ECE 653 Team 18 Project Report (Choice 0)

Yitan Li[1] and Linsen Gao[1]

University of Waterloo, Waterloo, ON, N2L 3G1, Canada
y3432li@uwaterloo.ca, l78gao@uwaterloo.ca

Abstract. Symbolic execution is a powerful technique for software testing and program verification, offering formal assurances about program behavior. However, its practical application is hindered by the path explosion problem and the high computational cost of constraint solving. In this project, we tackle the limitations of the symbolic execution engine for a simple imperative language, WHILE Language, by augmenting it with a concolic execution strategy inspired by EXE-style frameworks, in conjunction with the incremental solving features of the Z3 SMT solver. Concolic execution combines symbolic and concrete execution, enabling systematic path exploration and automatic generation of high-impact test inputs—specifically crafted to expose critical program vulnerabilities such as buffer overflows and division-by-zero errors. Our implementation exploits Z3's incremental solving mode to handle evolving constraint sets more efficiently, thereby minimizing redundant computations and lowering overhead. This integration enhances the scalability and efficiency of symbolic execution, facilitating deeper analysis of complex programs. We evaluate our approach through performance benchmarks, demonstrating substantial gains in managing intricate test scenarios.

Keywords: Symbolic execution · Concolic testing · EXE-style · Z3 · Incremental solving.

1 Introduction

Symbolic execution is a well-established technique for software testing and validation, providing formal guarantees about program behavior by systematically exploring execution paths using symbolic inputs instead of concrete (actual) input values [1]. This approach enables the exploration of exhaustive program paths in a single analysis run, which can uncover edge cases and hidden bugs that concrete tests might miss. By integrating with an SMT solver, it can check path feasibility and automatically generate valid inputs efficiently for high-coverage test cases. Despite its strengths, traditional symbolic execution suffers from two major limitations: the path explosion problem, where the number of paths grows exponentially with the number of conditional branches in a program, and the difficulty of modeling realistic inputs (e.g., files, complex data structures, etc.) to the program.

WHILE Language is a small imperative language with its basic syntax and operational semantics described in a book by Hanne Riis and Flemming Nielson [2]. It includes assignments to local variables, if statements, while loops,

and simple integer and boolean expressions. In the assignments for this course, we have implemented a symbolic execution engine for WHILE Language. This project aims to implement additional advanced features for the symbolic execution engine, primarily to address the path exploration problem described above by incorporating EXE-style concolic execution and leveraging the incremental solving capabilities of the Z3 SMT solver for efficient concrete test generation for small programs written in the WHILE Language. Concolic execution combines concrete and symbolic execution, enabling more efficient path exploration by grounding symbolic reasoning in actual execution traces. Incremental solving further improves performance by retaining and reusing previously solved constraints, significantly reducing redundant computation.

In this project, we detail our integration of concolic execution with Z3's incremental solving and demonstrate the resulting improvements in performance and scalability. Our evaluation highlights the enhanced engine's ability to effectively handle complex and divergent programs, such as those involving nested branches and loops, outperforming traditional symbolic execution approaches in efficiency.

2 Background

2.1 Concolic execution in EXE-style

Concolic execution, a dynamic symbolic execution (DSE) approach in which the symbolic state is computed in parallel with concrete execution ("concolic"), uses a solver to generate new concrete inputs to increase coverage and is a powerful technique for software testing and program analysis.

EXE-style is one of the main approaches to concolic execution, where execution begins with concrete inputs (generated by fuzzing or a seed) while simultaneously tracking symbolic expressions along the execution path in parallel with the concrete execution [3]. This hybrid approach enables systematic exploration of program behavior by leveraging the precision of concrete execution and the generality of symbolic reasoning.

A hallmark of EXE-style concolic execution is its ability to fork execution at every branch point. Specifically, when the program encounters a conditional statement that causes a branch, the current concrete execution takes one path (e.g., branch1), guided by the current input in the concrete state. Then, symbolic execution updates the path condition to force execution into the alternative branch (e.g., branch2) and solves a corresponding new concrete state that follows branch2. These newly formed paths are then checked for feasibility using a constraint solver. If both branches are feasible, execution forks, allowing both paths to be explored in parallel. Otherwise, the infeasible path is terminated early. These general execution and branching steps provide the theoretical foundation for our implementation.

