# ECE 653 Software Testing, Quality Assurance, and Maintenance Project Report

Mingyang Xu[1] and Zekun Zhu[2]

[1] University of Waterloo, Waterloo, Canada
m283xu@uwaterloo.ca
[2] University of Waterloo, Waterloo, Canada
z93zhu@uwaterloo.ca

**Abstract.** We successfully implemented two advanced features: EXE-style concolic execution and incremental solving of z3 on the symbolic execution engine for wlang. To verify the implementation's correctness, we built an extensive test suit which achieved 100% statement coverage and branch coverage. On top of this, our implemented incremental solving achieved an impressive 62.6% execution time reduction in comparison to the original "SymExec" execution engine.

## 1 Introduction

In this project, we have been tasked with implementing two additional advanced features for an imperative language. Our project builds upon the symbolic execution engine and verification engine developed in previous assignments. We have chosen to implement "Concolic execution in EXE-style" and "Utilize the incremental solving mode of Z3". Firstly, we will elucidate the theoretical foundations of these features. Subsequently, we will outline our implementation approach, encompassing the code structure and the execution procedure. Finally, we will present our testing strategy and the corresponding test results. To use "EXE-style concolic execution", please pass `-f 1` flag to the sym.py program. To use "incremental solving mode", please use `-f 2` flag.

## 2 Concolic Execution in EXE-style

### 2.1 Theoretical Foundations

Dynamic symbolic execution (DSE) is a well-established method for automatically creating tests to achieve higher coverage in a program. It differs from static (classic) symbolic execution in three main ways. Firstly, it executes or interprets the program with a concrete state. Secondly, it computes the symbolic state in parallel with concrete execution. Thirdly, the solver produces new concrete inputs to enhance coverage. There are two primary styles: EXE-style and DART-style. We have chosen the EXE-style for our programming feature.

The EXE-style dynamic symbolic execution operates according to the following guidelines. Its program state is represented as a tuple (Concrete state, Symbolic state). Initially, when generating a variable, a concrete value is simultaneously generated (using fuzzing or a seed). At each execution step, the concrete state is updated by executing a program instruction concretely, while the symbolic state is updated through symbolic execution. At each branch point, the concrete execution takes one branch based on the concrete value result (referred to as branch 1). Subsequently, the symbolic execution updates the current input to direct execution into a different branch (referred to as branch2). A new concrete state is created in branch 2 with compatible concrete values. Both branches are explored, with branch 2 being explored first.

To implement an EXE-style dynamic symbolic execution, certain detailed modifications need to be made. I will provide further illustrations in the following section.

### 2.2 Code structure

The key components of EXE-style dynamic symbolic execution are realized through two objects: EXEState and EXEExec. Within an EXEState, we have two member objects: symState and conState. ConState, inherited from SymState, maintains the current concrete state. ConExec, inherited from SymExec, rewrites a function to facilitate concrete execution. SymState and SymExec correspond to our A2 and A3 assignments.

```
class ConState(SymState)

class ConExec(SymExec)

class EXEState(SymState):

    def __init__(self):
        self.symState = SymState()
        self.conState = ConState()
        #......

class EXEExec(ConExec)
```

They are all located in the sym.py.

### 2.3 Implementation Detail

```
class ConState(SymState):
    def __init__(self, solver=None):
        super().__init__(solver)
        self.conEnv = {}
```

```
    def is_con_empty(self):
        fake_st = self.fork()[1]
        fake_path = self.path[:]
        new_fake_path = []
        solver = z3.Solver()
        for path in fake_path:
            for k, v in fake_st.env.items():
                if v not in path.children():
                    continue
                new_path = z3.substitute(...)
                new_fake_path.append(new_path)
                solver.append(new_path)
        is_empty = solver.check() == z3.unsat
        return is_empty

    def add_concrete_var(self, name, conValue=None):
        if conValue is None:
            if name not in self.conEnv:
                self.conEnv[name] = 1
        else:
            if type(conValue) == z3.IntNumRef:
                self.conEnv[name] = conValue.as_long()
            else:
                self.conEnv[name] = conValue
```

Within a ConState, we have a member dictionary called conEnv, which constitutes the primary component of the concrete state. It stores data in the following format: `varName: concreteValue`. VarName is a string identical to the varName saved in SymState.env. ConcreteValue is an integer value generated to represent the concrete value of a variable, which will be executed and updated subsequently.

