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

Initially my idea was to do another project. When I was answering the last question in the take home exam it was about symbolic execution and the answer needed some constraints solving using high school math. After solving only 2 formulas, it became tedious and I tried to use z3 online to solve them. That also became more tedious to write SMT-LIB formulas, so finally I decided to do this project when the first project didn't work out. As a bonus I verified my manual answer to that question using this project. So for that I am satisfied.

Another motivation is that symbolic execution is related to my research. So obviously this is supposed to be my first choice, but the first one involved reverse engineering which is really cool. Also I decided to use Z3 something different than CVC4 which I would be working on in the future.

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. In this project only variables are supported.
- Environment: how to symbolically interact with time, operating system, network. Environment is not supported in this project.
- 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. Loops are not supported in this project.
- 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 Project challenges

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 integer arithmetic. So I wrote one just for parsing the counter examples returned by the SMT solver.
- Deadlock with Z3 process: one way to interact with Z3 is to call it as a separate process, and send SMT-LIB commands and receive the output. However Z3 process blocks waiting for an input when it executes the previous command. Initially the symbolic execution engine was implemented to send multiple commands and then receive the output of the satisfiablility checking or the model retrieval commands. This resulted in a deadlock where Z3 process is waiting for an input from the symbolic execution engine, and the engine is waiting for the output of Z3. Fortunately z3 supports the "echo" command which I used to echo white space before reading the output of Z3.

  Another way to interact with Z3 is to use Z3 binding (e.g. com.microsoft.z3.jar for Java), however this approach makes it harder to use other SMT solvers like CVC4 with has its own Java binding (CVC4.jar).

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,

[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`

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

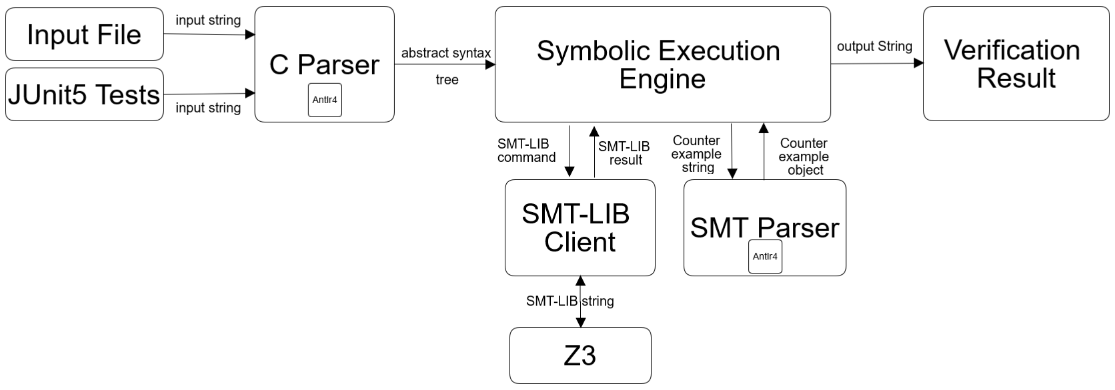

Figure 1: Project architecture.

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

```
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.c

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

Here is the output:

Listing 2: java -jar SymbolicEngine.jar -i test.c

```
------------------------------
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

The symbolic execution engine starts executing a function by symbolically assigning a new symbolic value to each argument variable. This would be the initial state of function and its block. The initial path constraint for the function would be the empty constraint (equivalent to *true*). For simplicity, I will just call the current state to mean both the current state and the current path constraint.

For convenience each statement keeps track of **start states** where it can start from, and **end states** where it can end with. The **start states** of the current statement is exactly the **end states** of the previous statement. The block of a function has only one **start state** which is the one defined for the function. The first statement of each block inherits the **start states** of its block. Each statement can not modify its **start states**, but it can modify its **end states**. An assignment statement modifies each **start state** by updating the value of the variable changed in the assignment. A variable definition statement without an assignment would add that variable to the state with a new symbolic value. This is useful to reason about the following function:

```
void f()
{
        int z;
        assert (z == z);
}
```

If the variable is defined with an assignment, the variable will be added to the state with value of the assigned expression. Only statements of assignments and variable definitions with assignment can modify a variable value in a state. Statements like (++i) or (i++) are rewritten as assignment statements (e.g. i = i + 1;)

Since loops are not supported, only **If statements** can modify path constraints in a state. The **If statement** will add its **condition** to the path constraint of each start state of its **true statement**. Likewise, the **If statement** will add the **negation of its condition** to the path constraint of each start state of its **false statement**. If the (**If statement**) has no **false**

statement, it will be rewritten such that **NoOperation** statement is its **false statement** (similar to **nop** in assembly language). This is explained with the following example:

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

The above will be rewritten to be equivalent to

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

Whenever there is a division in an expression, the symbolic execution engine would automatically add an assertion statement such that the numerator of the division is not equal to zero. The **start states** of this assertion statement would be the **start states** of the statement where the division operation resides. This explained in the following example:

```
void f (int x, int y)
{
        x = x / y;
}
```

The above would be rewritten to be equivalent to :

```
void f (int x, int y)
{
        assert(y != 0);
        x = x / y;
}
```

Finally since C allows integers to be used as booleans in assertions, this would be problematical in Z3 which doesn't support that. Therefore assertions with integer expressions are written to be logical with comparison to zero as follows:

```
void f (int x, int y)
{
        assert (x+y);
        assert (0);
}
```

The above will be rewritten to be equivalent to

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

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.