One of the most notable advantages of this approach is the automatic generation of high-impact test inputs, often referred to as "inputs of death" [3]. These

inputs are derived by solving the symbolic constraints collected along a path, ensuring comprehensive path coverage and highly effective automatic test case generation. As a result, the system can uncover critical errors such as memory overflows and division-by-zero bugs that are often missed by traditional testing techniques.

2.2 Z3 incremental solving mode

Incremental solving is a key optimization technique in modern constraint solvers, particularly in Z3, that enhances both performance and efficiency when dealing with evolving or complex constraint systems . Rather than solving constraints from scratch with each new query, incremental solving builds upon a base set of assertions and incrementally adds new constraints as needed. The central advantage of this approach lies in its ability to preserve solver state across multiple queries, allowing for the reuse of previous computations and reducing the need for repeated processing—especially when constraints are introduced gradually over time.

We chose to implement incremental solving using Z3's Scopes interface, specifically the push() and pop() operations. Assertions are managed within scopes that can be pushed and popped as needed. The push() operation creates a new local scope, while pop() reverts to the previous one. Any assertions added after a push() are automatically discarded when the corresponding pop() is executed. Behind the scenes, Z3 uses a one-shot solver for the initial check. If additional solver calls are made afterward, it automatically switches to incremental solving mode by default [4].

2.3 Combined Implementation

During concolic execution of WHILE Language, many similar queries are solved repeatedly, as potential new paths are added based on new constraints applied to the existing set of path conditions followed by feasibility checks. Therefore, the performance of concolic execution can be improved by leveraging the incremental solving capabilities of SMT solvers, which allow constraints to be added and withdrawn across multiple solver calls. This combined strategy significantly reduces redundant constraint solving and lowers the overall computational overhead associated with repeated feasibility checks.

3 Detailed design

3.1 Concolic execution in EXE-style

To implement concolic execution in EXE-style, we extend symbolic execution by maintaining a corresponding concrete state in parallel. This is achieve by defining **ConcolicState** class and **ConExec** class, the following code snippet shows the initialization of concolic excution.

```
class ConcolicState(object):
    def __init__(self, solver=None):
        self.env = dict()
        self.concrete = dict()
        self.path = list()
        self._solver = solver
        if self._solver is None:
            self._solver = z3.Solver()
        self._is_error = False

    def pick_concrete(self):
        res = self._solver.check()
        if res != z3.sat:
            return None
        model = self._solver.model()
        for (k, v) in self.env.items():
            self.concrete[k] = model.eval(v, model_completion
                =True).as_long()
        return z3.sat

    def fork(self):
        child = ConcolicState()
        child.env = dict(self.env)
        child.concrete = dict(self.concrete)
        child.add_pc(*self.path)

        return (self, child)
```

Listing 1.1: Symbolic State Initialization

In the **ConcolicState** class, we maintain two environments: **env** stores symbolic variables, while **concrete** stores their concrete parts. The **pick concrete** method uses the Z3 solver to extract a concrete model consistent with the symbolic constraints. The **fork** method duplicates both the symbolic and concrete state, enabling systematic path exploration with minimal recomputation. To orchestrate the simultaneous execution of symbolic and concrete states, we define the `ConExec` class. This class acts as the main concolic execution engine, interpreting WLang programs by traversing the abstract syntax tree. In the next part, we introduce how symbolic and concrete executions are run in parallel through the `run` and `visit` methods of `ConExec`

