ECE653 - Testing, Quality Assurance, and Maintenance

University of Waterloo

Department of electrical and computer Engineering

Final Project Report

Authors:
Zhengkang Chen (ID: 20789905)
Wenting Ju (ID: 20777538)

Date: December 20, 2023

Contents

1 Introduction

Symbolic execution is a way of executing a program abstractly so that one abstract execution covers multiple possible inputs of the program that share a particular execution path through the code. The execution treats these inputs symbolically, "returning" a result that is expressed in terms of symbolic constants that represent those input values.[3] Symbolic execution was originally proposed in the 1970s, but it relied on automated theorem proving, and the algorithms and hardware of that period weren't ready for widespread use. [3] With recent advances in SAT/SMT solving and Moore's Law applied to hardware, symbolic execution techniques have evolved significantly in the decades, with notable applications to compelling problems from several domains like software testing, security, and code analysis.[5]

Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas for combinations of background theories such as arithmetic, bit-vectors, arrays, and uninterpreted functions. Z3 is an efficient SMT solver with specialized algorithms for solving background theories. SMT solving enjoys a synergistic relationship with software analysis, verification and symbolic execution tools. [6]

A main limitation of classical symbolic execution is that it cannot explore feasible executions that would result in path constraints that cannot be dealt with.[4] Loss of soundness originates from external code not traceable by the executor, as well as from complex constraints.[5] As the constraint gets complicated, it will affect the performance of the engine. Therefore, here comes the idea of concolic execution.

A fundamental idea to cope with these issues and to make symbolic execution feasible in practice is to mix concrete and symbolic execution.[5] One popular concolic execution approach, known as dynamic symbolic execution (DSE) or dynamic test generation, is to have concrete execution and drive symbolic execution.[2] That is dynamic symbolic execution interprets the program with the concrete state while the symbolic state is computed in parallel with it. EXE style is one of the two main flavours of dynamic symbolic execution.

During symbolic execution many similar queries are solved over and over again. Here comes the idea of incremental solving of Z3. Performance of symbolic execution can be improved by using incremental solving abilities of SMT solver that allow adding and withdrawing constraints between multiple calls.

2 Objectives and Descriptions

One of the objectives of this project is to implement the Concolic Execution in an EXE style based on the existing symbolic execution engine we had from the assignments to improve its efficiency. The other objective of this project is to implement using incremental solving mode of Z3 to improve performance. We used Scopes incremental solving interfaces to implement this.

2.1 Symbolic Execution

(Pure) symbolic execution automatically explores program paths by executing a program on symbolic input values, forks execution at each branch and records path conditions. It uses constraint solvers to decide path feasibility. Consider the following program.

```
1  int proc(int x){
2      int r = 0;
3      if (x > 8){
4          r = x - 7;
5      }
6      if (x < 5){
7          r = x - 2;
8      }
9      return r
10 }
```

Listing 1: Program Example

Figure 1 shows how symbolic execution works. After getting the result from the symbolic execution engine, we can find out that satisfying assignments are all feasible paths. In this case, the satisfying assignments are X = 9, X = 4 and X = 7. And the test cases will be proc(9), proc(4) and proc(7) correspondingly.

