A Dynamic Symbolic Execution Engine in EXE style

Yinyouyang Bu[1] and Fan Zhang[1]

University of Waterloo, Waterloo Ontario, Canada
{m42zhang, ybu}@uwaterloo.ca

Abstract. **Keywords:** Dynamic Symbolic Execution · Incremental Solving · Program Analysis

1 Introduction

This project aims for two features:

- to implement a Dynamic Symbolic Engine (DSE) in EXE style for the vanilla programming language WHILE;
- to exploit the incremental solving feature of z3.

There are situations where relying on symbolic execution may be impractical or desired. Dynamic symbolic execution involves executing a program with both symbolic and concrete inputs, allowing for path exploration and reasoning about possible states.

When symbolic execution encounters computationally expensive or not desired paths, the engine may choose to switch to concrete execution for efficiency. Concrete execution provides actual values for inputs, facilitating the exploration of intricate program paths.

Moreover, if the program interacts with external libraries or input sources that are difficult to symbolize efficiently. Concrete execution is used to handle interactions with these external components.

EXE employs a dual execution approach, mixing both concrete and symbolic execution of a program[3]. The system tracks the value as either symbolic inputs or concrete inputs and calculates the path condition in terms of these inputs.

At each branching point in the program, the concrete execution follows one branch (e.g., Branch 1), while symbolic execution dynamically shifts to force exploration of an alternative branch (e.g., Branch 2). This adaptation is facilitated by updating the current path condition to correspond to Branch 2 and generating a new concrete state aligned with this updated condition. Consequently, both states are explored[2].

Throughout each execution step, the concrete state undergoes updates by concretely executed program instructions. Simultaneously, the symbolic state evolves through symbolic execution. If the previous instruction involved a decision point, and the condition for taking a different path is satisfied, the system

splits into two directions. This means it explores both possibilities. To do this, a new concrete state is calculated to match the adjusted conditions for the chosen path. This way, the system comprehensively navigates through different scenarios, updating both concrete and symbolic states[2].

Our symbolic execution engine operates similarly to the EXE model and employs a dual approach to program analysis. Much like EXE, our engine uses both concrete and symbolic modes. As the program progresses through each step, concrete execution mirrors the symbolic execution, providing a comprehensive exploration of both states while discarding infeasible or challenging paths.

2 Implementation

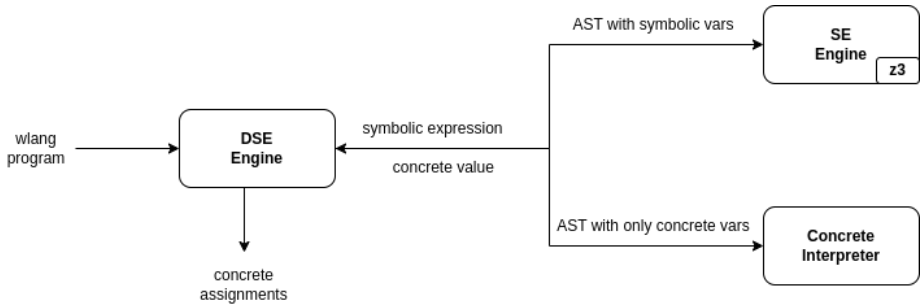

Fig. 1. The dataflow of DSE engine

As Figure 1 shows, we built the DSE engine on top of the SE engine from assignment 2 and concrete interpreter from the skeleton code. As mentioned before, DSE reduces to concrete execution when it is hard or unnecessary to execute symbolically. Our DSE engine dose so when checking Boolean conditions.

As Figure 2 shows, every program state will keep track of which variables are concrete or symbolic (the *concrete_varaibles* and *symbolic_variables* sets). We wrote a $VariableExtractor$ to extract all the variables used in the Boolean Conditions. If none of the variables are symbolic, the DSE engine will resort to the interpreter for a truth value of the condition. Symbolic Execution is unnecessary in this case. Neither path conditions gets updated, nor new program states gets spawned. However, if the Boolean condition contains any symbolic variables, DSE might update the path conditions and spawn new programs states with the symbolic expression of the condition returned by SE engine. More details can be found in the operational semantics of DSE for WHILE[5].