Expression Handling For expression nodes, each function returns a tuple comprising the symbolic and concrete interpretations of the expression. This dual representation allows us to maintain synchronized evaluation semantics across both execution modes.

```
    def visit_IntVar(self, node, *args, **kwargs):
        return kwargs["state"].env[node.name], kwargs["state"
            ].concrete[node.name]
```

```
3
4       def visit_BExp(self, node, *args, **kwargs):
5           kids = [self.visit(a, *args, **kwargs) for a in node.
                args]
6           kids_sym = [kid[0] for kid in kids]
7           kids_con = [kid[1] for kid in kids]
8           if node.op == "not":
9               assert node.is_unary()
10              assert len(kids) == 1
11              return z3.Not(kids_sym[0]), not kids_con[0]
12          fn = None
13          base = None
14          if node.op == "and":
15              fn = lambda x, y: x and y
16              base = True
17              assert fn is not None
18              return z3.And(kids_sym), reduce(fn, kids_con,
                    base)
19          elif node.op == "or":
20              fn = lambda x, y: x or y
21              base = False
22              assert fn is not None
23              return z3.Or(kids_sym), reduce(fn, kids_con, base
                    )
24          assert False
```

Listing 1.2: Expression Handling

visit IntVar: This function retrieves the symbolic and concrete values of a variable from the current state's environment. The symbolic value is a Z3 variable, while the concrete value is a native Python integer.

visit BExp: Boolean expressions are recursively evaluated, with results collected for both symbolic and concrete semantics. Z3 operators are used to form the symbolic constraints, while native Python operators compute the concrete truth values.

Statement Handling For statements that modify program state (such as assignments or havoc operations), we update both the symbolic and concrete representations of the environment. This ensures that state transitions remain aligned between the two execution modes.

```
1       def visit_AsgnStmt(self, node, *args, **kwargs):
2           for st in kwargs["state"]:
3               st.env[node.lhs.name], st.concrete[node.lhs.name]
                    = self.visit(node.rhs, *args, state = st)
4           return kwargs["state"]
5
6       def visit_HavocStmt(self, node, *args, **kwargs):
7           for st in kwargs["state"]:
```

```
            for v in node.vars:
                st.env[v.name] = z3.FreshInt(v.name)
                if v.name not in st.concrete:
                    st.concrete[v.name] = 0
        return kwargs["state"]

        self.concrete = didict()
```

Listing 1.3: Statement Handling

visit AsgnStmt: When processing assignment statements, we update both the symbolic environment (with Z3 expressions) and the concrete environment (with integer values). The symbolic state enables future path constraint formulation, while the concrete state reflects actual program behavior.

visit HavocStmt: The havoc statement introduces nondeterminism by assigning arbitrary values to variables. In symbolic execution, this is modeled with fresh symbolic variables. Concrete values default to 0 if the variable has not been assigned any concrete value yet.

Branch Handling in Conditional and Loop Constructs In EXE-style concolic execution, when the program reaches a branch point—such as an if or while statement—we first leverage concrete execution to follow one of the possible paths. This confirms that the current symbolic state is capable of taking that path. Next, we perform symbolic analysis by appending the negation of the branch condition, which allows the execution engine to explore the alternative, previously unvisited path. This approach ensures that both branches are systematically examined during the execution process.

```
    def visit_IfStmt(self, node, *args, **kwargs):
        then_states = []
        else_states = []
        for st in kwargs["state"]:
            cond_sym, cond_con = self.visit(node.cond, state
                = st)
            then_st, else_st = st.fork()

            then_st._solver.push()
            else_st._solver.push()
            then_st.add_pc(cond_sym)
            else_st.add_pc(z3.Not(cond_sym))

            if(cond_con):
                then_states.append(then_st)

                if else_st.is_empty():
                    else_st._solver.pop()
                else:
                    else_st.pick_concrete()
                    else_states.append(else_st)
```

```
            else:
                else_states.append(else_st)

                if then_st.is_empty():
                    then_st._solver.pop()
                else:
                    then_st.pick_concrete()
                    then_states.append(then_st)

        if then_states:
            self.visit(node.then_stmt, state=then_states)

        if node.has_else() and else_states:
            self.visit(node.else_stmt, state=else_states)

        kwargs["state"][:] = then_states + else_states

        return kwargs["state"]


    def visit_WhileStmt(self, node, *args, **kwargs):
        loop_states = []
        exit_states = []
        count = kwargs.get('count', 1)
        for st in kwargs["state"]:
            cond_sym, cond_con = self.visit(node.cond, state
                = st)
            lp_st, et_st = st.fork()

            lp_st._solver.push()
            et_st._solver.push()
            et_st.add_pc(z3.Not(cond_sym))
            lp_st.add_pc(cond_sym)
            if(cond_con):
                if count <= 10:
                    # lp_st.add_pc(cond_sym)
                    loop_states.append(lp_st)
                if et_st.is_empty():
                    et_st._solver.pop()
                else:
                    et_st.pick_concrete()
                    exit_states.append(et_st)
            else:
                exit_states.append(et_st)
                if count <= 10:
                    # lp_st.add_pc(cond_sym)
                    if lp_st.is_empty():
                        lp_st._solver.pop()
                    else:
                        lp_st.pick_concrete()
```

```
                        loop_states.append(lp_st)


        if count <= 10 and loop_states:
            self.visit(node.body, state=loop_states)
            self.visit(node, state=loop_states, count=count
                +1)
        kwargs["state"][:] = loop_states + exit_states
        return kwargs["state"]
```

