Research Topics in Software Quality

Assignment 1:
Automated Testing using Symbolic Execution

Academic Year 2014-2015
Coen De Roover, Quentin Stevenart
Vrije Universiteit Brussel

Submission deadline:
Sunday, 18 January 2015 by 23:59 (first session)
Sunday, 30 August 2015 by 23:59 (second session)

Submit a written report about the following assignments by e-mail to cderoove@vub.ac.be. The report should be typeset in \LaTeX at a font size of 10pt on an A4 paper size. A minimum of 10 and a maximum of 30 pages is expected—not including appendices and references. Make sure each of the following points is discussed at length. Where possible, use small code snippets to illustrate your arguments. Include the complete source code for the assignment in a zip archive.

Discussion Point 1: Static versus dynamic symbolic execution The Lime Concolic Tester (LCT), used during the exercise sessions, performs dynamic symbolic execution. As seen in the lectures, there also exist tools that perform static symbolic execution.

Using an example program, illustrate and explain the advantages of dynamic symbolic execution over static symbolic execution. To this end, use both the dynamic symbolic executor LCT and the static symbolic executor KEY on your example program. We suggest using a program with a \texttt{while}-loop of which the stop condition uses a symbolic variable. Discuss the kind of user input that both tools required in order to complete their analysis of your program.

To install KEY, follow the tutorial available at http://www.key-project.org/eclipse/SED/tutorial.html. For Eclipse Luna (the latest version of Eclipse), use the following update site instead of the one given in the tutorial: http://www.key-project.org/download/releases/eclipse/luna.

Discussion Point 2: An alternative approach to symbolic object creation The topic of the second discussion point is symbolic object creation in LCT. When an instrumented program requests a new symbolic object through a call to LCT.object, an instance of this class is created. Fields within this
import fi.hut.ics.lime.tester.LCT;

public class Foo
{
    public static void main(String[] args) {
        Foo x = (Foo) LCT.getObject("Foo");
        Foo y = (Foo) LCT.getObject("Foo");
        if (x == y) {
            System.out.println("equal");
        }
        System.out.println("done");
    }
}

Figure 1: Example code with explicitly requested symbolic objects.

instance will be assigned a value, possibly another symbolic object to be created, upon their first use.

Our goal is to explore an alternative approach to this object creation. We will keep the lazy aspect of field initialization, but will change object creation such that additional paths will be explored on which the newly value is not only a fresh object, but also the null reference or a previously created object of the requested type. In the current implementation, these two additional options are only explored when this is required by the path constraints.

Consider the example code depicted in Figure 1. It creates two symbolic objects. Driven by the constraints induced by the x==y condition, LCT will explore two paths: one path with two distinct objects (i.e., on distinct addresses), and another one on which both x and y point to the same object. Figure 2 depicts the corresponding computation tree.

The computation tree to be explored by your approach is depicted in Figure 3. It contains additional paths on which x and y point either to null, two different objects or the same one.

Provide a detailed description of your design and of the important implementation details in the report. Provide two example programs that each illustrate one advantage or disadvantage of the new approach compared to the approach used by LCT. Explain why this is the case. Also include and describe some reference constraints generated by LCT for those example programs.

Information about the way LCT initializes and deals with objects can be found in LCT’s technical report1. Section 3, paragraph “Symbolic Execution with Objects” explains how objects are created and initialized. Section 4.2 “Construct-

Figure 2: Symbolic object creation as implemented by LCT.

*Fig. 2.* Symbolic object creation as implemented by LCT.

*Fig. 2.* Symbolic object creation as implemented by LCT.

In Figure 2, a symbolic object creation is shown. The tree structure indicates how symbolic objects are created and used in the execution tree. The tree branches illustrate the symbolic variables and their relationships.

The following summarizes the most relevant information:

- **Relevant background on the architecture**
  
  We explored the architecture of LCT in the second exercise session. For this assignment, you will need to modify the test selector (i.e. the server). When running LCT, the test executor (i.e. the instrumented program) is in contact with the test selector to know which input to use and to inform the selector of the branches taken.

  During the execution of the program, the selector will receive messages that enable it to build part of the execution tree (as in Figure 1). At each received message, the selector will add nodes to the tree. For example, if the program reaches a condition containing a symbolic variable, such as if (x > 5) with 10 as concrete value of x, it will send a message such as:

  \[
  \text{New arithmetic condition: } \leq \text{i32_0} 5 \text{ true t}
  \]

  This message contains the path constraint (\(\leq \text{i32_0} 5\) where \text{i32_0} is the symbolic variable corresponding to x) as well as the branch and path taken during the concrete execution (the branch taken is the true branch, and the full path is represented by t). The message is handled by the server’s worker (server.Worker) class. The worker will dispatch the work to the search strategy class (strategies.scheduling.BasicScheduledStrategy is used by default, and you are only asked to modify this strategy). The strategy will add corresponding nodes to the tree (look at its addBranch method).
Once the instrumented program completes its execution, the test selector will generate input to explore a different path, by solving the constraints to reach an unexplored node in the execution tree. Once those inputs are found, they are sent to the new test executor and the analyzed program can run again.

- **Relevant background on the constraints**
  Each node in the execution tree can be associated with two types of constraints.

  1. Arithmetic constraints that act on arithmetic values, such as $\leq 132.0 5$.
  2. Reference constraints that act on objects. There are only two types of reference constraints:
     (a) $== 01 02$ indicates that the two logical objects 01 and 02 should be the same object,
     (b) $!= 01 02$ indicates that they should be different.

     Also, the logical object 00 is the value null.