Chapter 11
Debugging and Visualization

Martin Hentschel, Reiner Hähnle, and Richard Bubel

11.1 Introduction

Out of the four papers ([Burstall, 1974, Boyer et al., 1975, Katz and Manna, 1975, King, 1976]) that independently introduced symbolic execution as a program analysis technique during the mid 1970s, no less than three mention debugging as a motivation. Indeed, symbolic execution has a number of natural properties that make it attractive in helping to debug programs:

- A time-consuming task for users of classical interactive debuggers is to set up a (small) initial program state which leads to an execution that exhibits the failure. It is usually nontrivial to build the required, complex data structures. Symbolic program execution, on the other hand, permits to execute any method or any statement directly without setting up an initial state. This is possible by using symbolic values instead of concrete ones. The capability to start debugging from any code location makes it also easy to debug incomplete programs.
- Not only is it time-consuming to build concrete initial states, it is often also difficult to determine under which exact conditions a failure will occur. This can be addressed by symbolic execution, which allows one to specify initial states only partially (or not at all) and which generates *all* reachable symbolic states up to a chosen depth.
- Classical debuggers typically pass through a vast number of program states with possibly large data structures before interactive debugging mode is entered. Once this happens, it is often necessary to visit previous states, which requires to implement reverse (or omniscient) debugging, which is nontrivial to do efficiently, see [Pothier et al., 2007]. In a symbolic execution environment reverse debugging causes only little overhead, because (a) symbolic execution can be started immediately in the code area where the defect is suspected and (b) symbolic states are small and contain only program variables encountered during symbolic execution.
- The code instrumentation typically required by standard debuggers can make it impossible to observe a failure that shows up in the unaltered program (so-called

"Heisenbugs," see [Gray, 1985]). This can be avoided by symbolic execution of the unchanged code.

The question is then why—given these advantages of symbolic execution, plus the fact that the idea to combine it with debugging has been around for 40 years—all widely used debuggers are still based on interpretation of programs with concrete start states. Stable Mainstream debugging tools evolved slowly and their feature set remained more or less stable in the last decades, providing mainly the standard functionality for step-wise execution, inspection of the current program state, and suspension of the execution before a marked statement is executed. This is all the more puzzling, since debugging is a central, unavoidable, and time-consuming task in software development with an accordingly huge saving potential.

The probable answer is that, until relatively recently, standard hardware simply was insufficient to realize a debugger based on symbolic execution for real-world programming languages. On a closer look, there are three aspects to this. First, symbolic execution itself: reasonably efficient symbolic execution engines for interesting fragments of real-world programming languages are available only since ca. 2006 (for example, [Beckert et al., 2007, Grieskamp et al., 2006, Jacobs and Piessens, 2008]). Second, and this is less obvious, to make good use of the advantages of symbolic execution pointed out above, it is essential to *visualize* symbolic execution paths and symbolic states and navigate through them. Otherwise, the sheer amount and the symbolic character of the generated information make it impossible to understand what is happening. Again, high-quality visual rendering and layout of complex information was not possible on standard hardware in real-time until a few years ago. The third obstacle to adoption of symbolic execution as a debugging technology is lack of integration. Developers expect that a debugger is smoothly integrated into the development environment of their choice, so that debugging, editing, testing, and documenting activities can be part of a single workflow without breaking the tool chain.

These issues were for the first time addressed in a prototypic symbolic state debugger by Hähnle et al. [2010]. However, that tool was not very stable and its architecture was tightly integrated into the KeY system. As a consequence, the *Symbolic Execution Debugger* (SED) [Hentschel et al., 2014a] presented in this chapter was completely rewritten, much extended and realized as a reusable Eclipse extension.

The SED extends the Eclipse debug platform by symbolic execution and visualization capabilities. Although different symbolic execution engines can be integrated into the SED platform, we will consider in the following only the integration of KeY as symbolic execution engine. In contrast to the KeY verifier, the SED can be used without any specialist knowledge, exactly like a standard debugger. To make full usage of its capabilities, however, it is of advantage to know the basic concepts of symbolic execution. To make the chapter self-contained we give a short introduction into symbolic execution and to our notion of a symbolic execution tree in Section 11.2. The debugging and visualization capabilities of SED are explained in tutorial style in Section 11.3. We show how to employ the SED profitably in various use cases, including tracking the origin of failures, help in program understanding, and even actual program verification. We also explain its architecture, which has a highly modular design and allows other symbolic execution engines than KeY to be integrated into SED. How KeY is employed in the SED, and which technical features are necessary, is the topic of the final Section 11.4.

The reader who only wants to know how the SED is used and is not interested in its realization can safely skip Section 11.3.7 and Section 11.4.

11.2 Symbolic Execution

In this section we explain symbolic execution and our notion of a symbolic execution tree by way of examples.

Listing 11.1 shows Java method `min`, which computes the minimum of two given integers. When the method is called during a concrete execution, the variables `x` and `y` have defined values. The `if` statement can compare these values and decide to execute either the then or the else block. Concrete execution always follows exactly one path trough a (sequential) program. To explore different paths it is required to execute the program multiple times with different input values.

```java
public static int min(int x, int y) {
   if (x < y) {
      return x;
   }
   else {
      return y;
   }
}
```

Listing 11.1 Minimum of two integers

Symbolic execution uses symbolic in lieu of concrete values, so that when method `min` is called, variables `x` and `y` are assigned symbolic values x and y. As long as nothing is known about the relation of x and y, the `if` statement cannot decide whether to follow the then or the else branch. Consequently, symbolic execution has to split to follow both branches, resulting in a symbolic execution tree. One branch continues the execution in case that the *branch condition* $x < y$ is fulfilled and the other in case that $!(x < y)$ holds instead. The conjunction over all parent branch conditions is named *path condition* and defines a constraint on the input values that ensures this path to be taken. The knowledge gained from branch conditions is used in subsequent symbolic execution steps to prune infeasible execution paths. If method `min` is called a second time with the same symbolic values x and y and with one of the possible branch conditions from the first call, then symbolic execution will not split again. In this way symbolic execution discovers all feasible execution paths and each symbolic path may represent infinitely many concrete executions.

The complete symbolic execution tree of method `min` is shown in Figure 11.1. The root of each symbolic execution tree in our notion of symbolic execution is a *start node*, usually followed by a call of the method to execute.

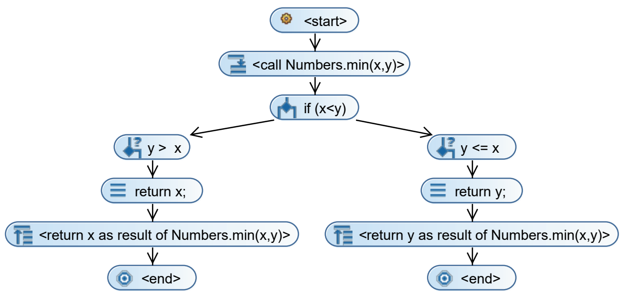

<start>
<call Numbers.min(x,y)>
if (x<y)
y > x
y <= x
return x;
return y;
<return x as result of Numbers.min(x,y)>
<return y as result of Numbers.min(x,y)>
<end>
<end>

Figure 11.1 Symbolic execution tree of static method `min` defined in class `Numbers`

Typically, an `if` statement splits execution. For this reason it is represented as a *branch statement*. Its child nodes are *branch conditions* representing the condition when a branch is taken. Branch conditions occur after branch statements if and only if execution splits. If a branch statement does not split, then its child is the next statement to execute. But also other statements than explicit branch statements can split execution, for instance, an object access that may throw a `NullPointerException`. Whenever a statement splits execution, its children show the relevant branch conditions and continue execution.

In the example, on each branch a return statement is executed which causes a method return and lets the program terminate normally (without an uncaught exception).

Loop statements are unwound by default, similar to a concrete program execution. The first time when a loop is entered it is represented as a *loop statement* in the symbolic execution tree. Whenever the loop guard is executed, it will be represented as a *loop condition* node and may split execution into two branches. One where the guard is false and the execution is continued after the loop and one where it is true and the loop body is executed once and the loop guard is checked again. As a consequence, unwinding a loop can result in symbolic execution trees of unbounded depth. As an illustration we use the method in Listing 11.2 which computes sum of array elements.

The beginning of a symbolic execution tree resulting from execution of `sum` with precondition `array != null` is shown in Figure 11.2. The left branch stops before the loop guard is evaluated the second time, whereas the right branch terminates after the computed sum is returned. When symbolic execution is continued on the left branch, similar child branches will be created until `Integer.MAX_VALUE` is reached.