Listing 1.4: Branch Handling in Conditional and Loop Constructs

Both if and while statements represent control-flow branching points in a program, and are treated similarly in EXE-style concolic execution. When such a branch point is encountered, concrete execution is used to determine the currently feasible path, while symbolic execution is employed to explore the alternate, uncovered path using Z3's constraint solving capabilities. This ensures full path coverage, even for branches that are not taken during the initial concrete execution.

For an if statement, the engine partitions states into two lists: one for the then-branch and another for the else-branch. Concrete execution determines the taken path, and Z3 is used to check the feasibility of the untaken branch. If both branches are feasible, the state is forked and path conditions are updated accordingly. This enables the engine to track and explore both paths in subsequent execution.

The while statement is conceptually treated as a repeating conditional branch, akin to an infinitely nested if. The engine uses concrete execution to determine whether to enter or exit the loop body. It then checks the feasibility of the opposite branch using Z3. If both the loop and exit conditions are feasible, the state is forked. To prevent infinite symbolic execution, the engine imposes an upper limit on loop iterations (e.g., 10). This strategy ensures both paths—looping and exiting—are symbolically analyzed while bounding the execution depth.

3.2 Z3 incremental solving mode

The main goal of using incremental solving via the scopes interface is to avoid repeatedly solving similar queries that differ by only one or two newly added constraints. This implies that the `push()` and `pop()` operations should be performed closely before and after the `solver.check()` operations. This situation mainly occurs in our implementation within the methods visit_IfStmt(self, node, *args, **kwargs), visit_WhileStmt(self, node, *args, **kwargs), visit_AssertStmt(self, node, *args, **kwargs), and visit_AssumeStmt(self, node, *args, **kwargs), as these involve the evaluation of relational expressions where new constraints may need to be added to the current path condition followed by feasibility checking, potentially leading to branching.

Specifically, following the steps of concolic execution, we perform a `push()` to save a snapshot of the current local scope before adding new constraints to the current path condition of the state and checking their feasibility. This implies that `push()` is called before `add_pc(constraints)` in our implementation. After checking the satisfiability of the updated state, we perform a `pop()` operation if the result is unsatisfiable—meaning the path with the newly added constraints is discarded and the previous state is restored. If the result is satisfiable, we keep and continue with the new state, and there is no need to perform a `pop()`. Therefore, `pop()` is called after the `solver.check()` operation, which is invoked within the `is_empty()` and `pick_concrete()` helper functions in our implementation.

