Symbolic Execution Debugger (SED)

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 for sequential Java programs. 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.

Keywords: Symbolic Execution, Debugging, Program Execution Visualization.

1 Introduction

We present the Symbolic Execution Debugger (SED),[1] a language independent extension of the Eclipse debug platform for symbolic execution. Symbolic execution [3,4,9,10] 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 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.

[1] The website www.key-project.org/eclipse/SED provides an installation & user guide (with instructions on how to use API classes), screencast and theoretical background.

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.

```
 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 }
```

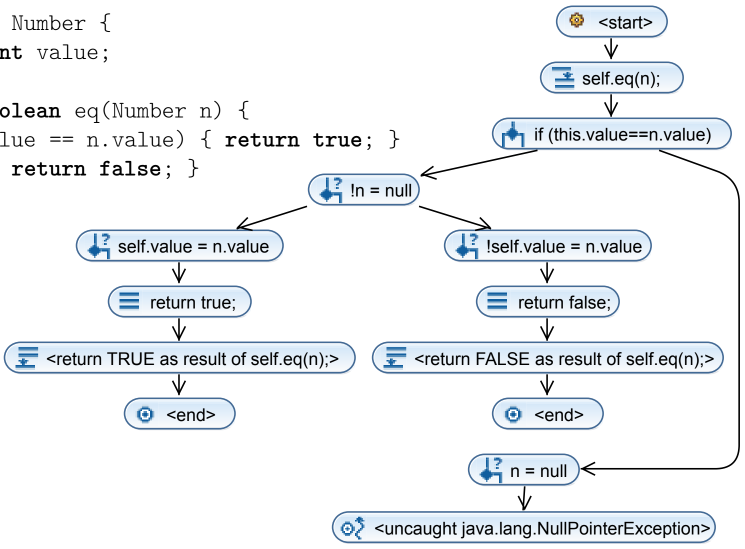


Fig. 1. Source code of class `Number` and SE tree of method `eq`

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 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

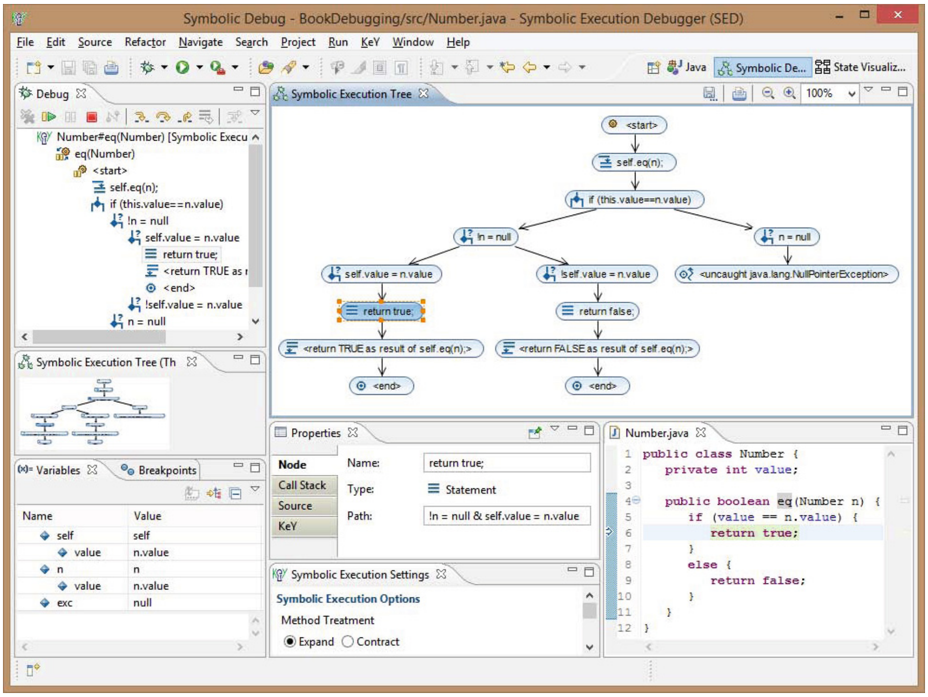
The SED is realized as an Eclipse plugin. SE of a selected method or selected statements in a method can be started via the Eclipse context menu item *Debug As, Symbolic Execution Debugger (SED).* The user is then offered to switch to the *Symbolic Debug* perspective, which provides all relevant views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

The *Debug* view allows, as usual, to switch between debug sessions and to control program execution. In case of SE, the view shows the traversed SE tree, instead of the current stack trace. The SE tree is also visualized in the *Symbolic Execution Tree* view (it is identical to the tree in Fig. 1). An SE tree sketch is provided by the *Symbolic Execution Tree (Thumbnail)* view to help navigation. The symbolic program state of a node consists of variables and their symbolic values. It can be inspected in the *Variables* view. Breakpoints suspend execution and are managed in the *Breakpoints* view. The details of a selected 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. The *Symbolic Execution Settings* view lets one customize SE, e.g., choose between method inlining and method contract application.

