Symbolic Execution Engine

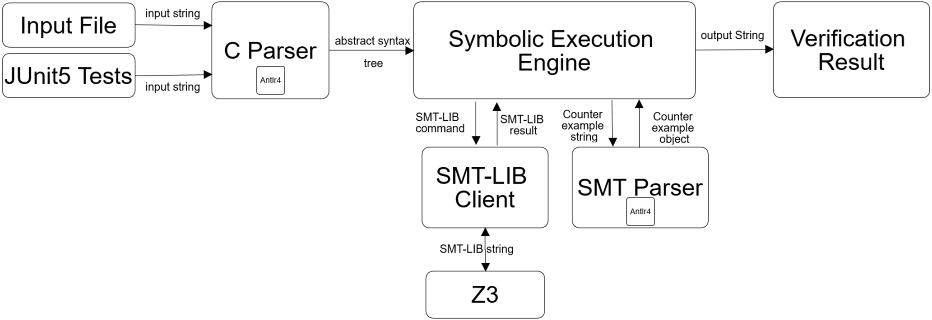
Mudathir Mahgoub

May 12, 2018

Project problem

Given a simple function (without pointers, loops, arrays, structs, etc) written in C which contains many assertions, are the boolean expressions in these assertions valid?

Software components



Execution

Input: test.c

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

Execution

Output: 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}
____________________________________
```

Demo

Motivation or Importance of the problem

Importance of symbolic execution

- Software verification for critical systems
- Security properties: e.g. access array within boundaries
- Safety properties: microwave should not run while its door is open

Personal motivation

- Related to my research
- A tedious question in the second midterm

Challenges

General challenges

- Scalability
- Memory
- Environment
- Loops
- Solver limitations

Challenges

Project challenges

- A grammar for C
- A grammar for SMT-LIB
- Deadlock with Z3 process

Existing approaches

- Concolic execution
- Dynamic symbolic execution
- Selective symbolic execution
- Path selection
- Symbolic backward execution
- Fully symbolic memory
- Address concretization
- Partial memory modeling
- Lazy initialization

My approach

- Only handles functions
- Simple integer variables
- No pointers, arrays, structs
- No bitwise operations
- No loops
- Fully symbolic execution

My approach

- Start with function definition
- Assign a new symbolic value to each argument variable
- Set the initial path constraint to empty (i.e. *true*)
- **Initial start state** = argument variables(with symbolic values) + empty constraint
- Each statement has **start states** and **end states**. A statement can't modify its **start states**
- The block of the function has one **start state** = **Initial start state**
- The first statement in each block has the same **start states** of its block
- Each subsequent statement has its **start state** = **end states** of the previous statement

My approach

- An assignment statement (*variable* = *expression*) has **end states** exactly as **start states** except with the updated *variable* which gets the symbolic value of the *expression*
- A variable definition statement without assignment has **end states** exactly as **start states** except with the addition of the new variable which gets a new symbolic value

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

- A variable definition statement with assignment has **end states** exactly as **start states** except with the addition of the new variable which gets the value of the *expression*

My approach

- Statements like (++i) or (i++) are rewritten as assignment statements (e.g. i = i + 1;)
- **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**

My approach

- If the **If statement** has no **false statement**, NoOperation statement will be used instead

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

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

My approach

- For each division operation, a new assertion statement would be added. The **start states** of this assertion would be the **start states** of the statement where the division resides

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

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

My approach

- Integers as booleans

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

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

My approach

- Unknown

```
void f (int x, int y, int z) +
{
            assert(!((x*x*x + y*y*y == z*z*z) &&
            (x>10) && (y>10) && (z>10)));
};
```

Work Division

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

References

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