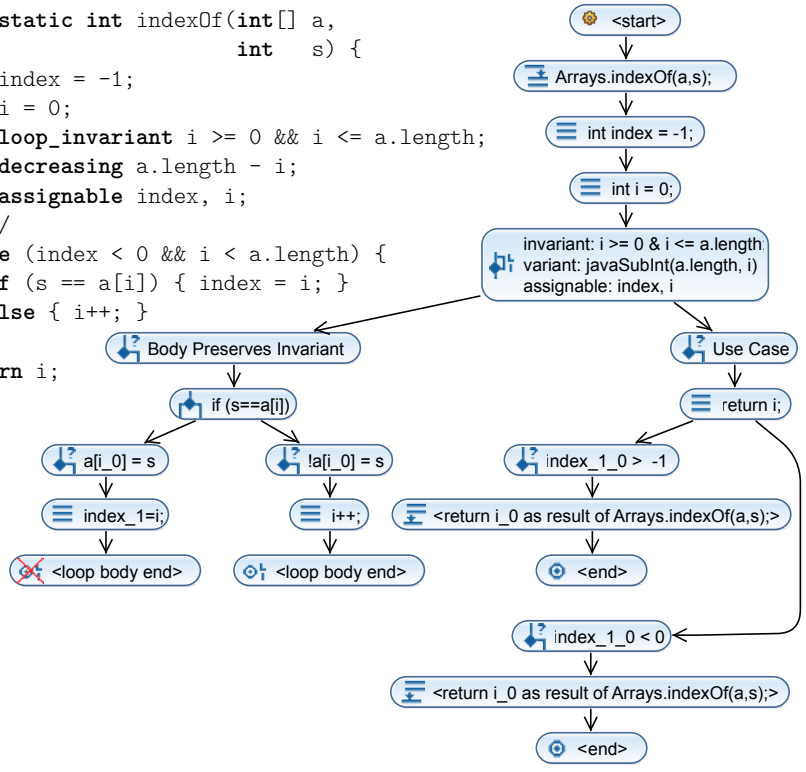


Fig. 5: Buggy and partially specified implementation of `indexOf` and its SE Tree

Our examples demonstrate that SE trees can be used to answer questions about thrown exceptions or returned values. In SED the full state of each node is available and can be visualized. Thus it is easily possible to see whether and where new objects are created and which fields are changed when (comparison between initial and current memory layout).

Using breakpoints, symbolic execution is continued until a breakpoint is hit on any branch. Breakpoints can be attached to a line of code with or without a condition or they may consist only of a condition. Thus they can be used to find execution paths that (i) throw a specified exception, (ii) access or modify a specified field, (iii) invoke or return from a specified method. Breakpoints can also be used to (iv) control loop unwinding and recursive method invocation and (v) to stop at an intermediate state that has a specified property.

5 Verification with SED and KeY

The SED platform allows to perform SE interactively, to visualize the resulting symbolic execution tree, and to inspect symbolic states. Together, this results in a powerful debugging tool that in addition can be used to control SE and to present results of an SE-based analysis.

Going beyond mere SE, the SED can also verify that a Java program satisfies a given specification written in JML, because it uses KeY as its underlying symbolic execution engine.[4] A program is correct with respect to its specification if and only if each branch in the SE tree ends with a termination node and no icons are crossed out in red are displayed in the whole tree. In this case all branches terminate in a state where the given postcondition (i.e., JML ensures clause) is fulfilled. If a method call was approximated by a method contract, the precondition- and caller-no-null checks must have been successful, too. In addition, all loop invariants present were valid at the start of their loop and were preserved by the loop body.

An SE tree produced by the SED displays considerably less information than a full proof tree in KeY [2]: while the former contains only nodes that correspond to reachable program states, the latter shows all intermediate SE steps performed during proof construction, including the proof steps for pure first-order verification conditions. Hence, the SED provides a software developer's view on a KeY proof, hiding intermediate and non-SE related steps. Program states are visualized in a user-friendly way and are not encoded as formulas often distributed and hidden within large proof goals. A major limitation of the SED compared to the KeY prover is that it is currently not possible in SED to continue a proof interactively in case KeY's proof strategy was not powerful enough to close some goal automatically. But it provides still the means to interact with the prover by adapting or inserting additional JML assertions and thus to use KeY with an auto-active flavor [16,13].

[4] The debug configuration allows to select a method contract alternatively to a precondition.

6 Architecture

The SED extends Eclipse and can be added to existing Eclipse-based products. In particular, SED is compatible with Eclipse's Java Development Tools (JDT). To ensure compatibility and to obtain a seamless integration with the Eclipse user interface, SED uses and extends the Eclipse platform as shown in Fig. 6.