Listing 1.1 shows a example for symbolic and concrete conditions. At first condition check, x is symbolic, spawning two states, one entering the while loop and the other exiting the loop. After entering the loop, at the second check of the condition, x is concrete such that the loop terminates. The DSE only generates two programs states.

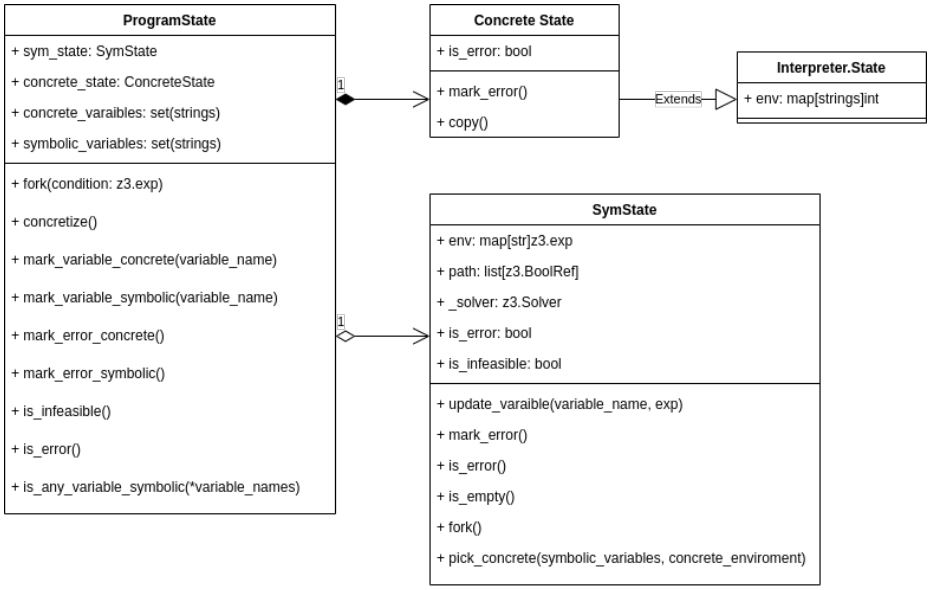

Fig. 2. The UML diagram for the Program-State class.

```
havoc x;
while x > 5 do {
    x := 4
}
--------
sym_state:
x: x!62
pc: [Not(5 < x!62)]
concrete_state:
x: 0
--------
sym_state:
x: 4
pc: [Not(Not(5 < x!62))]
concrete_state:
x: 4
--------
num_states: 2
```

Listing 1.1. while loop with a symbolic condition reducing to a concrete condition

Noticing that the DSE outputs are the same with SE output, readers may argue that DSE is not faster. Nonetheless, they ignored the fact the symbolic path condition $4 > 5$ is not added to second program state, saving computation in checking sastifiability of the branch for entering the loop again.

The DSE engine will concretize symbolic variables when symbolic execution is hard. We added a timeout of 1000 ms to the underlying solver. Whenever the solver outputs "unknown", the *pick_concrete* replace the symbolic expression in path condition for a symbolic variable with a $IntVal$ of the concrete value of the variable and add a path condition constraining the variable to be equal to the concrete value. As listing 1.2 shows, our implementation used the $z3.subsitute$ to concretize path conditions(line 18) and added the new path condition (line 19 and 20). If such efforts failed, the symbolic path is discarded (line 26 - 28).

```
1  def pick_concrete(self, symbolic_variables,
   concrete_environment):
2          res = self._solver.check()
3          if res == z3.unsat:
4              self._is_infeasible = True
5              return concrete_environment
6
7          symbolic_variables_copy = symbolic_variables.copy()
8          while res == z3.unknown or res == z3.unsat:
9              if len(symbolic_variables_copy) == 0:
10                 break
11             concretized_symbolic_variable =
   symbolic_variables_copy.pop()
12             rhs = z3.IntVal(concrete_environment[
   concretized_symbolic_variable])
13             lhs = self.env[concretized_symbolic_variable]
14             concretizing_path_condition = (lhs == rhs)
15             new_solver = z3.Solver()
16             for path_condition in self.path:
17                 new_path_condition = z3.substitute(
   path_condition, (lhs, rhs))
18                 new_solver.add(new_path_condition)
19             new_solver.add(concretizing_path_condition)
20             new_solver.set("timeout", 1000)
21             res = new_solver.check()
22             if res == z3.sat:
23                 self._solver = new_solver
24
25         if res != z3.sat:
26             self._is_infeasible = True
27             return concrete_environment
28     ......
```