To render symbolic execution trees finite in presence of loops, optionally, a loop invariant can be supplied [Hentschel et al., 2014b]. In this case a *loop invariant* node is shown in the symbolic execution tree splitting execution into two branches. The

```
public static int sum(int[] array) {
   int sum = 0;
   for (int i = 0; i < array.length; i++) {
      sum += array[i];
   }
   return sum;
}
```

Listing 11.2 Sum of all array elements

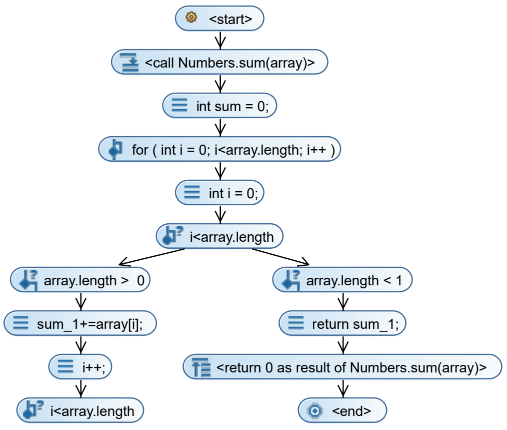


Figure 11.2 Symbolic execution tree of static method `sum` defined in class `Numbers`

first *body preserves invariant* branch represents all possible loop iterations ending in *loop body termination* nodes.[1] The second *use case* branch continues execution after the loop. It is possible that the invariant was initially not valid or that it is not preserved by the loop body. This would be a problem in a verification scenario, but a violated loop invariant should not stop one from debugging a program. Therefore, different icons indicate whether the loop invariant holds initially and in a *loop body termination* node.

The sum example in Listing 11.2 is extended by a weak (and wrong) loop invariant in Listing 11.3. A correct loop invariant would treat the case that `i` can be zero. For verification it is also required to specify how the value of `sum` is changed by the loop.

The resulting symbolic execution tree using the loop invariant and precondition `array != null` is shown in Figure 11.3. The icon of the loop invariant indicates that it is initially not fulfilled.

Method calls are handled by default by inlining the body of the called method. In case of inheritance, symbolic execution splits to cover all possible implementations indicated by *branch condition* nodes in front of the *method call* node.

[1] In case an exception is thrown or a jump outside of the loop is initiated by a `return`, `break` or `continue` statement, execution is continued directly in the *body preserves invariant* branch.

```
/*@ loop_invariant i > 0 && i <= array.length;
  @ decreasing array.length - i;
  @ assignable \strictly_nothing;
  @*/
for (int i = 0; i < array.length; i++) { /* ... */ }
```

Listing 11.3 Wrong and weak loop invariant of loop from Listing 11.2

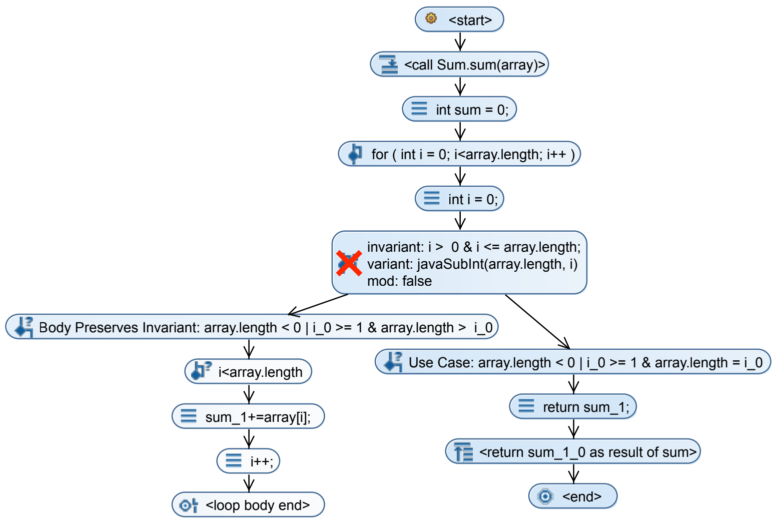

<start>
<call Sum.sum(array)>
int sum = 0;
for (int i = 0; i<array.length; i++)
int i = 0;
invariant: i > 0 & i <= array.length;
variant: javaSubInt(array.length, i)
mod: false
Body Preserves Invariant: array.length < 0 | i_0 >= 1 & array.length > i_0
i<array.length
sum_1+=array[i];
i++;
<loop body end>
Use Case: array.length < 0 | i_0 >= 1 & array.length = i_0
return sum_1;
<return sum_1_0 as result of sum>
<end>

Figure 11.3 Symbolic execution tree of static method `sum` using a loop invariant

The usage of inlined methods is explained with help of the example in Listing 11.4 which executes in method `run` of class `Main` the `run` method of an `IOperation`. Two different `IOperation` implementations are available.

The resulting symbolic execution tree under precondition `operation != null` is shown in Figure 11.4. The target method is inlined first and its body is executed between the *method call* and the corresponding *method return* node. The only statement calls method `run` on the argument `operation`. As the concrete implementation is unknown, symbolic execution has to split to consider both of them, shown by the child *branch condition* nodes. The left branch continues execution in case that `operation` is an instance of `BarOperation` and the right one in the other case. Both branches inline the target method next, execute the return statement, return from the called method, and finally terminate normally.

As in the case of loops, recursive method calls can lead to unbounded symbolic execution trees. But even unfolding nonrecursive calls can quickly lead to infeasibly large symbolic execution trees. To address this issue, instead of inlining the method body, it is possible to replace a method call by a *method contract* (see Chapter 7). This can also be useful when the source code of a method implementation is not available (for example, if it is proprietary code or simply unfinished).

```
public class Main {
   public static String run(IOperation operation) {
      return operation.run();
   }
}

interface IOperation {
   public String run();
}

class FooOperation implements IOperation {
   public String run() {
      return "foo";
   }
}

class BarOperation implements IOperation {
   public String run() {
      return "bar";
   }
}
```

Listing 11.4 Method call with inheritance

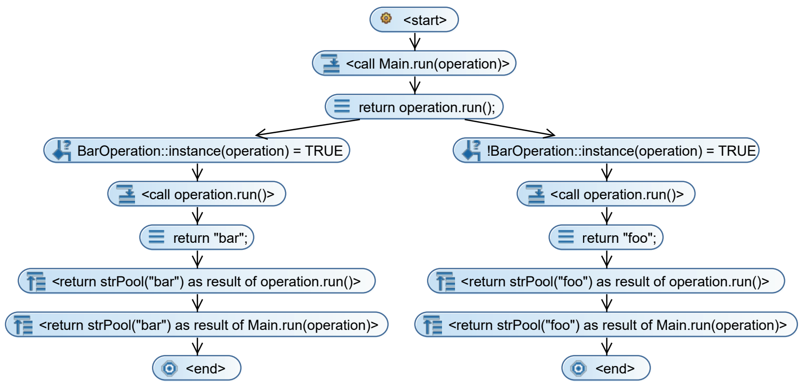

<start>
<call Main.run(operation)>
return operation.run();
BarOperation::instance(operation) = TRUE
!BarOperation::instance(operation) = TRUE
<call operation.run()>
return "bar";
<return strPool("bar") as result of operation.run()>
<return strPool("bar") as result of Main.run(operation)>
<end>
<call operation.run()>
return "foo";
<return strPool("foo") as result of operation.run()>
<return strPool("foo") as result of Main.run(operation)>
<end>

Figure 11.4 Symbolic execution tree of static method `run`

Upon application of a method contract, symbolic execution is continued separately for the specification cases corresponding to normal and to exceptional behavior. As in the case of loop invariants, node icons are used to indicate if certain conditions like preconditions or that the callee is not **`null`** could not be established.

Listing 11.5 shows the contract of method `sum` from Listing 11.2. The `sum` method is used to compute the average of all array elements in Listing 11.6.

The symbolic execution tree resulting from the execution of method `average`, where the contract of `sum` is used to handle the call to `sum`, is shown in Figure 11.5. The left branch terminates with an uncaught `ArithmeticException` in case that

```
/*@ normal_behavior
  @ requires array != null;
  @ ensures \result == (\sum int i; i >= 0 && i < array.length; array[i]);
  @
  @ also
  @
  @ exceptional_behavior
  @ requires array == null;
  @ signals_only NullPointerException;
  @ signals (NullPointerException) true;
  @*/
public static /*@ pure @*/ int sum(/*@ nullable @*/ int[] array) {
   // ...
}
```

Listing 11.5 Method contract of method `sum` from Listing 11.2

```
public static int average(/*@ nullable @*/ int[] array) {
   return sum(array) / array.length;
}
```

Listing 11.6 Average of all array elements

the array is empty whereas the middle branch terminates normally after the computed average is returned. The right branch terminates with an uncaught `Throwable` in case the array is **`null`**.

Table 11.1 summarizes the different nodes which are used in our notion of a symbolic execution tree. Readers familiar with the Eclipse IDE will notice that the icons in start and statement nodes are compatible with Eclipse usage.

11.3 Symbolic Execution Debugger

The Symbolic Execution Debugger with KeY as symbolic execution engine allows the user to execute any Java method or any Java statement(s) symbolically resulting in a symbolic execution tree as discussed in Section 11.2. The main goal of the tool is to help program understanding. Like a traditional debugger it allows the user to control the execution, to inspect states and to suspend execution at defined breakpoints.

