Visualization Support for JDart

Chaofeng Zhou
University of Utah
chaofeng.zhou@utah.edu

Kasper S. Luckow
Carnegie Mellon University
kasper.luckow@sv.cmu.edu

Falk Howar
TU Clausthal / IPSSE
falk.howar@tu-clausthal.de

Zvonimir Rakamarić
University of Utah
zvonimir@cs.utah.edu

ABSTRACT

JDART is a tool for performing dynamic symbolic execution of a Java program. The result is a constraints tree that describes the decisions taken during program exploration. Such a tree typically contains thousands of nodes even for medium-sized programs. It is very difficult to comprehend such large trees since, for example, identifying nodes that match particular program branches is extremely tedious. Hence, debugging and program understanding is all but impossible. To address this, we describe recent advances in the reporting facility in JDART that uses an interactive visualization of the tree, thus enabling a developer to traverse and search for specific behaviors.

Keywords

Dynamic Symbolic Execution, Visualization, Program Understanding

1. INTRODUCTION

JDART [2] is a tool for performing dynamic symbolic execution of Java programs; it is built on top of the JAVA PATHFINDER tool-set [1]. The aim of dynamic symbolic execution is to leverage automatic *Satisfiability Modulo Theories* (SMT) solvers in order to explore additional program behaviors by generating input values which result in a different path being taken through a program.

The result of dynamic symbolic execution is a constraints tree, i.e., a tree with its inner nodes reflecting the decisions (involving at least one symbolic variable) that were made during the exploration of a particular path in the program. Leaves in the tree (i.e., explored paths) are labeled as follows: *OK* upon normal exploration, *ERROR* if an exception is thrown or an assertion is violated (i.e., an error in the program was discovered), or *DONT_KNOW* if no valuation could be generated for the respective path (e.g., due to limitations of the underlying solver or constraints from undecidable theories). By default, JDART only support outputting the constraints tree in either plain text or JSON.

JPF-Workshop 2016

DOI: 10.1145/1235

Listing 1 gives a simple Java program, while Fig. 1 shows the respective constraints tree as outputted by JDART.

```
public void baz(Data d) {
  if (d.getX() < 5) {
    System.err.println("x < 5");
  }
  if (d.getY() > 40) {
    System.err.println("y > 40");
  }
  assert (d.getX() + d.getY() < 43);
}
```

Listing 1: Simple Java example.

```
-('d.x' >= 5)
  |-[+]-('d.y' <= 40)
  |      |-[+]-(('d.x' + 'd.y') < 43)
  |      |      |-[+]_/OK: [ d.x:='d.x', d.y:='d.y',  ]
  |      |      +-[-]_/ERROR: java.lang.AssertionError
  |      +-[-]-(('d.x' + 'd.y') < 43)
  |             |-[+]_/OK: [ d.x:='d.x', d.y:='d.y',  ]
  |             +-[-]_/ERROR: java.lang.AssertionError
  +-[-]-('d.y' <= 40)
         |-[+]-(('d.x' + 'd.y') < 43)
         |      |-[+]_/OK: [ d.x:='d.x', d.y:='d.y',  ]
         |      +-[-]_/ERROR: java.lang.AssertionError
         +-[-]-(('d.x' + 'd.y') < 43)
                |-[+]_/OK: [ d.x:='d.x', d.y:='d.y',  ]
                +-[-]_/ERROR: java.lang.AssertionError
```

Figure 1: Constraints tree example.

One of the main purposes of symbolic analysis is *program comprehension*, such as understanding the behaviors (i.e., decisions) leading to an *ERROR* state in the program. However, it is extremely hard and tedious to do this in the simplistic representation currently supported in JDART (see Fig. 1). To address this issue, we describe our ongoing effort supported by Google Summer of Code to add a visualization component to JDART, called JDART-VIS. This component enables the developer to interactively explore the constraints and search for specific behaviors and analysis results. We describe this new component and demonstrate its applicability in a preliminary case study.

2. VISUALIZATION SUPPORT

JDART-VIS is a web-based tool that enables interactive exploration of the resulting constraints tree. The web-based approach to visualization brings interesting new ways of how symbolic analysis is conducted and the analysis result conveyed in a team of developers. One can imagine it as part of

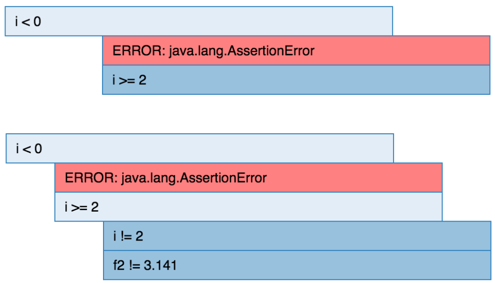

Figure 2: Expanding decision nodes.

