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 \cite{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 \cite{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 \cite{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.

```java
public class Number {
    private int value;

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

    // ...
}
```

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,