11.3.1 Installation

The Symbolic Execution Debugger and other Eclipse extensions provided by the KeY project can be added to an existing Eclipse installation via an update-site. The supported Eclipse versions and the concrete update-site URLs are available on the

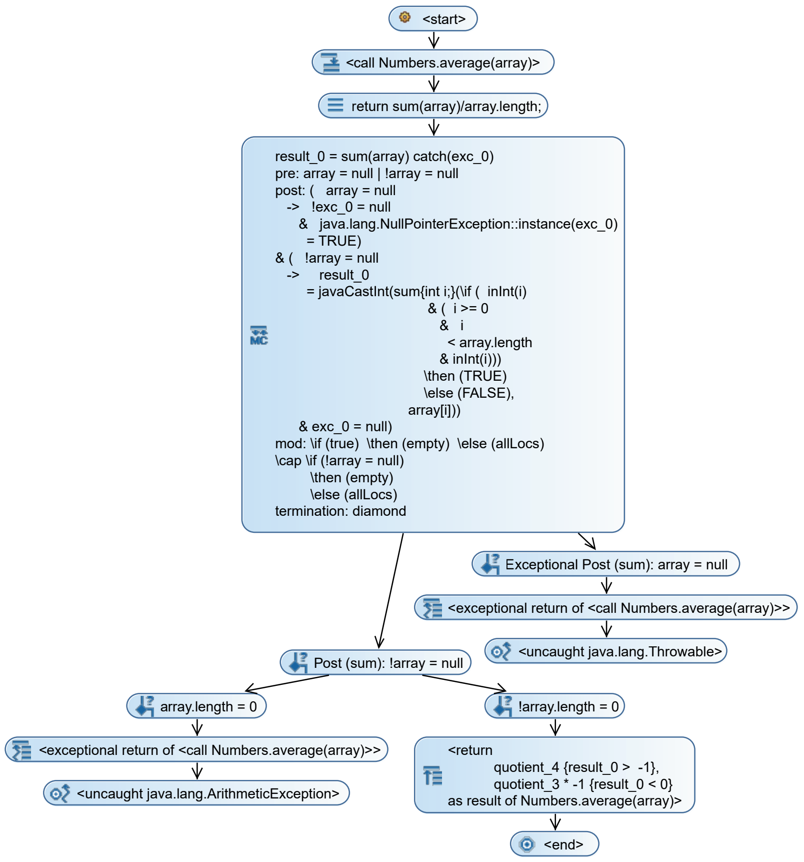

<start>
<call Numbers.average(array)>
return sum(array)/array.length;
result_0 = sum(array) catch(exc_0)
pre: array = null | !array = null
post: (array = null
-> !exc_0 = null
& java.lang.NullPointerException::instance(exc_0)
= TRUE)
& (!array = null
-> result_0
= javaCastInt(sum{int i;}(\if (inInt(i)
& (i >= 0
& i
< array.length
& inInt(i)))
\then (TRUE)
\else (FALSE),
array[i]))
& exc_0 = null)
mod: \if (true) \then (empty) \else (allLocs)
\cap \if (!array = null)
\then (empty)
\else (allLocs)
termination: diamond
Exceptional Post (sum): array = null
<exceptional return of <call Numbers.average(array)>>
<uncaught java.lang.Throwable>
Post (sum): !array = null
array.length = 0
!array.length = 0
<exceptional return of <call Numbers.average(array)>>
<uncaught java.lang.ArithmeticException>
<return
quotient_4 {result_0 > -1},
quotient_3 * -1 {result_0 < 0}
as result of Numbers.average(array)>
<end>

Figure 11.5 Symbolic execution tree of method `average` using a contract for the called method

KeY website (www.key-project.org). When reading the following sections for the first time, we strongly recommend to have a running Eclipse installation with the SED extension at hand, so that the various features can be tried out immediately. We assume that the reader is familiar with the Java perspective of the Eclipse IDE.

11.3.2 Basic Usage

The main use case of the SED using KeY is to execute a Java method symbolically. It can be achieved by opening the context menu of a method and by selecting **Debug As, Symbolic Execution Debugger (SED)**. Alternatively, it is possible to execute individual Java statements by selecting them first in the Java text editor and then by selecting the same context menu entry. Additional knowledge to limit feasible execution paths

Table 11.1 Symbolic execution tree nodes

Icon & Node Type	Description
Start	The root of a symbolic execution tree.
Branch Statement	The program state before a branch statement (`if` and `switch` in Java) is executed.
Loop Statement	The program state before a loop (`while`, `do`, `for` and for-each loop in Java) is executed. It occurs only once when the loop is entered the first time.
Loop Condition	The program state before a loop condition is executed. It is repeated in every loop iteration.
Statement	The program state before a statement which is not a branch statement, loop statement and loop condition is executed.
Branch Condition	The condition under which a branch is taken.
Termination	The last node of a branch indicating that the program terminates normally. If the postcondition does not hold icon is used instead.
Exceptional Termination	The last node of a branch indicating that the program terminates with an uncaught exception. If the postcondition does not hold icon is used instead.
Method Call	The event that a method body is inlined and will be executed next.
Method Return	The event that a method body is completely executed. Execution will be continued in the caller of the returned method.
Exceptional Method Return	The event that a method returns by throwing an exception. Execution will be continued where the exception is caught. Otherwise, execution finishes with an exceptional termination node.
Method Contract	A method contract is applied to treat a method call. If the object on which the method is called can be `null`, icon is used instead. If the precondtion does not hold, icon shows this circumstance. If both do not hold, icon is used.
Loop Invariant	A loop invariant is applied to treat a loop. If it is initially not fulfilled the icon is used instead.
Loop Body Termination	The branch of a loop invariant node which executes only loop guard and loop body once is completed. If the loop invariant does not hold, the icon is used instead.

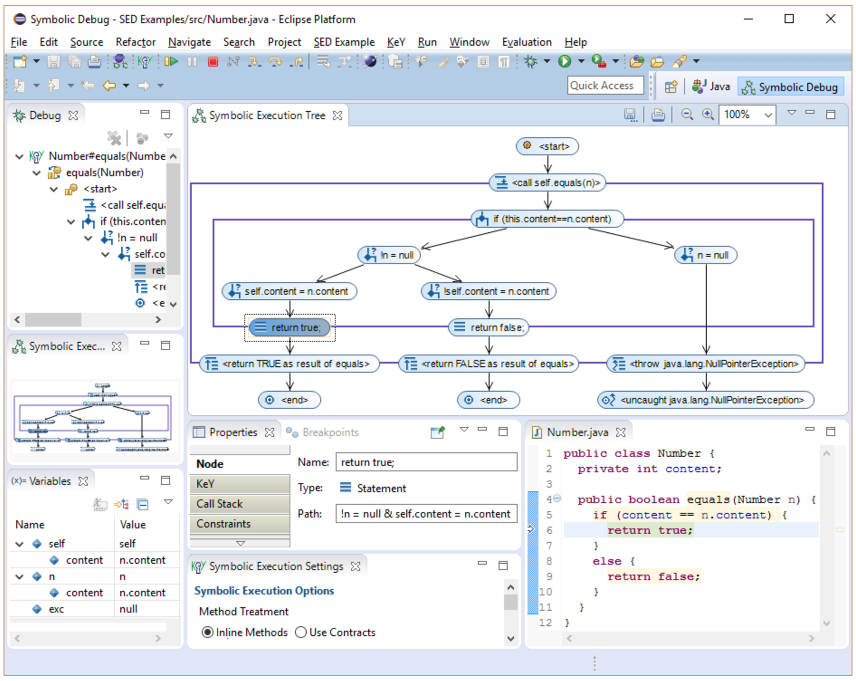
can be supplied as a precondition in the *Debug Configuration*. Also a full method contract can be selected instead of specifying a precondition.[2] In this case icons of termination nodes will indicate whenever the postcondition is not fulfilled. After starting execution, it is recommend to switch to the perspective **Symbolic Debug** which contains all relevant views explained in Table 11.2.

Figure 11.6 shows a screenshot of the **Symbolic Debug** perspective in which the symbolic execution tree of method `equals`, whose implementation is shown in the bottom right editor, is visualized. The method checks whether its `Number` argument instance has the same content as `this`, which is named `self` in KeY. The left branch represents the case when both instances have the same content, whereas the content is different in the middle branch. The right branch terminates with an uncaught `NullPointerException`, because the argument is `null`.

[2] The use of a method contract activates full JML support including `non_null` defaults.

Table 11.2 Views of perspective Symbolic Debug

View	Description
Debug	Shows symbolic execution trees of all launches, as well as to switch between them and to control execution.
Symbolic Execution Tree	Visualizes symbolic execution tree of selected launch.
Symbolic Execution Tree (Thumbnail)	Miniature view of the symbolic execution tree for navigation purposes.
Variables	Shows the visible variables and their symbolic values.
Breakpoints	Manages the breakpoints.
Properties	Shows all information of the currently selected element.
Symbolic Execution Settings	Customizes symbolic execution, e.g., defines how to treat method calls and loops.