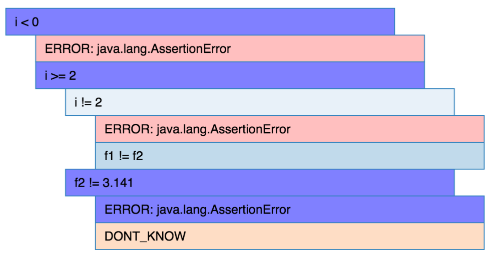

Figure 3: Highlighting *ERROR* paths.

a continuous integration environment or as a team collaboration tool, where developers can consult the visual output to get a better understanding of the behaviors in critical components or to facilitate understanding of causes of bugs.

The front-end itself is largely based on the JavaScript D3.js[1] library. D3.js is a powerful visualization tool that calculates the coordinates for geometric elements (rectangles in our case) and places them in SVG of HTML. Furthermore, it is a data-driven JavaScript library, which means the generation of the rectangles is driven by a JavaScript Object parsed from a JSON file. The JSON file in our case contains the symbolic decisions (i.e., the nodes in the constraints tree), status labels and, information about child nodes.

At a basic level, a user has an intuitive overview of the whole generated tree. Users can navigate the tree in an interactive fashion by expanding/collapsing decision nodes as shown in Figure 2.

Also, users have the ability to expand all paths filtered by the status labels. This is particularly useful in those cases where JDART computes constraint trees with thousands of deep paths and decisions, but only relatively few paths lead to *ERROR*. In this case, the user can easily isolate those paths and study in detail the program behavior leading to those outcomes. This functionality is shown in Figure 3.

Finally, hovering over a node highlights its ancestor nodes to easily see the execution path leading to the particular decision being explored.

JDART-VIS can be initiated using the command-line interface application we developed; after JDART terminates and generates the JSON-formatted constraints tree, a browser-based web-panel starts up and automatically loads the JSON file in order to render the tree. At a later stage, we will enable the user to invoke JDART directly from the web-interface. We have created an online demo site[2] to show how the visualization and the described features work.

[1] https://d3js.org/
[2] http://chaofz.me/jdart-vis

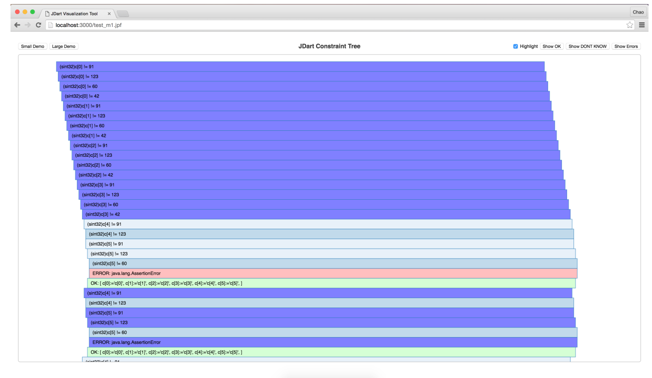

Figure 4: Finding *ERROR* paths with JDART-VIS.

2.1 Example

To demonstrate the merits of JDART-VIS, we use the method shown in Listing 2.

```
public int m1(char[] c, int n) {
  String str = new String(c);
  int state = 0;
  for(int i = 0; i < c.length; i++) {
    if(c[i] == '[') state = 1;
    else if (state == 1 & c[i] == '{') state = 2;
    else if (state == 2 & c[i] == '<') assert(false);
    else if (state == 3 & c[i] == '*') {
      state = 4;
      if(c.length == 15) {
        state = state + n;
      }
    }
  }
  return 1;
}
```

Listing 2: Simple Java example.

For this example, JDART explores 31,249 constraint nodes. Among these, there are 611 *ERROR* paths. Identifying those paths from a textual representation—as the one shown previously—is inherently difficult due to the number of paths and their depth.

However, with the help from JDART-VIS, we were able to isolate the *ERROR* paths and study the constraints (and thus the input) that expose them. As Figure 4 shows, the enormous constraint node are presented graphically and hierarchically on the panel: regular nodes have blue background, while leaf nodes with status *OK* are green; yellow denotes *DONT_KNOW* status nodes, and *ERROR* status nodes are red.

3. REFERENCES

[1] Java Pathfinder. http://jpf.byu.edu.

[2] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman. JDart: A dynamic symbolic analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 442–459, 2016.