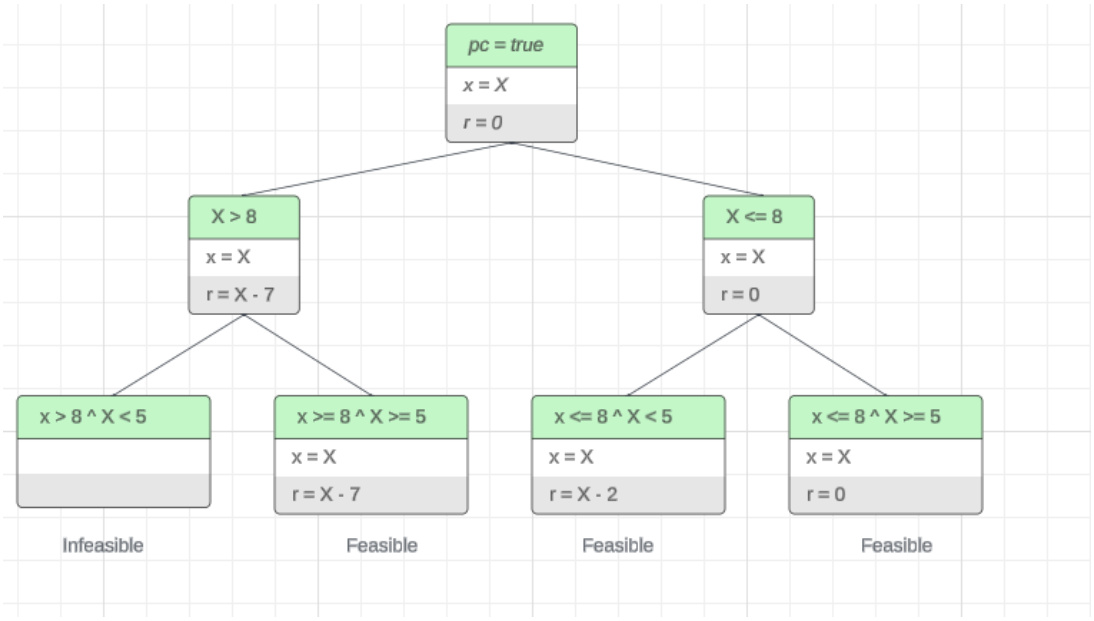

Figure 1: Example - Symbolic Execution

2.2 Concolic Execution in EXE-style

Concolic execution, a synergistic combination of concrete and symbolic execution, presents a powerful approach to program analysis. Our implementation of Concolic execution in EXE-style involved executing the program with both symbolic and concrete values simultaneously.

In EXE style, it runs the program with concrete inputs. Symbolic execution follows concrete execution in parallel to concrete execution. It keeps track of what values are inputs (symbolic) or not inputs (concrete). It computes path conditions in terms of inputs. At every branch point, concrete execution takes one branch and symbolic execution updates the current input to force execution into a different branch. This is done by updating the current path condition to go to another branch. It creates a new concrete state that follows into this branch. Thus, both states for two branches are explored.[1]

Figure 2 shows how EXE-style concolic execution works for the program example provided in listing 1. S() represents the symbolic state and C() represents the concrete state. Unlike the symbolic state, the concrete state will update its values to satisfy the path condition along the traversal of all the paths. After the concolic execution, the concrete state values will be our satisfying assignments. In this case, the satisfying assignments are X = 9, X = 4 and X = 7 and test cases proc(9), proc(4) and proc(7) correspondingly.