Listing 1.2. smart concretization

2.1 Incremental Solving

As mentioned before, EXE-like DSE is collecting paths in a breadth-first manner. Because z3 scopes is a stack data structure and BFS algorithms work better with queues. Thus, it is hard to use the push and pop of z3 scopes for incremental

solving for our implementation. Using z3 assumptions will compound the path conditions.

Instead, we chose to copy the intermediate computation from the parent state to the children states. As listing 1.3 shows, we use the *translate* method to copy the parent $z3.ctx$, including intermediate assertions, models and solver settings, to the child state. The child state defaults to incremental solving mode[7].

```
    def fork(self):
        child_solver = z3.Solver()
        child_solver = self._solver.translate(child_solver.
    ctx)
        child = SymState(solver=child_solver)
    ......
```

Listing 1.3. smart concretization

3 Evaluation

```
havoc x, y;
i := 0;
product := 0;
while i < x do {
    j := 0;
    while j < y do {
        j := j + 1
    };
    product := product + j;
    i := i + 1
}
--------
......
sym_state:
x: x!22
y: y!23
i: 7
product: 56
j: 8
pc: [Not(Not(0 < x!22)), Not(Not(0 < y!23)), Not(Not(1 < y
    !23)), Not(Not(2 < y!23)), Not(Not(3 < y!23)), Not(Not(4
    < y!23)), Not(Not(5 < y!23)), Not(Not(6 < y!23)), Not(Not
    (7 < y!23)), Not(8 < y!23), Not(Not(1 < x!22)), 0 < y!23,
     1 < y!23, 2 < y!23, 3 < y!23, 4 < y!23, 5 < y!23, 6 < y
    !23, 7 < y!23, Not(8 < y!23), Not(Not(2 < x!22)), 0 < y
    !23, 1 < y!23, 2 < y!23, 3 < y!23, 4 < y!23, 5 < y!23, 6
    < y!23, 7 < y!23, Not(8 < y!23), Not(Not(3 < x!22)), 0 <
    y!23, 1 < y!23, 2 < y!23, 3 < y!23, 4 < y!23, 5 < y!23, 6
     < y!23, 7 < y!23, Not(8 < y!23), Not(Not(4 < x!22)), 0 <
     y!23, 1 < y!23, 2 < y!23, 3 < y!23, 4 < y!23, 5 < y!23,
```

```
    6 < y!23, 7 < y!23, Not(8 < y!23), Not(Not(5 < x!22)), 0
    < y!23, 1 < y!23, 2 < y!23, 3 < y!23, 4 < y!23, 5 < y!23,
     6 < y!23, 7 < y!23, Not(8 < y!23), Not(Not(6 < x!22)), 0
     < y!23, 1 < y!23, 2 < y!23, 3 < y!23, 4 < y!23, 5 < y
    !23, 6 < y!23, 7 < y!23, Not(8 < y!23), Not(7 < x!22)]

concrete_state:
x: 7
y: 8
i: 7
product: 56
j: 8
......
--------
num_states: 111
```