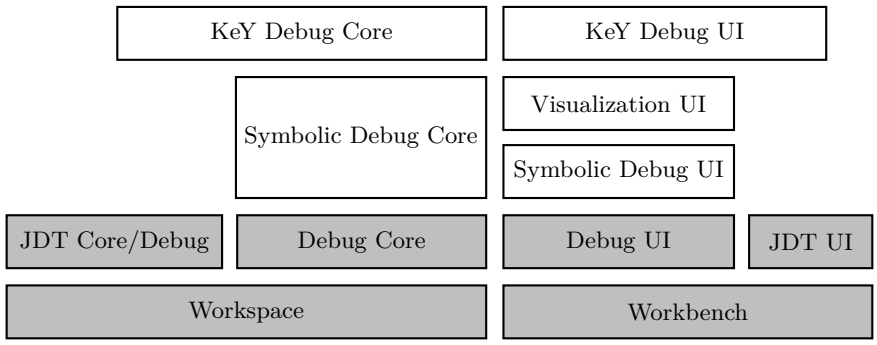


Fig. 6: Architecture of the Symbolic Execution Debugger (SED). Eclipse components are shaded in grey, our extensions have a white background.

The core of Eclipse is the Workspace which manages the projects and the user interface (Workbench) with its editors, views and perspectives. The Debug Platform extends these with language independent facilities for debugging (Debug Core and Debug UI). Finally, JDT offers functionality to edit and debug Java programs (JDT Core/Debug and JDT UI).

The Symbolic Debug Core component extends the debug model of the Debug Platform for symbolic execution, independently from specific target languages and symbolic execution engines. Additional UI extensions (Symbolic Debug UI) and visualization capabilities (Visualization UI) are available.

The KeY Debug Core component implements the extended debug model for symbolic execution based on KeY's symbolic execution engine (available as pure Java API). The user interface extensions required to launch Java methods and to execute statements symbolically are provided by the KeY Debug UI.

The architecture of the SED platform allows us to integrate different symbolic execution engines for the purpose of debugging, program understanding, and to control analyses based on symbolic execution. In order to integrate a new SE engine, it suffices to implement the extended debug model for symbolic execution and to create the user interface extensions to start and control the symbolic execution. [5]

[5] The website www.key-project.org/eclipse/SED provides additional documentation and an example SED implementation as a starting point.

7 Related and Future Work

A number of recent tools implement SE for program verification [9] or test generation [1,15], which are complementary to SED. In fact, SED could be employed to control or visualize these tools. As far as we know, EFFIGY [11] was the first system that allowed to interactively execute a program symbolically in the context of debugging. It did not support specifications or visualization.

The Eclipse plugin of Java Path Finder (JPF) [14] prints the analysis results obtained from SE as a text report, but does neither provide graphical visualization nor interactive control of SE. JPF is prototypically supported by SED as an alternative SE engine.

The SE engine and its Eclipse integration described in [8] features non-interactive graphic visualization of the SE tree. SED allows to interact with the visualization as a means to control SE and to inspect symbolic states.

A prototypic symbolic state debugger that could not make use of method contracts and loop invariants was presented in [6]. However, that tool was not very stable and its architecture was tightly integrated into the KeY system. As a consequence, the SED was developed from scratch as a completely new application featuring significant extended and new functionality. It is realized as a reusable Eclipse extension which allows to integrate different SE engines.

We plan to extend the verification capabilities of the SED to create a full-fledged alternative GUI of the KeY verification system [2]. The visualization capabilities and a debugger-like interface will flatten the learning curve to use a verification system. On the other hand, exploiting verification results during SE allows to classify execution paths automatically as correct or wrong.

References

1. E. Albert, I. Cabanas, A. Flores-Montoya, M. Gomez-Zamalloa, and S. Gutierrez. jPET: An Automatic Test-Case Generator for Java. In *Proc. of the 18th Working Conf. on Reverse Engineering*, WCRE '11, pages 441–442. IEEE CS, 2011.
2. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
3. R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, June 1975.
4. R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.
5. M. Grottke and K. S. Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, 2005.
6. R. Hähnle, M. Baum, R. Bubel, and M. Rothe. A visual interactive debugger based on symbolic execution. In *ASE*, pages 143–146, 2010.
7. M. Hentschel, R. Bubel, and R. Hähnle. Symbolic Execution Debugger (SED). In B. Bonakdarpour and S. A. Smolka, editors, *Proceedings of Runtime Verification 2014*, LNCS, pages 255–262. Springer, Sept. 2014.
8. A. Ibing. Parallel SMT-Constrained Symbolic Execution for Eclipse CDT/Codan. In *ICTSS*, pages 196–206, 2013.

9. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proc. of the 3rd Intl. Conf. on NASA Formal Methods*, pages 41–55. Springer, 2011.
10. S. Katz and Z. Manna. Towards automatic debugging of programs. In *Proc. of the Intl. Conf. on Reliable software, Los Angeles*, pages 143–155. ACM Press, 1975.
11. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
12. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. *JML Reference Manual*, Sept. 2009.
13. K. R. M. Leino and M. Moskal. Usable Auto-Active Verification. `http://fm.csl.sri.com/UV10/submissions/uv2010_submission_20.pdf`, 2010.
14. C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software. In *Proc. of the 2008 Intl. Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26. ACM, 2008.
15. N. Tillmann and J. De Halleux. Pex: White Box Test Generation for .NET. In *Proc. of the 2nd Intl. Conf. on Tests and Proofs*, pages 134–153. Springer, 2008.
16. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. Autoproof: Auto-active functional verification of object-oriented programs. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 566–580. Springer, 2015.