Bachelor/Master thesis: Algorithmic Debugging with Symbolic Execution Trees

Software Engineering Group — Martin Hentschel — hentschel@cs.tu-darmstadt.de

Context

Symbolic execution is a program analysis technique that runs a program with symbolic values in lieu of concrete values. Thus whenever the knowledge about symbolic values is not enough to made a decision symbolic execution splits to cover all possible executions. This results in a *symbolic execution tree* which captures the entire program behavior up to a certain point.

The Symbolic Execution Debugger (SED), available at www.key-project.org, extends the Eclipse debug platform by symbolic execution and by visualization capabilities. SED works for nearly full sequential Java and provides a symbolic execution engine based on the KeY verification system. The user can directly execute any method or statement(s) without setting up a fixture. Being based on symbolic execution, all feasible execution paths are discovered simultaneously. The users can focus on the paths of interest and control symbolic execution using classical navigation functionality like step into/over or breakpoints. The symbolic execution tree is built up incrementally and visualized.

Algorithmic debugging (also called declarative debugging) is a semi-automatic debugging technique that allows the developer to identify buggy methods by answering questions about the expected behavior of method calls. Method calls are recorded during a buggy program execution and represented as *execution tree*. The user is iteratively asked to classify recorded method calls as buggy or correct until a single method as source of the bug is isolated. Different strategies and transformations of the execution tree exist to reduce the number of questions that have to be answered.

Thesis

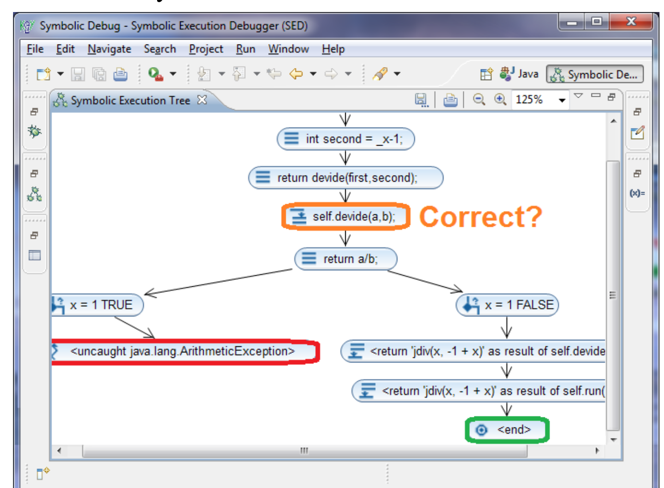
The aim of this thesis is to identify sources of bugs as precise and as automatic as possible using symbolic execution and algorithmic debugging. This requires to

a) apply and extend the idea of algorithmic debugging on symbolic execution trees.

b) implement the developed concept as part of Symbolic Execution Debugger.

c) evaluate the approach using case studies.

The following mockup gives an impression how the user interface could look like. The node with a red border represents a buggy program state, while the green outlined node represents an expected program state. The node with the orange border has been identified as a potential candidate as origin for the bug and the user is asked to confirm or deny this claim.



Contact

Martin Hentschel — Software Engineering Group — hentschel@cs.tu-darmstadt.de — S2 02 | A223