Listing 1.4. An WLang program that calculates products using nested while loops

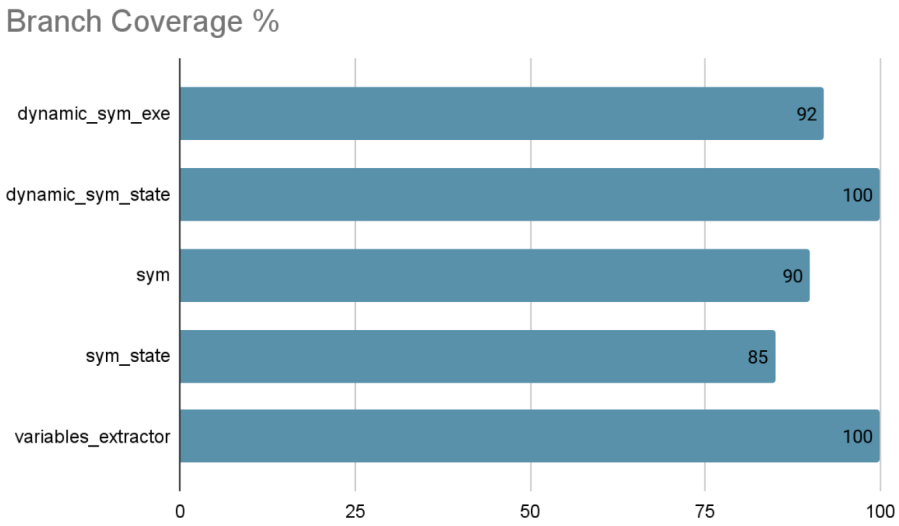

Fig. 3. Branch Coverage of added code

It is almost hard to test a DSE engine as to test a compiler. It is cumbersome to manually encode valid output for every testcase. Thus, we use the DSE engine to automatically generate computation results and check these results manually. If these outputs are valid, for regression testing, newer versions of the DSE engine should output the same. To demonstrate our confidence in this implementation, listing 1.4 shows a program that calculates the product of two symbolic variables x and y. We generate 111 states in the end, with the product correctly calculated.

(1 state exits the outer loop; 10 states enter the outer loop. For each of 10 states, 11 states are generated for inner loop.)

As Figure 3 shows, We achieved 92 percent in Dynamic Symbolic Execution. Parse_args and main are not able to be covered based on the fact that they are initialized before the symbolic execution or the coverage analysis begins. Dynamic Symbolic State and Variables extractor are fully covered as the branch coverage is 100 percent. We achieved 90 percent in Symbolic Execution Engine. The Parse_args and main functions are unable to be covered either because they are initiated before the execution or coverage analysis begins.

4 Conclusion

The project develops a Dynamic Symbolic Execution engine with a underlying a Z3 incremental solver. Our DSE implementation uses concrete execution to guide symbolic execution. The incremental solver, saving computation efforts, picks up the intermediate assertions and results from parent states. If the solver find a path too hard to solve, our DSE will concretize one variable at a time, hoping to unblock the solver. While our current implementation demonstrates the capabilities of DSE, we could have improved it in two ways. First, the interesting outputs are not the end computation result but the test vectors that can drive the program into an error state. We should have implemented a record for these test vectors. Second, We should have a smarter concretization strategy.The "Deferred Concretization in Symbolic Execution via Fuzzing" paper introduces an algorithm, implemented in the Colossus tool, that demonstrates significant improvements in coverage, divergence reduction, and test reproducibility, compared to the state-of-the-art symbolic execution engine, KLEE. The algorithm introduces a new category of symbolic values, called "symcrete values", to handle concretized values. It also designed a fuzz-based constraint solver to handle constraints on symcrete values. This approach is achievable by preserving information during concretization and defining values that can act as both symbolic and concrete. We could have build our DSE engine using this idea.

References

1. Arie Gurfinkel: Foundations: Syntax, Semantics, and Graphs. Slides, 5-8 (2023)
2. Arie Gurfinkel: Dynamic Symbolic Execution. Slides, 6-22 (2023)
3. Irlbeck, M. "Deconstructing dynamic symbolic execution." Dependable Software Systems Engineering 40, no. 2015 (2015): 26.
4. Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. "Interfacing with Solvers" 15-17
5. Gurfinkel, Arie . 2023. "Symbolic Execution Semantics for WHILE Language." February 3, 2023. https://git.uwaterloo.ca/stqam-1239/pdfs/-/raw/main/opsymexec.pdf.
6. Awanish Pandey, Phani Raj Goutham, and Subhajit Roy. Deferred Concretization in Symbolic Execution via Fuzzing. https://wcventure.github.io/FuzzingPaper/Paper/ISSTA19_Deferred.pdf. 228-238

7. Bjørner, Nikolaj, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. "Programming Z3." Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures 4 (2019): 148-201. Dec 2023