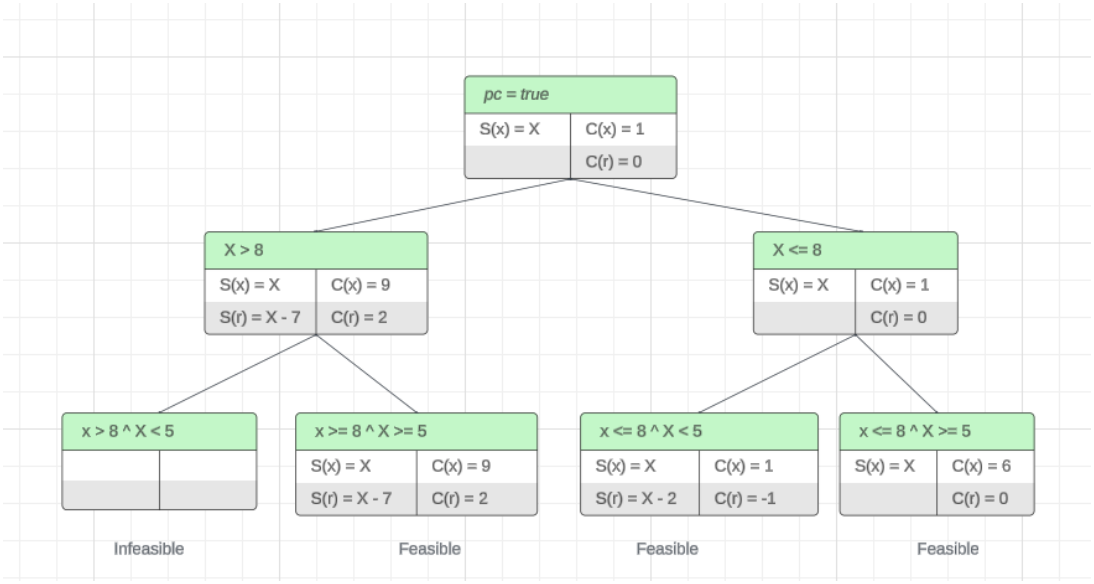

Figure 2: Example - EXE style concolic execution

2.3 Incremental Solving Mode of Z3

Solvers can be used to check the satisfiability of assertions incrementally. An initial set of assertions can be checked for satisfiability followed by additional assertions and checks. Assertions can be retracted using scopes that are pushed and popped. Under the hood, Z3

uses a one-shot solver during the first check. If further calls are made to the solver, the default behaviour is to switch to an incremental solver. [6]

The operations push and pop create, respectively revert, local scopes. Assertions that are added within a push are retracted on a matching pop. Thus, the following session results in the verdicts sat, unsat, and sat.[6]

```
p, q, r = Bools('p q r')
s = Solver()
s.add(Implies(p, q))
s.add(Not(q))
print(s.check())
s.push()
s.add(p)
print(s.check())
s.pop()
print(s.check())
```

Listing 2: Example of Scopes

3 Implementation

3.1 Concolic Execution

3.1.1 class Concolic_State

The key point of implementing Concolic execution in EXE-style is to handle symbolic and concrete states separately. Therefore, we modify the class SymState into `Concolic_State` where it has two different dictionaries called `sym_env` and `con_env` to store symbolic state and concrete state. Similarly, we added the printing statement to output the concrete state as well.

```
class Concolic_State(object):
    def __init__(self, solver=None):
        self.sym_env = dict()
        self.con_env = dict()
    # existing code...
    def fork(self):
        child.sym_env = dict(self.sym_env)
        child.con_env = dict(self.con_env)
    # existing code...
    def __str__(self):
        buf.write('con: ')
        for k, v in self.con_env.items():
            buf.write(str(k))
            buf.write(': ')
            buf.write(str(v))
```

```
16              buf.write('\n')
17      # existing code...
```

Listing 3: class Concolic_State

3.1.2 class Concolic_Exec

Similar to `Concolic_State`, in the `Concolic_Exec` we separated each function into two situations, symbolic and concrete states and implemented them separately if needed. Therefore, in the run function, we set the kwargs as a dictionary with the word 'state' to represent the state values and the word 'mode' to represent what type of execution it is currently running. 'con' represents a concrete and 'sym' represents symbolically. Each execution has its environment variables and needs to be assigned separately. Since concrete execution only has concrete values, it cannot use the Z3 SAT solver. Thus, only the execution in symbolic should use Z3 solver to evaluate the satisfiability. Listing 3 shows an example in function: `visit_AsgnStmt` and `visit_IntConst`. Similar implementations were done in functions: `visit_BoolConst`, `visit_BExp`, `visit_AExp`, and `visit_IntVar`. As for HavocStmt, it randomly assigns concrete value to the concrete state and symbolic value to the symbolic state.

```
1  class Concolic_Exec(ast.AstVisitor):
2      def run(self, ast, state):
3          kwargs = dict()
4          kwargs["state"] = state
5          kwargs["mode"] = "con"
6          args = []
7          res = self.visit(ast, *args, **kwargs)
8          return res
9      # existing code ...
10     def visit_IntConst(self, node, *args, **kwargs):
11         if (kwargs["mode"] == "sym"):
12             return z3.IntVal(node.val)
13         elif (kwargs["mode"] == "con"):
14             return node.val
15     # existing code ...
16     def visit_AsgnStmt(self, node, *args, **kwargs):
17         state = kwargs["state"]
18         kwargs["mode"] = "sym" # change mode
19         state.sym_env[node.lhs.name] = \
20         self.visit(node.rhs, *args, **kwargs)
21         kwargs["mode"] = "con"
22         state.con_env[node.lhs.name] = \
23         self.visit(node.rhs, *args, **kwargs)
24         return [state]
25     # existing code ...
```