Symbolic Debug - SED Examples/src/Number.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project SED Example KeY Run Window Evaluation Help
Quick Access
Java
Symbolic Debug
Debug
Number#equals(Numbe
equals(Number)
<start>
<call self.equ
if (this.conten
!n = null
self.co
ret
<re
<e
Symbolic Execution Tree
100%
<start>
<call self.equals(n)>
if (this.content==n.content)
!n = null
n = null
self.content = n.content
!self.content = n.content
return true;
return false;
<return TRUE as result of equals>
<return FALSE as result of equals>
<throw java.lang.NullPointerException>
<end>
<end>
<uncaught java.lang.NullPointerException>
Symbolic Exec...
Variables
Properties
Breakpoints
Number.java
Symbolic Execution Settings

Name	Value
self	self
content	n.content
n	n
content	n.content
exc	null

Node
KeY
Call Stack
Constraints

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

```
public class Number {
  private int content;

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

Symbolic Execution Options
Method Treatment
[✓ Inline Methods] Use Contracts

Figure 11.6 Symbolic Execution Debugger: Interactive symbolic execution

The additional frames (rectangles) displayed in view **Symbolic Execution Tree** of Figure 11.6 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 conditions of that particular path), displaying the constraint under which the end of the corresponding code block is reached. In Figure 11.7, the method call node is collapsed. Collapsed frames are colored green, if all execution paths reached the end of the frame. Otherwise they are colored orange. Expanded frames are colored blue.

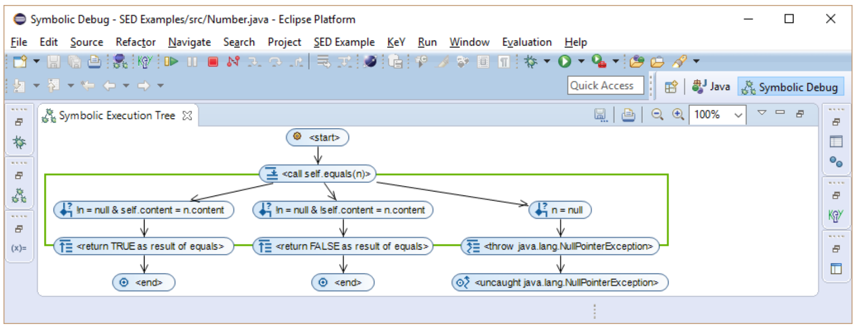

Symbolic Debug - SED Examples/src/Number.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project SED Example KeY Run Window Evaluation Help
Quick Access
Java
Symbolic Debug
Symbolic Execution Tree
100%
<start>
<call self.equals(n)>
!n = null & self.content = n.content
!n = null & !self.content = n.content
n = null
<return TRUE as result of equals>
<return FALSE as result of equals>
<throw java.lang.NullPointerException>
<end>
<end>
<uncaught java.lang.NullPointerException>

Figure 11.7 Symbolic Execution Debugger: Collapsed frame (frame color is green)

The symbolic program state of a selected node is shown in the view **Variables**. The details of a selected variable (e.g. additional constraints) or symbolic execution tree node (e.g. path condition, call stack, etc.) are available in the **Properties** view. The source code line corresponding to the selected symbolic execution 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 symbolic execution, e.g., one can choose between method inlining and method contract application. Breakpoints suspend the execution and are managed in the **Breakpoints** view.

In Figure 11.6 the symbolic execution tree node `return true;` is selected, which is indicated by a darker color. The symbolic value of field `content` of the current instance `self` and of the argument instance `n` are identical. This is not surprising, because this is exactly what is enforced by the path condition. A fallacy and source of defects 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. This is because 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 a symbolic execution tree node creates a visualization of possible memory layouts as a *symbolic object diagram*, see Figure 11.8. It resembles a UML object diagram and shows (i) the dependencies between objects, (ii) the symbolic values of object fields and (iii) the symbolic values of 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 Figure 11.8, the local variables `n` and `self` refer to objects visualized as rectangles. The symbolic value of the instance field `content` 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 scrollbar of the toolbar (near the origin of the callout) allows one 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. Figure 11.8 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.

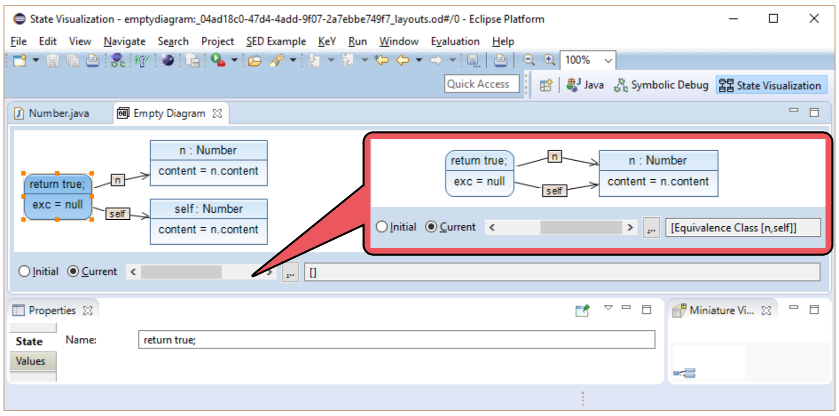


Figure 11.8 Symbolic Execution Debugger: Possible memory layouts of a symbolic state

11.3.3 Debugging with Symbolic Execution Trees

The Symbolic Execution Debugger allows one to control execution like a traditional debugger and can be used in a similar way. A major advantage of symbolic execution is that it is not required to start at a predefined program entry point and to run the program until the point of interest is reached. Instead, the debug session can start directly at the point of interest. This avoids building up large data structures and the memory will contain only the variables used by the code of interest. If knowledge about the conditions under which a failure can be observed is available, it can be given as a precondition to limit the number of explored execution paths.

The main task of the user is, like in a traditional debugger, to control execution and to comprehend each performed step. It is helpful to focus on a single branch where the execution is expected to reach a faulty state. If this is not the case, the focus can be changed to a different branch. There is no need for a new debugging session or to find new input values resulting in a different execution path. It is always possible to go back to previous steps, because each node in the symbolic execution tree provides the full symbolic state.

Of special interest are splits, because their explicit rendering in the symbolic execution tree constitutes a major advantage of the SED over traditional debuggers. Unexpected splits or missing expected splits are good candidates for possible sources of defects. This is explained by example. Listing 11.7 shows a defective part of a *Mergesort* implementation for sorting an array called `intArr`. The exception shown in Listing 11.8 was thrown during a concrete execution of a large application that contained a call to `sort`. It seems that method `sortRange` calls itself infinitely often in line 9 until the call stack is full, which happened in line 7.

Either the value of `l` or the value of `r` is the termination criterion. Using a traditional debugger the user has to execute the whole program with suitable input values until method `sort` is executed. From this point onward, she may control the execution, observe how the `r` value is computed and try to find the origin of the failure. With the SED, however, she can start execution directly at method `sort`. Clearly, the array `intArr` needs to be not `null`. This knowledge can be expressed as precondition `intArr != null`. The resulting symbolic execution tree in Figure 11.9 shows already after a few steps that the `if`-statement is not branching in case that `intArr` is not empty and the defect is found (the comparison should have been `l < r`).

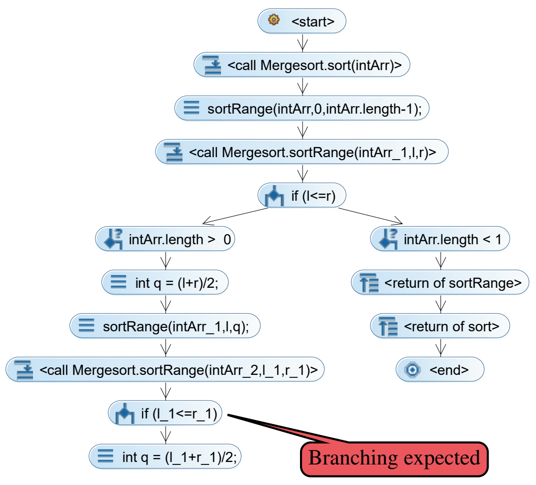


Figure 11.9 Symbolic Execution Tree of the mergesort implementation in Listing 11.7

[3] Modified version of the Mergesort implementation by Jörg Czeschla, see javabeginners.de/Algorithmen/Sortieralgorithmen/Mergesort.php

```
public class Mergesort {
   public static void sort(int[] intArr) {
      sortRange(intArr, 0, intArr.length - 1);
   }

   public static void sortRange(int[] intArr, int l, int r) {
      if (l <= r) {
         int q = (l + r) / 2;
         sortRange(intArr, l, q);
         sortRange(intArr, q + 1, r);
         merge(intArr, l, q, r);
      }
   }

   private static void merge(int[] intArr, int l, int q, int r) {
      int[] arr = new int[intArr.length];
      int i, j;
      for (i = l; i <= q; i++) {
         arr[i] = intArr[i];
      }
      for (j = q + 1; j <= r; j++) {
         arr[r + q + 1 - j] = intArr[j];
      }
      i = l;
      j = r;
      for (int k = l; k <= r; k++) {
         if (arr[i] <= arr[j]) {
            intArr[k] = arr[i];
            i++;
         }
         else {
            intArr[k] = arr[j];
            j--;
         }
      }
   }
}
```

Listing 11.7 Defective part of a mergesort implementation[3]

11.3.4 Debugging with Memory Layouts

It is easy to make careless mistakes in operations which modify data structures. To find them with a traditional debugger can be time consuming, because large data structures have to be inspected after each execution step. A complication is that a program state contains not only the data structure of interest, but all information computed before the state of interest is reached. Traditional debuggers present the current state typically as variable-value pairs in a list or tree. This representation makes it very hard to figure out object type data structures.