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. In an object-oriented setting one could think that `self` and `n` refer to different instances, but this needs not to be the case. The path condition is also satisfied if `n` and `self` reference the same object. Unintended *aliasing* is a source of bugs. The SED helps to find these by determining and visualizing all possible memory layouts w.r.t. the 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* (see Fig. 3). It resembles a UML object diagram and shows the dependencies

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



Name	Value
self	self
value	n.value
n	n
value	n.value
exc	null

Name: return true;
Type: Statement
Path: !n = null & self.value = n.value

```
public class Number {
   private int value;

   public boolean eq(Number n) {
      if (value == n.value) {
         return true;
      }
      else {
         return false;
      }
   }
}
```

Symbolic Execution Options
Method Treatment
[✓ Expand] Contract

Fig. 2. Symbolic Execution Debugger: Interactive symbolic execution

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 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.

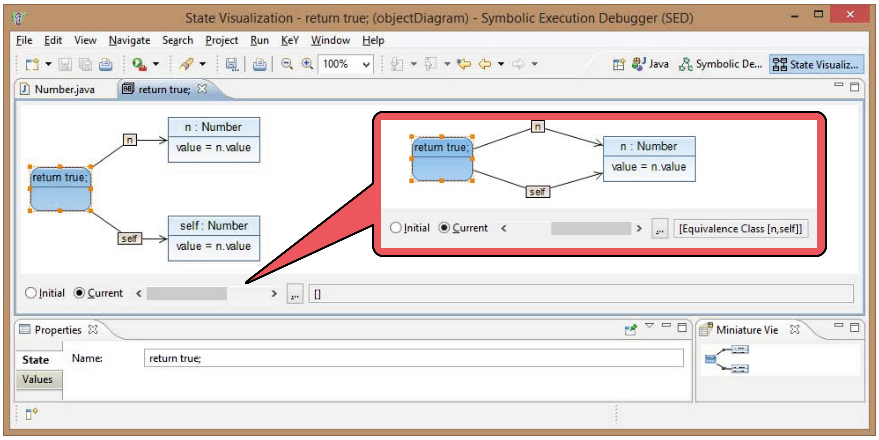


Fig. 3. Symbolic Execution Debugger: Different memory layouts

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]

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 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 JML. We configured the symbolic execution engine to apply

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

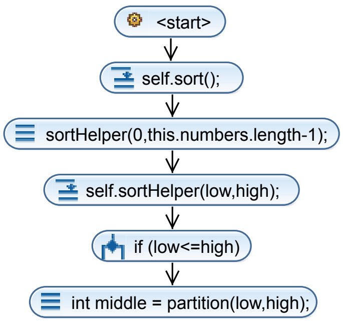
```
public class QuickSort {
   private int[] numbers;

   public void sort() {
      sortHelper(0, numbers.length - 1);
   }

   private void sortHelper(int low, int high) {
      if (low <= high) {
         int middle = partition(low, high);
        sortHelper(low, middle);
         sortHelper(middle + 1, high);
      }
   }

   private int partition(int low,
                              int high) {
      // ...
   }
}
```

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


<start>
self.sort();
sortHelper(0,this.numbers.length-1);
self.sortHelper(low,high);
if (low<=high)
int middle = partition(low,high);

