Assignment 03: Symbolic Execution

Angelo Porcella, Emery Call, Jasmine Vang, Prakriti Baral

November 3rd, 2024

1 Introduction

Our symbolic execution engine meets the requirements of the assignment, both supporting a small portion of the C language and symbolically executing specified expressions. The engine has the indicated output and additionally prints a graph representation of the symbolic states. The engine uses tree-sitter and Z3.

2 Architecture

The symbolic engine starts by parsing the source code to find the target function and abstract syntax tree (AST) node. Then the interpreter traverses the AST. The interpreter handles scope for variables in nested statements and interprets statement nodes such as 'if' and 'while' with symbolic constraints. Using Z3 the constraints are checked for feasibility. The results are output to the terminal and shown in a graph.

2.1 Parse Source Code (runner.py)

The parsing is completed with the assistance of tree-sitter. The parser is instantiated with the C language and generates the AST. Within the AST all functions are found, which are used to find the target function node. The target function is then used in the interpreter.

2.2 Interpreter (interpreter.py)

The interpreter uses a loop to symbolically evaluate each AST node. The nodes include: compound statements, declarations and assignments, if statements, while statements, and return statements. The interpreter accumulates non-forking nodes like variable declarations and assignments into its list of constraints. Variable assignments and declarations will also cause updates to the mappings of each variable name within the C program to a new corresponding Z3 variable name (Ex: $x \rightarrow x_0$, $x \rightarrow x_1$, ... $x \rightarrow x_i$) which captures the order of assignments within the constraints. When it encounters execution forking nodes like if statements and while statements it creates two copies of itself to recursively explore the true or false branch of the condition. It only continues down a branch if, when adding the condition constraint to the constraint list of either child interpreter, it doesn't make that branch infeasible.

3 Implementation

Provided below are some implementation details of how we get some specific things to work.

3.1 Pre/Post Increment/Decrement

To support post increment $x++$ and pre-increment $++x$ operators within an expression we have to assign variables at different times. For the pre-operator ($++x$ and $--x$), when we encounter it within an expression we immediately assign the variable and then return the updated Z3 variable for the expression to evaluate. For the post operator ($x++$ and $x--$) we maintain a list called post_assignments where we accumulate all encountered post operators within the expression and then apply all of them once the expression has finished evaluating.

3.2 Truthiness

To implement C style truthiness where the int value 0 is false and all others are true we added some helper methods forceInt(exp) and forceBool(exp). These functions coerce the passed Z3 expression to an int or bool expression following C's rules. We then use these helper methods on the left and right

side of bool operators like || to coerce either side to be bools or on the left and right side of arithmetic operators like * to coerce either side to be integers.

4 Instructions: Set Up and Running Symbolic Execution Engine

4.1 Set Up

There are several libraries necessary for running the SEE. To get the engine running as simply as possible, Tree-sitter and z3-solver are the only two main packages needed. These can be installed using the following commands:

1. pip install tree-sitter tree-sitter-c
2. pip install z3-solver

The additional outputted state tree can be used to analyze the paths through the program and provide a visual that can make it easy to debug any issues. In order to display this, several more packages are required. Graphviz, tkinter, networkx and matplotlib are all utilized to display the state tree. To install these, first run these three commands:

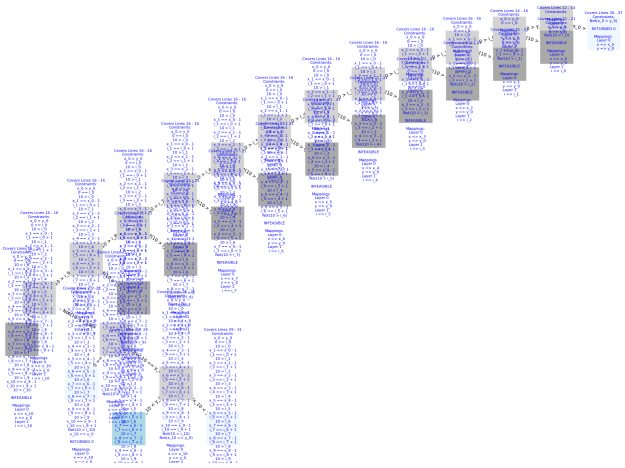
1. sudo apt install graphviz
2. sudo apt install python3-tk
3. sudo apt install libgraphviz-dev

After installing these libraries this way, it is then possible to run a pip command to finish the installation:

1. pip install pygraphviz networkx matplotlib

4.1.1 Reading the State Tree

In the plots produced, dark grey squares are infeasible states, light grey squares are a part of feasible paths, light blue squares are states that return 0, and the dark blue square is the target state that returns 1.



4.2 Run the Engine

In order to run the engine on a function, command line arguments are used. The file runner.py is the driver for the engine. The command takes in the filepath to the C code, as well as the name of the function to execute. In the following example, tests/real_test.c is our filepath to our desired code, and "test" is the name of the function in the file we are analyzing:

1. python3 runner.py tests/real_test.c test

The engine should then display several metrics regarding the analyzed function. These include the number of infeasible states found, the number of feasible paths through the function to a return statement, and the number of feasible paths to reach the target statement (return 1). In addition, it will print the constraints required to reach the target statement and give concrete values needed to reach the target statement.

```
###### Stats on symbolic state tree:
Infeasible states: 11

Feasible paths: 4

Feasible paths to target (return 1;): 1

#### Feasible path to target 1/1:
Constraints on function parameters to reach target statement:
Not(x_0 <= y_0)
Not(x_0 == 10 + y_0)
Not(y_0 <= -10 + x_0)

Concrete values to reach target statement:
y_0 = -1
x_0 = 0
```