```
Exception in thread "main" java.lang.StackOverflowError
        at Mergesort.sortRange(Mergesort.java:7)
        at Mergesort.sortRange(Mergesort.java:9)
        at Mergesort.sortRange(Mergesort.java:9)
        at Mergesort.sortRange(Mergesort.java:9)
        ...
```

Listing 11.8 Exception thrown by the mergesort implementation of Listing 11.7

With the Symbolic Execution Debugger it is possible to visualize the current state as well as the initial state from which the execution started in the form of a symbolic object diagram. As an example, consider the rotate left operation of an AVL tree. Each node in such a tree has a left and a right child and it knows its parent as well. Again, symbolic execution is started directly in the method of interest, here the `rotateLeft` method and we let the SED compute all memory layouts for one of its return nodes.

Consider the initial state in Figure 11.10. The node to rotate is named `current` and it is the root of the tree because its parent is **`null`**. It has a right child, which in turn has a left child. The AVL tree itself is named `self`. Additionally, precondition `current != `**`null`**` && current.right != `**`null`** is used to ensure that the nodes to rotate exist.

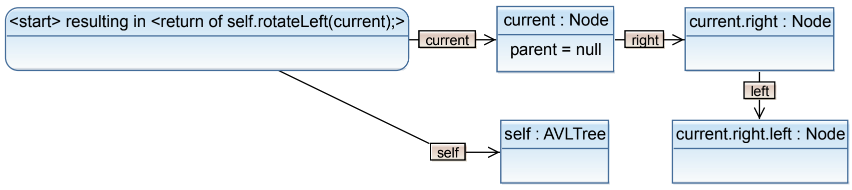

<start> resulting in <return of self.rotateLeft(current);>
current
current : Node
parent = null
right
current.right : Node
left
self
self : AVLTree
current.right.left : Node

Figure 11.10 Initial symbolic object diagram of an AVL Tree rotate left operation

The symbolic state automatically computed and visualized by the SED after performing the rotation is shown in Figure 11.11. It shows the initial objects with all performed changes. By inspecting this diagram it is obvious that the parent of object `current.right.left` was not correctly updated because its parent is now the node itself.

11.3.5 Help Program and Specification Understanding

An important feature of symbolic execution trees is that they 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

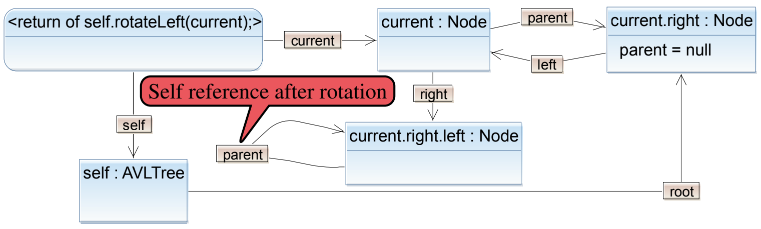


Figure 11.11 Current symbolic object diagram of an AVL Tree rotate left operation

works best, when some method contracts and/or invariants are available to achieve compact and finite symbolic execution trees. However, useful specifications can be much weaker than would be required for verification.

For example, Listing 11.9 shows a defective implementation of method `indexOf` which should return the first array index excepted by the given `Filter` or `-1` in case none of the array elements were accepted. The method is specified by a basic method contract limiting the expected input values. In addition, a very simple loop invariant is given.

The corresponding symbolic execution tree is shown in Figure 11.12. It captures the full behavior of `indexOf`. Without checking any details, one can see that the left-most 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 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.

As this example demonstrates, symbolic execution trees can be used to answer questions, for example, about thrown exceptions (none in the example) or returned values. Within the 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 layout). Using breakpoints, symbolic execution is continued until a breakpoint is hit on any branch. Thus it can be used to find execution paths (i) throwing a specified exception, (ii) accessing or modifying a specified field, (iii) calling or returning a specified method or (iv) causing a specified state.

11.3.6 Debugging Meets Verification

As the SED is based on symbolic execution it actually *verifies* the target program for the contract specified in the debug configuration. The program was proven correct if and only if each branch in the symbolic execution tree ends with a termination

```
public class ArrayUtil {
    /*@ normal_behavior
      @ requires \invariant_for(filter);
      @*/
    public static int /*@ strictly_pure @*/ indexOf(Object[] array,
                                                    Filter filter) {
        int index = -1;
        int i = 0;
        /*@ loop_invariant i >= 0 && i <= array.length;
          @ decreasing array.length - i;
          @ assignable \strictly_nothing;
          @*/
        while (index < 0 && i < array.length) {
            if (filter.accept(array[i])) {
                index = i;
            }
            else {
                i++;
            }
        }
        return i;
    }

