Symbolic Execution Engine

Mudathir Mahgoub

May 11, 2018

1 Project Problem

Symbolic execution engines are commonly used in formal verification and software testing. Traditional testing methods use different input values to check specified properties about some code. Normally it is difficult to handle all possible input values especially when the input space is huge or infinite. Symbolic execution on the other hand relies on unknown symbolic values for the input and explores all possible execution paths while maintaining a symbolic state for each path. With the help of constraint solvers like SAT solvers or SMT solvers, we can reason about the specified properties and formally verify whether a property holds in all execution paths, or give a counter example if it doesn't.

In this project the target code is a simple portion of the C programming language, and the properties are specified using assertions.

2 Motivation or Importance of the problem

Symbolic execution is mostly used in software verification for critical systems. In such systems, strong security and safety properties need to be guaranteed. Examples of security properties include an array access within its range, and the absence of division by zero. An example of a safety property for software embedded in a microwave is guaranteeing that microwave is never running while its door is open.

2.1 Personal motivation

3 Challenges

The paper [1] describes many challenges in symbolic execution that include:

- Scalability: exploring all execution paths doesn't scale well for large programs where number of paths increases exponentially. Therefore in large programs, mixing concrete execution with symbolic execution (concolic execution), and other heuristic approaches are usually used.
- Memory: how to symbolically execute operations on pointers, arrays, objects
- Environment: how to symbolically interact with time, operating system, network
- Loops: the number of paths depends on the number of iterations in the loop. In general, determining whether a loop terminates is not decidable. So symbolic execution can only work with restricted loops or use some heuristics.

- Constraint solving: symbolic execution is limited by the power of the underlying solvers. For example non-linear arithmetic constraints, which are common, are not usually efficient in SMT solvers.

3.1 Personal challenges in this project

Some issues arose during the implementation of the project. Here are some of them:

- A grammar for C: the official grammar is too much for this project, and many simple grammars in the internet were not enough. So I prepared a specific one with the help of 3 grammars cymbol[1], tinyC[2] and antlr C[3] .
- A grammar for SMT-LIB: the official grammar for SMT-LIB[4] is also a lot for this project which only uses arithmetic theory.

4 Existing Approaches

Paper [1] describes several approaches to symbolic execution including fully symbolic execution, concolic execution, dynamic symbolic execution, selective symbolic execution, path selection, symbolic backward execution. It also describes many ways to handle memory including fully symbolic memory, address concretization, partial memory modeling, lazy initialization.

In this project I used fully symbolic execution with fully symbolic memory.

5 My approach

A general overview of the project is given in section 5.1. The symbolic execution engine is described in section 5.2.

5.1 Software description

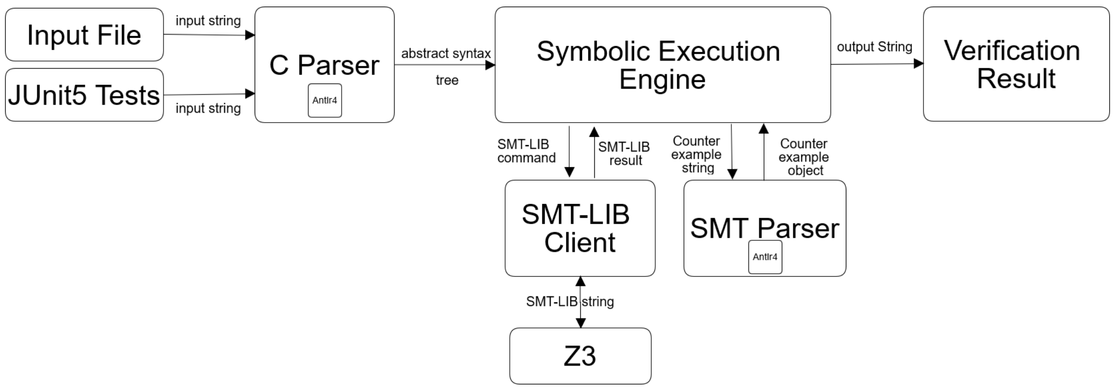

Figure 1: Project architecture.

The project is implemented using Java and the executable is a jar file (target/SymbolicEngine.jar) which is generated using the command:

[1]https://github.com/remenska/Grammars/blob/master/book-examples/examples/Cymbol.g4

[2]https://github.com/antlr/grammars-v4/blob/master/tinyc/tinyc.g4

[3]https://github.com/antlr/grammars-v4/blob/master/c/C.g4

[4]https://github.com/julianthome/smtlibv2-grammar/blob/master/src/main/resources/SMTLIBv2.g4

```
mvn install
```

The program receives as an input a source file containing a simple C function. For testing, **JUnit5** was used to test the program directly using 45 unit tests. The input (from the input file or the unit test) is passed to the **C Parser** which uses the ANTLR library to parse the input into an abstract syntax tree. This abstract syntax tree is consumed by the **Symbolic execution engine** which executes statements and evaluates expressions symbolically to build assertion formulas which are encoded in SMT-LIB commands. These commands are passed to the **SMT-LIB client** which interacts with the SMT solver Z3. Z3 results are passed to **Symbolic execution engine** through **SMT-LIB Client**. Z3 counter examples are handled by the **SMT Parser**. Finally the **Symbolic execution engine** returns the verification results which include a validity answer for each formula: **Yes**, **No** with a counter example, or **Unknown**.

Here is an example of an input:

Listing 1: test.txt

```
void f (int x, int y)
{
    if(x > 0)
    {
        y = x;
    }
    assert (y == x);
}
```

Here is the output:

Listing 2: java -jar TypeChecker.jar -i test.txt

```
-----------------
Overall Answer: No
-----------------
Assertion: (BinaryExpression(Variable(y) == Variable(x)))
AssertionFormula:
(assert (> _x1 0))
(assert (not (= _x1 _x1)))
Answer: Yes
---------------------------
AssertionFormula:
(assert (not (> _x1 0)))
(assert (not (= _x2 _x1)))
Answer: No
Counter example: {x=0, y=1}
---------------------------
```

The following section focuses on the symbolic execution engine.

5.2 Symbolic execution engine

6 Work Division

Since there is only one person in this project, all the work is done by him.

References

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *CoRR*, abs/1610.00502, 2016.