Listing 4: class Concolic_Exec

The following Listing 4 shows the implementation on function `visit_IfStmt`. The key idea is at every branch point, concrete execution takes one branch and symbolic execution updates the current input to force execution into a different branch. The symbolic execution will evaluate the feasibility of the other branch using the Z3 SAT solver. If the other branch is satisfiable then the concrete state will update its values to take that branch. If the branch is unsatisfiable, then the concrete state won't take that branch. The implementation of `visit_whileStmt`, `visit_AssumeStmt` and `visit_AssertStmt` is using the similar idea.

```
class Concolic_Exec(ast.AstVisitor):
    # existing code ...
     def visit_IfStmt(self, node, *args, **kwargs):
        kwargs["mode"] = "sym"
        cond_sym = self.visit(node.cond, *args, **kwargs)
        kwargs["mode"] = "con"
        ...
        cond_con = self.visit(node.cond, *args, **kwargs)
        ...
        true_branch = True
        #-----------------------------------------------
        if cond_con: # concrete evaluate if stmt first
            true_branch = True
            ...
        else: # conrete takes false
            true_branch = False
            ...
        #-----------------------------------------------
        if true_branch:  # Symbolic evaluate the branch not taken
            if not new_state.is_empty():# if SAT
                new_state.con_env = new_state.pick_concrete()
                ...
            else:
                new_state.mk_error()
                ...
        else: # takes false branch
            if not curr_state.is_empty(): # if SAT
                state.con_env = state.pick_concrete()
                ...
            else: # if UNSAT
                curr_state.mk_error()
                ...
    # existing code ...
```

Listing 5: visit_IfStmt in class Concolic_Exec

3.2 Incremental Solving Mode of Z3

In this part, we used a class called `SymState_incremental` which is the same as the pure symbolic execution engine we had from assignment 3. As for the class `SymExec_incremental`, it is extended from the symbolic execution engine. We have made modifications to a few functions to achieve the incremental solving mode of Z3. The key idea is to use Solvers to check the satisfiability of assertions in an incremental way. Assertions can be retracted using scopes that are pushed and popped. The operations push and pop create and respectively revert local scopes. Assertions that are added within a push are retracted on a matching pop.[6] Here is the modification we made for the `visit_IfStmt` function.

```
class SymExec_incremental(ast.AstVisitor):
    # existing code ...
     def visit_IfStmt(self, node, *args, **kwargs):
        st = kwargs["state"]
        cond = self.visit(node.cond, *args, **kwargs)
        curr_state, new_state = st.fork()
        out = []
        curr_state.add_pc(cond)
        new_state.add_pc(z3.Not(cond))
        solver = st._solver
        solver.push()
        solver.add(cond)
        if solver.check() == z3.sat:
            ...
        solver.pop()
        solver.push()
        solver.add(z3.Not(cond))
        if solver.check() == z3.unsat:
            ...
        solver.pop()
        return out
    # existing code ...
```