    public static interface Filter {
        /*@ normal_behavior
          @ requires true;
          @ ensures true;
          @*/
        public boolean /*@ strictly_pure @*/ accept(/*@ nullable @*/
                                                    Object object);
    }
}
```

Listing 11.9 A defective and only partially specified implementation

node and no warning icons are raised in the whole tree. This means that all branches terminate in a state where the specified postcondition is fulfilled. If a method call is approximated by a method contract, the precondition- and caller-no-null checks must have been successful, too. Likewise, all applied loop invariants are valid at the start of their loop and are preserved by the loop body.

Whereas a proof tree in KeY shows all performed steps during the proof, including intermediate steps of symbolic execution and proofs of first-order verification conditions, a symbolic execution tree contains only nodes that correspond to reachable program states. Hence, the debugger provides a view on a KeY proof from the developer's perspective, hiding intermediate and nonprogram related steps. Program states are visualized in a user-friendly way and are not encoded into side formulas of sequents.

Another advantage of SED over the KeY system is that insufficient or wrong specifications are directly highlighted. Whenever a symbolic execution tree node

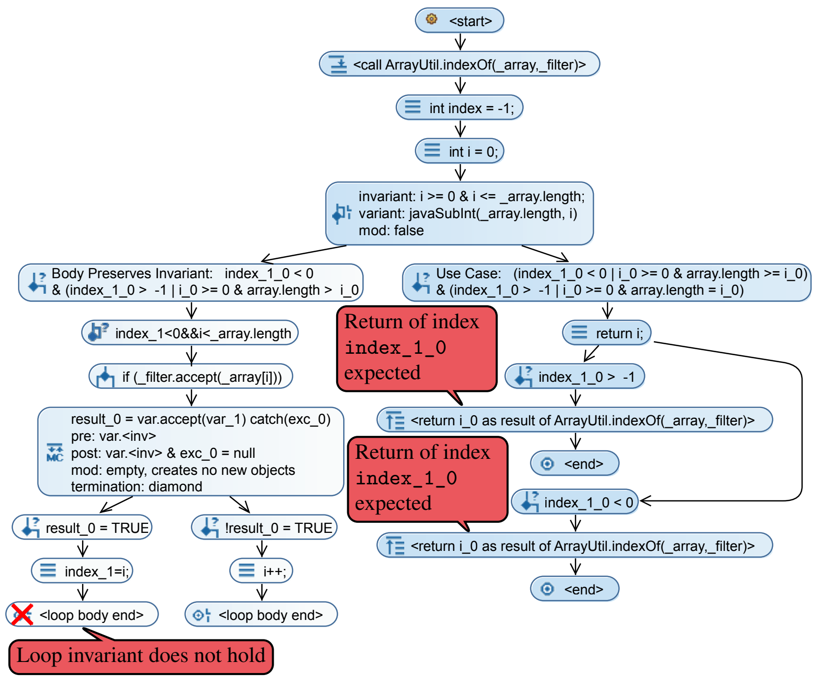

<start>
<call ArrayUtil.indexOf(_array,_filter)>
int index = -1;
int i = 0;
invariant: i >= 0 & i <= _array.length;
variant: javaSubInt(_array.length, i)
mod: false
Body Preserves Invariant: index_1_0 < 0 & (index_1_0 > -1 | i_0 >= 0 & array.length > i_0)
Use Case: (index_1_0 < 0 | i_0 >= 0 & array.length >= i_0) & (index_1_0 > -1 | i_0 >= 0 & array.length = i_0)
index_1<0&&i<_array.length
if (_filter.accept(_array[i]))
result_0 = var.accept(var_1) catch(exc_0)
pre: var.<inv>
post: var.<inv> & exc_0 = null
mod: empty, creates no new objects
termination: diamond
result_0 = TRUE
!result_0 = TRUE
index_1=i;
i++;
<loop body end>
<loop body end>
Loop invariant does not hold
Return of index index_1_0 expected
return i;
index_1_0 > -1
<return i_0 as result of ArrayUtil.indexOf(_array,_filter)>
<end>
Return of index index_1_0 expected
index_1_0 < 0
<return i_0 as result of ArrayUtil.indexOf(_array,_filter)>
<end>

Figure 11.12 Symbolic execution tree of method `indexOf` (see Listing 11.9)

is crossed out, then something went wrong in proving the verification conditions for that path. The user can then inspect the parent nodes and check whether the implementation or the specifications contain a defect. More specifically, if the post-condition in a termination node is not fulfilled, then the symbolic program state at that point should be inspected. Wrong values relative to the specified behavior indicate a defect in the implementation. Values that have been changed as expected, but which are not mentioned in the specification indicate that the specification has to be extended. Moreover, crossed out method call and loop invariant nodes indicate that the precondition of the proven method contract is too weak or that something went wrong during execution. If a loop invariant is not preserved, the state of the loop body at the termination nodes gives hints on how to adjust the loop invariant.

11.3.7 Architecture

The Symbolic Execution Debugger (SED) is an Eclipse extension and can be added to existing Eclipse-based products. In particular, SED is compatible with the Java Development Tools (JDT) that provide the functionality to develop Java applications in Eclipse. To achieve this and also a user interface that seamlessly integrates with Eclipse, SED needs to obey a certain architecture, which is shown in Figure 11.13. The gray-colored components are part of the Eclipse IDE, whereas the remaining components are part of the SED extension.

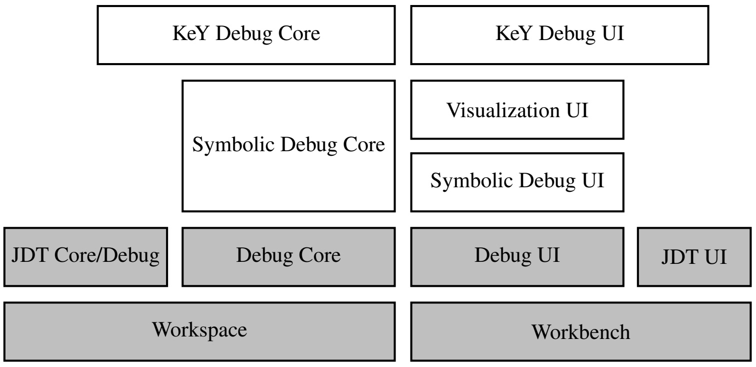


Figure 11.13 Architecture of the Symbolic Execution Debugger (SED)

The foundation is the Eclipse Workspace which provides resources such as projects, files and folders, and the Eclipse Workbench which provides the typical Eclipse user interface with perspectives, views and editors. Eclipse implements on top of these the debug platform which defines language-independent features and mechanisms for debugging. Specifically, Debug Core provides a language-independent model to represent the program state of a suspended execution. This includes threads, stack frames, variables, etc. Debug UI is the user interface to visualize the state defined by the debug model and to control execution. JDT Core defines the functionality to develop Java applications, including the Java compiler and a model to represent source code, whereas JDT Debug uses the debug platform to realize the Java debugger. Finally, JDT UI provides the user interface which includes the editor for Java source files.

The Symbolic Execution Debugger is based on the components provided by Eclipse. First, it extends the debug platform for symbolic execution in general. Second, it provides a specific implementation based on KeY's symbolic execution engine, described in Section 11.4.

Symbolic Debug Core extends the debug model to represent symbolic execution trees. This is done in a way that is independent of the target programming language and of the used symbolic execution engine.[4] It is also necessary to extend the debugger user interface, which is realized in Symbolic Debug UI. It contains in particular the tree-based representation of the symbolic execution tree that is displayed in the **Debug** view. The graphical representation of the symbolic execution tree shown in

[4] Each implementation of the symbolic debug model can define new node types to represent additional language constructs not covered by Table 11.1.

the **Symbolic Execution Tree** view as well as the visualization of memory layouts is provided language-independently by Visualization UI. Finally, KeY Debug Core implements the symbolic debug model with help of KeY's symbolic execution engine (implemented as pure Java API without any dependency to Eclipse). The functionality to debug selected code and to customize the debug configuration is provided by KeY Debug UI.

The extendable architecture of SED allows one to reuse the symbolic debug model for symbolic execution to implement alternative symbolic debuggers while profiting from the visualization functionality. All that needs to be done is to provide an implementation of the symbolic debug model for the target symbolic execution engine. KeY's symbolic execution API itself is part of the KeY framework and has no dependencies to the Symbolic Execution Debugger or to Eclipse. This makes it possible to use it like any other Java API.

11.4 A Symbolic Execution Engine based on KeY

The KeY verification system (see Chapter 15) is based on symbolic execution, but it is not directly a symbolic execution engine. In this section we describe how to realize a symbolic execution engine as API based on the KeY system. It is used for instance by the Symbolic Execution Debugger (see Section 11.3). We attempted to make this section self-contained, but it is certainly useful to have read Chapter 3 in order to appreciate the details.

11.4.1 Symbolic Execution Tree Generation

All the required functionality is implemented by KeY, because it already performs symbolic execution to verify programs. The simplified[5] schema of proof obligations to verify a piece of Java `code` looks as follows in KeY:

$$\Longrightarrow \mathit{pre} \rightarrow \mathcal{U} \left\langle \begin{array}{l} \texttt{try \{}\mathit{codeOfInterest}\texttt{\}} \\ \texttt{catch (Exception e) \{exc = e\}} \end{array} \right\rangle \mathit{post}$$

The meaning is as follows: assuming precondition *pre* holds and we are in a symbolic state given by $\mathcal{U}$, then the execution of the code between the angle brackets terminates, and afterwards postcondition *post* holds. The catch-block around *codeOfInterest* is used to assign the caught exception to variable `exc` which can be used by the post condition to separate normal from exceptional termination. The code of interest is usually the initial method call but can be also a block of statements.

Rules applied on a $\langle\texttt{code}\rangle\mathit{post}$ modality rewrite the first (active) statement in `code` and then continue symbolic execution. Symbolic execution is performed at

[5] The proof obligation is explained in detail in Section 8.3.1

the level of atomic expressions, such that complex Java statements and expressions have to be decomposed before they can be executed. For example, the method call `even(2 + 3)` requires a simple argument expression, so that the sum must be computed before the method call can be performed. As a consequence, many intermediate steps might be required to execute a single statement of source code. An empty modality $\langle\rangle$*post* can be removed, and the next step will be to show that the postcondition is fulfilled in the current proof context.

All symbolic execution rules have in common that, if necessary, they will split the proof tree to cover all conceivable execution paths. This means that the rules themselves do not prune infeasible paths. It is the task of the automatic proof strategy (or the user) to check the infeasibility of new proof premisses before execution is continued.

We realize a symbolic execution engine on top of the proof search in KeY by extracting the symbolic execution tree for a program from the proof tree for the corresponding proof obligation. The main tasks to be performed are:

- Define a 'normal form' for proof trees that makes them suitable for generation of a symbolic execution tree.
- Design a proof strategy that ensures proof trees to be of the expected shape.
- Separate feasible and infeasible execution paths.
- Identify intermediate proof steps stemming from decomposition of complex statements into atomic ones. Such intermediate steps are not represented in the symbolic execution tree.
- Realize support for using specifications as an alternative to unwind loops and to inline method bodies.

It is important to postpone any splits of the proof tree caused by an attempt to show the postcondition until symbolic execution has completely finished. Otherwise, multiple proof branches representing the same symbolic execution path might be created. Whereas this does not affect the validity of a proof, it would cause redundant branches in a symbolic execution tree.

We also want to have at most one modality formula (of the form $\langle$`code`$\rangle$*post*) per sequent, otherwise it is not clear what the target of symbolic execution is. Later, we will see that to support the use of specifications, this condition has to be relaxed.

The standard proof strategy used by KeY for verification almost ensures proof trees of the required shape. It is easy to modify this strategy: first, we forbid for the moment symbolic execution rules that introduce multiple modalities; second, we stipulate that all rules not pertaining to symbolic execution and that cause splitting are only applied after finishing symbolic execution. Even with these restrictions the proof strategy is often powerful enough to close infeasible execution paths immediately.

After the strategy stops, symbolic execution tree generation takes place. During this it is required to separate proof branches representing a feasible execution path from infeasible ones. This information is not available in the proof itself, because it is not needed for proving. Complicating is also the fact that KeY throws information away that is not needed for verification, however, it might later be needed for symbolic execution tree generation. This can be easily solved with the following trick.

The uninterpreted predicate *SET* is added to the postcondition of the initial proof obligation:

$$\Longrightarrow pre \to \mathcal{U} \left\langle \begin{array}{l}\texttt{try \{}\mathit{codeOfInterest}\texttt{\}}\\ \texttt{catch (Exception e) \{exc = e\}}\end{array}\right\rangle post \land SET(\texttt{exc})$$