Here is `visit_AssumeStmt(self, node, *args, **kwargs)` as a simple example of our implementation design for incremental solving using `push()` and `pop()`.

```
def visit_AssumeStmt(self, node, *args, **kwargs):
        new_states = []

        for st in kwargs["state"]:
            cond_sym, cond_con = self.visit(node.cond, state=
                st)

            if cond_con:
                st.add_pc(cond_sym)
                new_states.append(st)
            else:
                st._solver.push()
                st.add_pc(cond_sym)
                if st.pick_concrete() is None:
                    st._solver.pop()
                    continue

                new_states.append(st)

        kwargs["state"][:] = new_states
        return kwargs["state"]
```

Listing 1.5: Incremental Solving in visit_AssumeStmt

The incremental solving is placed inside the else branch, as this is where `pick_concrete()` and `solver.check()` are called. The sequence here is important: we first save the local scope with `push()`, then call `add_pc(constraints)` to add new constraints, followed by satisfiability checking in `pick_concrete()`, and finally use `pop()` to discard the newly added constraints if necessary.

4 Testing strategy

In order to achieve 100% statement and branch coverage, we performed unit tests on the concoli_new.py file. We began with programs containing simple if-else branches and basic while loops. To ensure full branch coverage, we needed to cover all possible execution paths in a typical if-else structure, so we included test cases such as if_no_else and if_no_then, among others. At the same time, we aimed to incorporate various types of relational and arithmetic expressions in these programs. We then progressed to more complex programs that combined multiple statement types, such as a while loop following an if-else branch, as well as programs that integrated havoc, assume, and assert statements to test both the correct buildup of valid paths and the accurate detection of error states. Additionally, we added some programs that include nested if-else branches and while loops to test the performance of unrolling loops up to the max_iteration limit and for further performance analysis. Finally, we used the coverage report to identify missed cases and generated new test cases to achieve full coverage.

As shown in Figure 1, the two uncovered lines are located within the main function and do not pertain to the core implementation of the concolic execution engine that is intended to be covered by unit tests. We therefore consider the coverage to be effectively complete

Fig. 1: Coverage Report

5 Results

We performed a runtime analysis to evaluate the performance of three different implementations: Symbolic Execution, Concolic Execution, and Concolic Execution with Incremental Solving. We used the same test set which contains the same two divergent test cases involving nested if-else branches and while loops for all three implementations. Each implementation was run 10 times, and the average time spent on the test cases was calculated. Additionally, we changed the max_iterations parameter to 10, 15, and 20 to observe how the test runtime divergent.

```
    def test_diverges(self):
        prg1 = """
        havoc x, y, z;

        while x > 0 do {
            if y < 5 then {
```

```
                    z := z + 1;
                    if z = 3 then {
                        y := y + 2
                    } else {
                        y := y + 1
                    }
                } else {
                    while z > 0 do {
                        z := z - 1
                    };
                    x := x - 1
                }
            };
            x := x + y + z
            """
            ast1 = ast.parse_string(prg1)
            engine = concolic_incremental.ConExec()
            st = concolic_incremental.ConcolicState()
            out = [s for s in engine.run(ast1, st)]
            self.assertEqual(len(out), 606)


    def test_execution_engine_diverges(self):
            prg1 = """
            havoc x, y;
            while x > 5 do {
                while x < y do {
                    y := y / 2 - 1
                };
                x := x - 2
            }
            """
            ast1 = ast.parse_string(prg1)
            engine = concolic_incremental.ConExec()
            st = concolic_incremental.ConcolicState()
            out = [s for s in engine.run(ast1, st)]
            self.assertEqual(len(out), 615)
```

Listing 1.6: two divergent test cases used in analysis

Data Analysis As illustrated in Figure 2, Concolic Execution with Incremental Solving consistently outperforms both plain Concolic Execution and pure Symbolic Execution across increasing loop unrolling limits. At `max_iterations` = 10, its advantage is already visible. As the unrolling limit increases to 15 and 20, the performance difference becomes more significant. This trend highlights the benefit of incremental constraint solving in mitigating the path explosion and constraint redundancy typical in symbolic analysis.

Symbolic Execution incurred the highest runtime due to its exhaustive exploration and constraint-solving overhead. Plain Concolic Execution performed moderately better but still suffered from solver re-invocation at each branch. In contrast, the incremental variant retained solver context across paths, leading to improved scalability.