Listing 6: visit_IfStmt in class SymExec_incremental

4 Results and Evaluation

4.1 Concolic Execution

Back to the program example shown in Listing 1, Listing 7 shows the output of our concolic execution engine. It only shows the results for the feasible paths. In this case, there are three feasible paths. The first feasible path condition is [8 < X, Not(5 > X)] with symbolic state: x = X and r = X - 7, and concrete state: x = 56 and r = 49. The second feasible path condition is [Not(8 < X), Not(5 > X)] with the symbolic state: x =X and r = 0, and concrete state: x = 8 and r = 0. The third feasible path condition is [Not(8 < X), 5 > X] with symbolic state: x = X

and r = X - 2, and concrete state: x = 0 and r = - 2. This is the result that we are expecting according to Figure 2. In addition to this, we tried to extend the test cases from the symbolic engine from assignment 3 to achieve complete statement coverage and branch coverage for the concolic execution engine and all the output of test cases met the expectation.

```
sym: x: X!0
r: -7 + X!0
con: x: 43
r: 36
pc: [8 < X!0, Not(5 > X!0)]

sym: x: X!0
r: 0
con: x: 8
r: 0
pc: [Not(8 < X!0), Not(5 > X!0)]

sym: x: X!0
r: -2 + X!0
con: x: 0
r: -2
pc: [Not(8 < X!0), 5 > X!0]
```

Listing 7: Output of Concolic Execution

4.2 Incremental Solving Mode of Z3

The main purpose of implementing the incremental solving mode of Z3 is to improve the engine performance. The best way to compare the difference between models is to compare the execution time of each engine. In this part, we randomly pick several test cases with complex path conditions and using Python built-in methods to keep track of the execution time. Then we calculated the average of the execution time of those test cases for each model. To reduce bias, we ran two different tests. Table 1 below shows the results of these two tests.

Type of Execution	Execution Time of Test 1	Execution Time of Test 2
Pure Symbolic Execution	0.1298 seconds	0.6617 seconds
Concolic Execution	0.1178 seconds	0.2975 seconds
Z3 Incremental	0.0854 seconds	0.1990 seconds

Table 1: Execution Time Among Three Engines

According to the results, it is clear to see that pure symbolic execution takes significantly longer time than concolic execution. In addition, Z3 incremental solving mode has also improved a good amount of execution time.

5 Conclusion

In this project, we have successfully implemented Concolic Execution in EXE-style from the existing symbolic execution engine. The Concolic Execution has integrated concrete and symbolic execution to improve the efficiency of the engine with complex path conditions. We also successfully implemented the Incremental Solving Mode of Z3 with scope interfaces from the existing symbolic execution engine. It made a significant improvement in the engine's performance compared to the original symbolic execution engine by reducing similar queries that are solved over and over again. It has a even better performance than the concolic execution with complex path conditions. These improvements make the symbolic execution be a more powerful tool of program verification and testing.

References

[1] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler. *EXE: Automatically Generating Inputs of Death*. Computer Systems Laboratory, Stanford University, Stanford, CA 94305, U.S.A. `https://web.stanford.edu/~engler/exe-ccs-06.pdf` pages 5

[2] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. *DART: Directed Automated Random Testing*. In Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI'05), 213–223. `https://doi.org/10.1145/1065010.1065036` pages 3

[3] Jonathan Aldrich and Claire Le Goues. 2018. *Symbolic Execution*. Lecture Notes for 17-355/17-665/17-819O: Program Analysis (Spring 2018), Carnegie Mellon University. `https://www.cs.cmu.edu/~aldrich/courses/17-355-18sp/notes/notes14-symbolic-execution.pdf` pages 3

[4] Cristian Cadar and Koushik Sen. 2013. *Symbolic Execution for Software Testing: Three Decades Later*. Commun. ACM 56, 2 (2013), 82–90. `https://doi.org/10.1145/2408776.2408795` pages 3

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. *A Survey of Symbolic Execution Techniques*. ACM Computing Surveys, Vol. 51, No. 3, Article 0, Publication date: 2018. `https://arxiv.org/pdf/1610.00502.pdf` pages 3

[6] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. *Programming Z3*. Microsoft Research. `https://theory.stanford.edu/~nikolaj/programmingz3.html#sec-incrementality` pages 3, 6, 9