The effect is that infeasible paths will be closed as before and feasible paths remain open since no rules exist for the predicate *SET*. Variables of interest are listed as parameters, so KeY is not able to remove them for efficiency if no longer needed.

To separate statements that occur in the source code from statements that are introduced by decomposition we use meta data in the form of suitable tags. Each statement occurring in the source code contains position information about its source file as well as the line and column where it was parsed. Statements introduced during a proof have no such tags.

The mechanisms described above are sufficient to generate a symbolic execution tree by iterating over a given proof tree. Each node in a proof tree is classified according to the criteria in Table 11.3 and added to the symbolic execution tree. Java API methods can optionally be excluded. In this case only method calls to non-API methods are added and statement nodes are only included if they are contained in non-API methods.

Table 11.3 Classification of proof nodes for symbolic execution tree nodes (excluding specifications)

SET node type	Criterion in KeY proof tree
Start	The root of the proof tree.
Method Call	The active statement is a method body statement.
Branch Statement	The active statement is a branch statement and the position information is defined.
Loop Statement	The active statement is a loop statement, the position information is defined, and it is the first loop iteration.
Loop Condition	The active statement is a loop statement and the position information is defined.
Statement	The active statement is not a branch, loop, or loop condition statement and the position information is defined.
Branch Condition	The parent of proof tree node has at least two open children and at least one child symbolic execution tree node exist (otherwise split is not related to symbolic execution).
Normal Termination	The emptyModality rule is applied and `exc` variable has value `null`.
Exceptional Termination	The emptyModality rule is applied and `exc` variable has not value `null`.
Method Return	A rule which performs a method return by removing the current method frame is applied and the related method call is part of the symbolic execution tree.

To detect the use of specifications in the form of method contracts and loop invariants it is sufficient to check whether one of the rules UseOperationContract or LoopInvariant was applied. The problem is that specifications may contain method calls, as long as these are side effect-free (so-called query methods). During the KeY proof these give rise to additional modalities in a sequent. Hence, we must separate such 'side executions' from the target of symbolic execution. This is again done with the help of meta information. We add a so-called *term label SE* to the modality of the proof obligation, such as in:

$$\Longrightarrow pre \rightarrow \mathscr{U} \left\langle \begin{array}{l} \texttt{try \{}\mathit{codeOfInterest}\texttt{\}} \\ \texttt{catch (Exception e) \{exc = e\}} \end{array} \right\rangle (post \land SET(\texttt{exc}))\,«SE»$$

A term label is a noncorrectness relevant information attached to a term and maintained during a proof. When symbolic execution encounters a modality with an *SE* label, it will be inherited to any child modalities. It is easy to modify the KeY proof strategy to ensure that modalities without an *SE* label are executed first, because their results are required for the ongoing symbolic execution. Finally, during symbolic execution tree generation only nodes with an *SE* label are considered.

A complication is that symbolic execution of modalities without an *SE* label may cause splits in the proof tree, but the knowledge gained from their execution is used in symbolic execution of the target code. Such splits have to be reflected in the symbolic execution tree. We will discuss later in Section 11.4.3 how they can be avoided.

When a method contract is applied, two branches continue symbolic execution, one for normal and one for exceptional method return. Two additional branches check whether the precondition is fulfilled in the current state and whether the caller object is **`null`**. The latter two are proven without symbolic execution and their proof branches will be closed if successful. Boolean flags (represented as crossed out icons in the SED) on a method contract node indicate their verified status as described in Section 11.2.

The situation is similar for loop invariant application: one proof branch checks whether the loop invariant is initially (at the start of the loop) fulfilled. A Boolean flag (icon in the SED) on the loop invariant node indicates its verified status. A second branch continues symbolic execution after the loop and a third branch is used to show that the loop invariant is preserved by the loop guard and the loop body. The latter is complex, because in case an exception is thrown or that the loop terminates abnormally via a **`return`**, **`break`** or **`continue`**, the loop invariant does not need to hold. The loop invariant rule (see Section 3.7.2) of KeY solves this issue by first executing loop guard and loop body in a separate modality. If this modality terminates normally, then the proof that the loop invariant holds is initiated. Otherwise, symbolic execution is continued in the original modality, without assuming that the invariant holds. As above, the problem of multiple modalities is solved by term labels. We add a (proof global) counter to each *SE* label. The label of the original proof obligation is $SE(0)$ and it is incremented whenever needed. KeY's proof strategy is modified to ensure that symbolic execution is continued in the modality with the highest counter first.

The loop invariant rule encodes in the preserves branch whether a loop iteration terminated abnormally or normally. Depending on the kind of termination different properties have to be shown. The different cases are distinct subformulas of the form

$$\textit{reasonForTermination} \rightarrow \textit{propertyToHold}$$

We label the subformula *reasonForTermination* which characterizes the normal termination case with a *LoopInvariantNormalBehaviorTermLabel* term label[6]. If this labeled formula could be simplified to true, then a loop body termination node is added to the corresponding branch of the symbolic execution tree.

There is one special case we have not covered yet. The proof branches that check whether a loop invariant holds initially, whether a precondition holds, and whether the caller object is `null`, each can be proven without symbolic execution, as they contain no modality. This does not hold, however, when a loop invariant or a method contract is applied on the branch that shows the invariant to be preserved by loop condition and loop body. The reason is that in this case the modality which continues symbolic execution in case of an abnormal loop exit is still present and the proof strategy is free to continue symbolic execution on it. We are not interested in this execution, because it does not contribute to the verification of the actual proof obligation. Consequently, all term labels have to be removed from proof branches that only check the conditions listed above.

11.4.2 Branch and Path Conditions

Applicability of a proof rule in KeY generally depends only on the sequent it is applied to, not on other nodes in the proof tree. Consequently, KeY does not maintain branch and path conditions during proof construction, because the full knowledge gained by a split is encoded in the child nodes. A branch condition can be seen as the logical difference between the current node and its parent and the path condition is simply the conjunction over all parent branch conditions or, in other words, the logical difference between the current node and the root node.

In the case of symbolic execution rules, branch conditions are not generated from modalities to avoid a proliferation of modality formulas. Instead, splitting symbolic execution rules rewrite the active statement contained in their modality and add knowledge gained by the split in the succedent of the premisses. Consequently, branch conditions in symbolic execution trees are defined by:

$$\left(\bigwedge \textit{added antecedent formula}\right) \land \neg \left(\bigvee \textit{added succedent formula}\right)$$

Method contract and loop invariant rules are so complex that they cannot be expressed schematically in KeY with the help of taclets (see Chapter 4), but are computed. After applying a method contract the branch conditions contain the knowledge that the caller object is not `null` and that the conjunction of all preconditions (both for normal and exceptional termination) hold. The branch condition on the proof branch ensuring that an invariant is preserved is the conjunction of the loop invariant

[6] For technical reasons the label is currently around the whole implication and the analysis checks the evaluation of the left subformula (*reasonForTermination*).

and the loop guard. The branch condition on the branch that continues symbolic execution after the loop is the conjunction of the loop invariant and the negated loop guard.

11.4.3 Hiding the Execution of Query Methods

As pointed out above, the presence of query methods in specifications or in loop guards may spawn modalities that have nothing to do with the target code. These are used to compute a single value, such as a method return value or a Boolean flag. Even though their execution is hidden in the symbolic execution tree, possible splits in the proof tree caused by them are visible, because the knowledge gained from them is used during subsequent symbolic execution. Such splits complicate symbolic execution trees, so we want to get rid of them.

These modalities have in common the fact that they are top level formulas in a sequent that compute a single value *const* in the current symbolic state $\mathcal{U}$:

$$\mathcal{U}\langle\texttt{tmp = ...}\rangle \mathit{const} \doteq \texttt{tmp}$$

This computation is 'outsourced' from the main proof via a built-in rule that executes the modality in a side proof. The initial proof obligation of the side proof is:

$$\Gamma \Longrightarrow \mathcal{U}\langle\texttt{tmp = ...}\rangle \mathit{ResultPredicate}(\texttt{tmp}), \Delta$$

It executes the modality in state $\mathcal{U}$ with an uninterpreted predicate *ResultPredicate* as postcondition. That predicate is parameterized with variable `tmp`, which will be replaced during the proof by its computed value. Γ and Δ are all first-order top-level formulas of the original sequent, representing the context knowledge[7].

The standard KeY verification strategy is used in the side proof. If it stops with open goals, where no rule is applicable, the results can be used in the original sequent.[8] Each open branch in the side proof contains a result *res* as parameter of the predicate $\mathit{ResultPredicate}(\mathit{res})$ that is valid relative to a path condition *pc* (Section 11.4.2). Now for each such open branch a new top-level formula is added to the sequent from which the side proof was outsourced. If the modality with the query method was originally in the antecedent, then $\mathit{pc} \rightarrow \mathit{const} \doteq \mathit{res}$ is added to the antecedent, otherwise, $\mathit{pc} \wedge \mathit{const} \doteq \mathit{res}$ is added to the succedent. The last step is to clean up the sequent and to remove the now redundant modality of the query.

[7] In the context of this chapter, formulas containing a modality or a query are excluded from the context knowledge. Otherwise, a side proof would reason about the original proof obligation as well.

[8] The side proof is never closed, because the predicate in the postcondition is not provable. If the proof terminates, because the maximal number of rule applications has been reached, then the side proof is abandoned.

11.4.4 Symbolic Call Stack

KeY encodes the method call stack with help of method frames directly in the Java program of a modality. For each inlined method, a new method frame is added that contains the code of the method body to execute. For more details, we refer to Section 3.6.5.