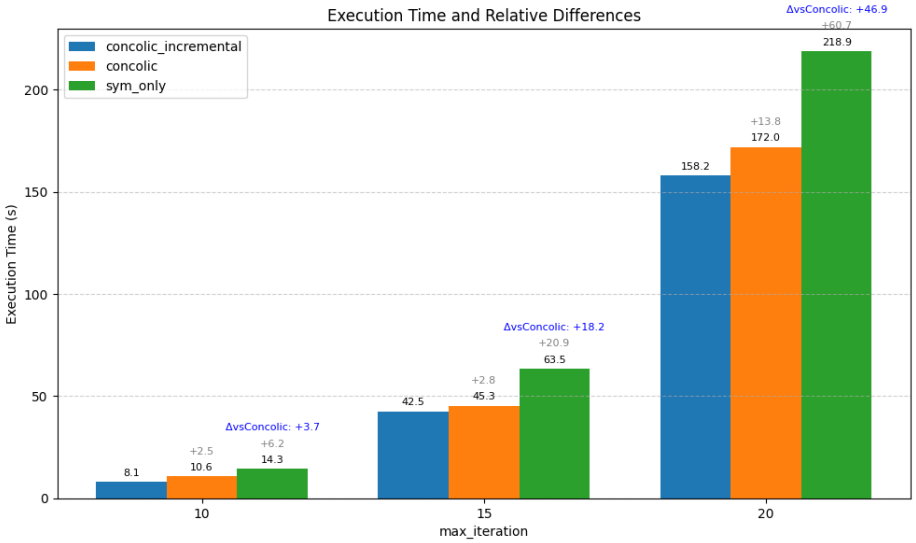

Fig. 2: Average execution time for each implementation under different `max_iterations` settings.

6 Conclusion

Through runtime analysis of three different implementations, we conclude that our EXE-style concolic execution benefits from incorporating solver state reuse. The data clearly demonstrates that this approach provides performance improvements as symbolic complexity increases, highlighting that concolic execution provides better performance than symbolic execution in cases of path explosion. For further evaluation, more detailed runtime and static analyses could be conducted to gain deeper insights into the performance and scalability of EXE-style concolic execution combined with incremental solving, which could hopefully be generalized to more complex programs in other languages.

Due to time constraints in the project, we limited our exploration to a finite number of loop iterations by setting a max_iteration, similar to traditional symbolic execution. For further work, we could try to implement the EXE-style loop unrolling for input-dependent loops that use a search algorithm to decide whether to continue unrolling the loop or to break out after forking execution.

One limitation of our project is that, since WHILE Language is a simple imperative language that doesn't support complex input types or environments (e.g., pointers, files, data structures), we did not demonstrate concolic execution's capability to handle such complex scenarios or perform corresponding automatic bug detection (e.g., detecting and generating test cases for dangerous operations like pointer dereferences). However, the continued exploration of potential assertion-failure branches, rather than stopping execution, helps demonstrate the strength of concolic execution in "generating inputs of death". Similarly, we did not demonstrate EXE-style concolic execution's ability to use concretization for handling complex cases such as virtual functions (function pointers) or system calls, since the WHILE language lacks support for them. However, these remain important cases worthy of further analysis, as symbolic execution struggles with them while concolic execution proves advantageous.

References

1. King, J.C.: Symbolic execution and program testing. Communications of the ACM, vol. 19, no. 7, pp. 385–394. ACM, New York (1976). https://doi.org/10.1145/360248.360252
2. Nielson, H. R., Nielson, F.: Semantics with Applications: An Appetizer. Springer-Verlag New York, Secaucus, NJ, USA (2007)
3. Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., Engler, D. R.: EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS' 06), pp. 322–335. ACM, New York (2006). https://doi.org/10.1145/1180405.1180445
4. Bjørner, N., de Moura, L., Nachmanson, L., Wintersteiger, C. M., Zhang, Z., Bowen, J. P., Liu, Z.: Programming Z3. In: Engineering Trustworthy Software Systems (ETSS 2019), vol. 11430, pp. 148–201. Springer International Publishing AG (2019). https://doi.org/10.1007/978-3-030-17601-3_4