`is_con_empty()` is used to verify if the current concrete state is consistent with the path condition. To test its feasibility, we employ `z3.substitute` to substitute all symbolic execution variables with concrete values. For instance, if the current path condition is `x == 9` and the current concrete value of x is 1, we modify the path condition to `1 == 9` and use `solver.check() == z3.unsat` to assess its feasibility.

`add_concrete_var()` is employed to introduce a new concrete variable to the concrete state, which will subsequently be stored in conEnv. If the value is added from havoc, a new value is generated for it. While it would be more appropriate to use randomization, in this instance, we simply assign them a value of 1 to ensure a consistent behavior for the test case. If a constant integer value is assigned to a variable, check its type and do the assignment.

```
class ConExec(SymExec):
    def visit_AExp(self, node, *args, **kwargs):
```

```
        kids = [self.visit(a, *args, **kwargs) ...]
        # Calculate concrete result
        conKids = kids[:]
        inverseDict = {...}
        for i in range(len(conKids)):
            if type(kids[i]) != z3.IntNumRef:
                key = inverseDict[kids[i]]
                conKids[i] = kwargs['state'].conEnv[key]
            if type(conKids[i]) == z3.IntNumRef:
                # Convert concrete result to int number
                conKids[i] = conKids[i].as_long()
        # Following is the same with SymExec
```

The only difference in `ConExec` is `visit_AExp()`. When we do arithmetic in symbolic execution, we add a new variable and add this function to path condition. In concrete execution, we need to replace the value with the integer and do arithmetic execution.

```
class EXEState(SymState):
    def __init__(self):
        self.symState = SymState()
        self.conState = ConState()
        self._lastFeasibleState = []
        self.curStIndex = 0

    def update_concrete_value(self):
        res = self.conState._solver.check()
        if res != z3.sat:
            return
        model = self.conState._solver.model()
        st = int.State()
        for (k, v) in self.conState.env.items():
            if type(model.eval(v)) != z3.IntNumRef
            and k in self.conState.conEnv:
                st.env[k] = self.conState.conEnv[k]
            else:
                st.env[k] = model.eval(v).as_long()
        self.conState.conEnv = st.env
    # ......
    def is_empty(self):
        return self.symState.is_empty()
    # ......
    def __str__(self):
        #......
```

`EXEState` is the state we used in the `EXEExec`. We have a `symState` and a `conState`. The `self._lastFeasibleState` is used in if branch, when we first

compute a concrete path, we save that state in `self._lastFeasibleState` to execute later. '`update_concrete_value()`' is used to update the current concrete value based on the path condition. It is used when the concrete state is infeasible but a symbolic state is feasible.

Besides, we also rewrite other functions like '`is_empty()`' and '`__str__()`' to adapt the new implementation.

In the implementation detail of `EXEExec`, I will show the code skeleton and explain the idea within three main functions: '`visit_IfStmt()`', '`visit_StmtList()`' and '`visit_WhileStmt()`'.

```
def visit_IfStmt(self, node, *args, **kwargs):
    cond = self.visit(node.cond, *args, **kwargs)
    trueStateKwargs = copy.deepcopy(kwargs)
    falseStateKwargs = copy.deepcopy(kwargs)
    # ......
    # Find later state
    trueStateKwargs['state'].add_pc(cond)
    if not trueStateKwargs['state'].is_con_empty():
        laterStateKwargs = trueStateKwargs
    falseStateKwargs['state'].add_pc(z3.Not(cond))
    if not falseStateKwargs['state'].is_con_empty():
        laterStateKwargs = falseStateKwargs

    # Find current state
    currentStateKwargs = trueStateKwargs # ...

    # Process later state
    if laterStateKwargs == trueStateKwargs:
        trueNewStates = self.visit(...)
        # ......
    elif laterStateKwargs == falseStateKwargs:
        if node.has_else():
            falseNewStates = self.visit(...)
            # ......

    # Process current state
    currentStateKwargs['state'].update_concrete_value()
    tmp = [currentStateKwargs['state']]
    if node.has_else()
    and currentStateKwargs == falseStateKwargs:
        curNewStates = self.visit(...)
    elif currentStateKwargs == trueStateKwargs:
        curNewStates = self.visit(...)

    # Save later state in current state
    if tmp:
```

```
        tmp[0].add_later_explore_state(x)
        # ......
```