During symbolic execution tree generation the symbolic call stack has to be maintained. Whenever a method call node is detected, it is pushed onto the call stack. All other nodes remove entries from the maintained call stack until its size is equal to the number of method frames in their modality.

The branch of the loop invariant rule that checks whether the loop body preserves the invariant contains multiple modalities with different call stacks. The modality that executes only the loop guard and the loop body contains only the current method frame. All parent method frames are removed. This requires to maintain a separate call stack for each counter used in *SE* term labels. Whenever a modality with a new counter is introduced, its call stack is initialized with the top entry from the call stack of the modality where the loop invariant was applied.

11.4.5 Method Return Values

Method return nodes in a symbolic execution tree that return from a method declared as nonvoid allow one to access return values.

Several proof rules are involved in a method return. Assuming that the argument of the **`return`** statement has been decomposed into a simple expression, the methodCallReturn rule executes the return statement. For this, the rule adds an assignment statement that assigns the returned value to the result variable given in the current method frame. As the result variable is then no longer needed, it is removed from the method frame. A subsequent rule executes that assignment and yet another rule completes the method return by removing the, by now, empty method frame.

According to Table 11.3 a method return node is the proof tree node that removes the current method frame, say `cmf`. At this point, however, the name of the result variable is no longer available. This requires to go back to the parent proof tree node r, where rule methodCallReturn which assigns the returned value to the result variable of `cmf` was applied.

A side proof, similar to the one in Section 11.4.3, can be performed to compute returned values and the conditions under which they are valid. The proof obligation is:

$$\Gamma \Longrightarrow \mathcal{U} \left\langle \begin{array}{l} \texttt{cmf(result->resVar, ...):} \\ \texttt{return resExp;} \end{array} \right\rangle \mathit{ResultPredicate}(\texttt{resVar}), \Delta$$

The symbolic state $\mathcal{U}$ is that of the return node r. Only the return statement is executed in the current method frame `cmf`. Postcondition is the uninterpreted predicate

ResultPredicate that collects the computed result. Γ and Δ are all first-order top-level formulas of the sequent of r representing the context knowledge. After applying the standard verification strategy, each open branch represents a return value valid under its path condition.

11.4.6 Current State

The values of visible variables in each symbolic execution tree node can be inspected. Visible variables are the current `this` reference, method parameters, and variables changed by assignments. This includes local variables and static fields, but highlights also two differences as compared to Java:

- KeY does not maintain local variables on the call stack. If a name is already in use it is renamed instead. As a consequence, the current state contains also local variables from all previous methods in the call stack.
- For efficiency, KeY removes variables from symbolic states as soon as they are no longer needed. This means that a previously modified local variable may get removed if it is not used in the remaining method body.

Each visible variable can have multiple values caused by, for instance, aliasing, or because nothing is known about it yet. The values for a variable `loc`, together with the conditions under which they are valid, are computed in a side proof, similar as in Section 11.4.3. The proof obligation is $\Gamma \Longrightarrow \mathcal{U}\,\mathit{ResultPredicate}(\mathtt{var}), \Delta$ using the same notation as above.

If a value is an object, then it is possible to query its fields in the same way. This brings the problem that it is possible to query fields about which no information is contained in the current sequent. Imagine, for instance, class `LinkedList` with instance variable `LinkedList next` and a sequent which says that `obj.next` is not `null`. When `obj.next` is now queried, its value will be a symbolic object. Since the value is not `null` we can query `obj.next.next`. But this time, the sequent says nothing about `obj.next.next`, consequently it could be `null` or not. In case it is not `null`, the query `obj.next.next.next` can be asked, etc. To avoid states with unbounded depth, the symbolic execution engine returns simply $<$`unknown value`$>$ in case a field is not mentioned in the queried sequent.

Defining the current state by visible variables offers a view related to the source code. Alternatively, the current state could be defined as all locations and objects contained in the update (ignoring visibility). This offers a view related to verification with JavaDL.

11.4.7 Controlled Execution

A proof strategy not only decides which rule is applied next, but also selects the branch on which the next rule is applied and it decides when to stop rule application. The strategy used for verification performs a depth first proof search. It applies rules on one branch until it is closed or no more rules are applicable. It continues then with another branch in the same way until the whole proof is closed or a preset maximal number of rule applications is reached.

This behavior is not suitable for symbolic execution because a single path may never terminate. Instead, the symbolic execution strategy applies rules on a branch as long as the next rule application would generate a new symbolic execution tree node. Before that rule is applied, the strategy continues on another branch. When the next rule on all branches would cause a new symbolic execution tree node, the cycle starts over on the first branch. This ensures that one symbolic execution step at a time is performed on all branches. A preset number m of maximally executed symbolic execution tree nodes per branch is used as a stop condition in case that a symbolic execution tree has an unbounded depth.

If m is set to one, this corresponds to a *step into* instruction in an interactive debugger. A *step over* can be realized by stopping when a node with the same or lower stack trace size than the current one is encountered. The instruction *step return* is even more strict and requires that the stack trace size is indeed lower. More advanced stop conditions are available for each supported breakpoint type (e.g., line or exceptional breakpoints).

11.4.8 Memory Layouts

Aliased references do not necessarily result in different execution paths. One single symbolic execution path can represent many concrete execution paths with differently aliased references, corresponding to different data structures in memory. The symbolic execution engine allows one to compute for each node in the symbolic execution tree all possible aliasing structures and the resulting data structures in memory. Each equivalence class of variables referring to the same object, together with the resulting memory structure, is named a *memory layout*.

Memory layouts can be computed for the current state as well as for the initial state where the current computation started. The first step in doing this is to compute all possible equivalence classes of the current state. Based on this, it is then possible to compute the specific values resulting in the memory structure.

To compute the equivalence classes, the used objects occurring in the current sequent must be known. These are all terms with a reference type, meaning that they represent an object in Java, except those objects created during symbolic execution, and the variable `exc` in the proof obligation (11.4.1). Symbolic states $\mathcal{U}$ in KeY explicitly list objects created during symbolic execution, so they can be easily filtered out. The constant `null` is also added to the used objects because we want to check whether an object can be `null`.

After the used objects are identified, a side proof checks which of them can be aliases. The initial proof obligation is simply the current context knowledge

$$\Gamma \Longrightarrow \Delta$$

where Γ and Δ are all first-order top-level formulas of the original sequent.

For each possible combination of two used objects o_1 and o_2 (ignoring symmetry), first a case distinction on $\mathcal{U}_{root}(o_1 \doteq o_2)$ is applied to all open goals of the side proof, then the automatic proof strategy is started. The updates $\mathcal{U}_{root}$ of the proof tree root is considered because it backups the initial state and thus provide additional equality constraints.

This will close all branches representing impossible equivalence classes. The branch conditions from the case distinctions on each open branch of the side proof represent the equivalence classes of a memory layout m. The symbolic values of variables $\mathtt{var}_1, \ldots, \mathtt{var}_n$ can be queried as shown in Section 11.4.6, but with the slightly modified initial sequent $\Gamma, cbc \Longrightarrow \mathcal{U}\, ResultPredicate(\mathtt{var}_1, \ldots, \mathtt{var}_n), \Delta$, where cbc is the conjunction of the branch conditions from case distinctions on the path specifying m. As the case distinctions were exhaustive on all used objects, only a single value can be computed from this query. The side proof can be based either on the current node or on the root of the proof to inspect how the memory was before symbolic execution started.

The symbolic execution API does not query field by field to compute the full data structures of the memory. Instead, all variables used in the sequent are queried at once, which is achieved by adding them as parameters $\mathtt{var}_1, \ldots, \mathtt{var}_n$ to predicate $ResultPredicate$.

11.5 Conclusion And Future Work

Recent years witnessed a renewed dynamics in research devoted to debugging. To a considerable degree this is based on breakthroughs in static analysis of software, see [Ayewah et al., 2008]. The book by Zeller [2006] presents a systematic approach to debugging and an overview of currently developed and researched debugging techniques.

The Symbolic Execution Debugger is the first debugging tool that is (a) based on symbolic execution and first-order automated deduction, (b) visualizes complex control and data structures, including reference types, (c) can render unbounded loops and method calls with the help of specifications, and (d) is seamlessly integrated into a mainstream IDE (Eclipse). Other tools have capabilities (b) and (d), but to the best of our knowledge, the SED is the first tool to realize (a) and (c). A prototype of the SED was presented by Hähnle et al. [2010], however, it lacked (c).

The SED can also be used as alternative GUI for the KeY prover. It is possible to use the SED for formal verification (see Section 11.3.6) and in addition to switch into interactive mode when KeY's proof strategy was not powerful enough to close a goal automatically. The advantages are obvious: the SED-like interface for the KeY prover inherits properties (b) and (d) from above. In addition it is not only attractive to software developers unfamiliar with formal methods, but it also constitutes a continuous transition from the world of software developers into the world of formal verification.

In future work we plan to develop the SED further into a software analysis tool that supports *code reviews*, as pioneered by Fagan [1976]. For this it is necessary to increase the coverage of Java beyond what is currently supported by the KeY verifier. The most important gaps are floating-point types and concurrent programs. Both areas constitute open research problems for formal verification, however, it is not at all unrealistic to implement support of these features in the context of debugging. The reason is that for debugging purposes often an approximation of the program semantics is already useful. For example, floating-point types might be approximated by fixed point representations or by confidence intervals, whereas symbolic execution of multithreaded Java would simply concentrate on the thread from which execution is started.