(b) SE tree

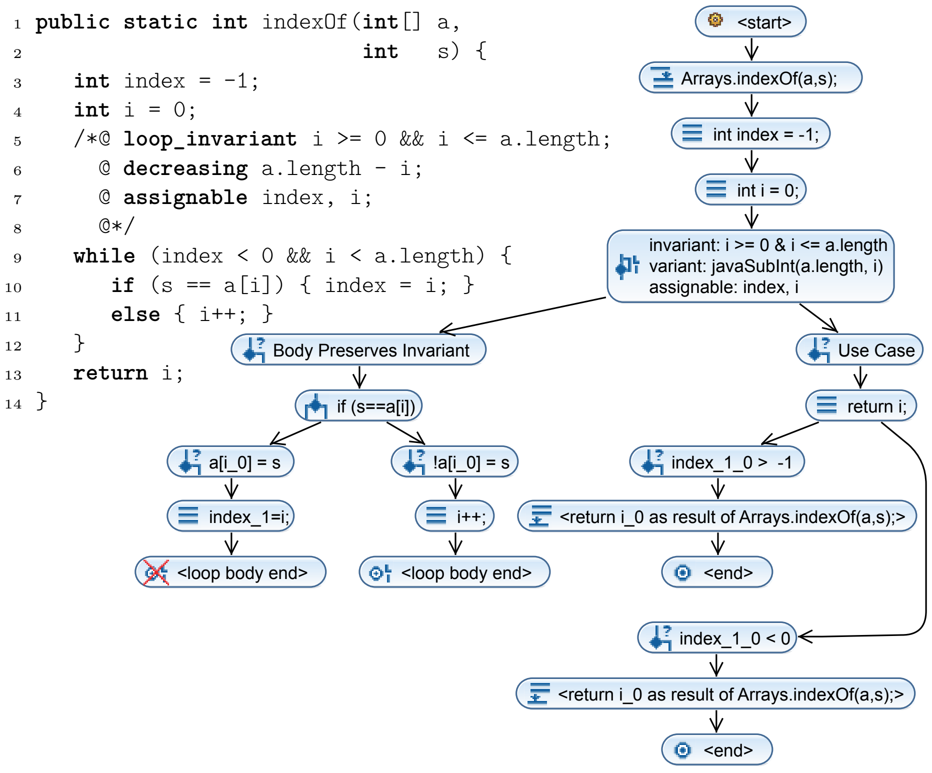
Fig. 4. Quicksort example

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).

Without checking further details, one can see 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.

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



```
public static int indexOf(int[] a,
                          int   s) {
   int index = -1;
   int i = 0;
   /*@ loop_invariant i >= 0 && i <= a.length;
     @ decreasing a.length - i;
     @ assignable index, i;
     @*/
   while (index < 0 && i < a.length) {
      if (s == a[i]) { index = i; }
      else { i++; }
   }
   return i;
}
```

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

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 Related and Future Work

A number of recent tools implement SE for program verification [8] or test generation [1,12], which are complementary to SED. In fact, SED could be employed to control or visualize these tools. As far as we know, EFFIGY [10] 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) [11] 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 [7] 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 symbolic execution engines.

We plan to use the visualization of an SE tree as an 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. Complementary techniques to SE like backward slicing could help the user to find the origin of bugs more easily. Visualization capabilities could be improved by grouping nodes based on code members like methods or loop bodies. In this way more information is visualized and fully executed groups could be collapsed.

References

1. Albert, E., Cabanas, I., Flores-Montoya, A., Gomez-Zamalloa, M., Gutierrez, S.: jPET: An Automatic Test-Case Generator for Java. In: Proc. of the 18th Working Conf. on Reverse Engineering, WCRE 2011, pp. 441–442. IEEE CS (2011)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT—A formal system for testing and debugging programs by symbolic execution. ACM SIGPLAN Notices 10(6), 234–245 (1975)
4. Burstall, R.M.: Program proving as hand simulation with a little induction. In: Information Processing 1974, pp. 308–312. Elsevier/North-Holland (1974)
5. Grottke, M., Trivedi, K.S.: A classification of software faults. Journal of Reliability Engineering Association of Japan 27(7), 425–438 (2005)
6. Hähnle, R., Baum, M., Bubel, R., Rothe, M.: A visual interactive debugger based on symbolic execution. In: ASE, pp. 143–146 (2010)
7. Ibing, A.: Parallel SMT-Constrained Symbolic Execution for Eclipse CDT/Codan. In: Yenigün, H., Yilmaz, C., Ulrich, A. (eds.) ICTSS 2013. LNCS, vol. 8254, pp. 196–206. Springer, Heidelberg (2013)
8. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
9. Katz, S., Manna, Z.: Towards automatic debugging of programs. In: Proc. of the Intl. Conf. on Reliable Software, Los Angeles, pp. 143–155. ACM Press (1975)
10. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM 19(7), 385–394 (1976)
11. Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: 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 2008, pp. 15–26. ACM (2008)
12. Tillmann, N., de Halleux, J.: Pex: White Box Test Generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)