In the `visit_IfStmt()`, we initially create two branches: one for entering the if condition and the other for entering the else condition. Then, we assess the feasibility of its concrete state at each branch. If it is feasible, we judge it as a later state, and the other as the current state. We then proceed with concrete and symbolic execution for the current state. The later state is then handled in the `visit_StmtList()`.

```
def visit_StmtList(self, node, *args, **kwargs):
    # ......
    while True:
        if not trackStateList:
            # When current running states are empty
            trackStateList.append(laterStateStack.pop())
        for state in trackStateList:
            freshKwargs = copy.deepcopy(kwargs)
            freshKwargs['state'] = state
            if state.curStIndex >= len(node.stmts):
                # add state to final result
                resultStateList.append(state)
                continue
            # run current state by curStIndex
            stmt = node.stmts[state.curStIndex]
            newStates = self.visit(...)
            trackNewGenStates += newStates
            for s in newStates:
                # Save concrete running result
                laterStates += s.get_later_state()
            self._assign_index(newStates, ...)
            self._assign_index(laterStates, ...)
        laterStateStack.extend(laterStates)
        trackStateList = trackNewGenStates
    finalStates = []
    for state in resultStateList:
        # Check empty state......
        finalStates.append(state)
    return finalStates
```

In the `visit_StmtList()`, we iterate and run the target statement list. The special occasion we need to solve here is when we are processing a branch state, we need to first explore the concrete state result, then save it in `laterStateStack` and explore the negation of that condition. At last, when we find that our current tracked state list is empty, we pop from the `laterStateStack` and process the later state we saved before.

`curStIndex` attribute is used to record the current running statement, it will help to figure out what is the next state to run.

```
def visit_WhileStmt_NoInvariant(self, node, ...):
    freshKwargs = copy.deepcopy(kwargs)
    finalStates = []
    for i in range(11):
        cond = self.visit(...)
        trueStateKwargs = copy.deepcopy(freshKwargs)
        falseStateKwargs = copy.deepcopy(freshKwargs)
        trueStateKwargs['state'].add_pc(cond)
        falseStateKwargs['state'].add_pc(z3.Not(cond))

        # Check if it is not concretely feasible
        if trueStateKwargs['state'].is_con_empty():
            trueStateKwargs['state'].up_concrete_value()

        # Check if it is concretely feasible, visit
        if not trueStateKwargs['state'].is_con_empty():
            self.visit(node.body, ...)

        # Symbolic execution is feasible, exit the loop
        if not falseStateKwargs['state'].is_empty():
            falseStateKwargs['state'].up_concrete_value()
            finalStates.append(falseStateKwargs['state'])

        # assign trueStateKwargs to freshKwargs
        freshKwargs = trueStateKwargs

    # Add last state to final states
    if not finalStates and not f...['state'].is_empty():
        finalStates.append(freshKwargs['state'])
    return finalStates
```

In the `visit_WhileStmt_NoInvariant()`, we create two distinct branches and assess the feasibility of the concrete state for the true condition. If it is not feasible, we update its concrete value. Subsequently, we visit the body of the while loop and append the negation of the condition to the path condition. Since we only append the negation of the condition in each iteration, we iterate a total of 11 times to simulate the 10 runs of the while loop body. If none of the negated conditions are feasible, we add the true state as the final state.

### 2.4 Testing

In order to test the dynamic symbolic execution in EXE-style, we should use `engine = sym.EXEExec()` and `st = sym.EXEState()` to generate engine and state.

The test cases, from `test_exe1()` to `test_exe6()` and `test_exe_while1()` to `test_exe_while4()` are situated in the file `test_sym.py`. These test cases evaluate the behavior of EXEExec under if and while statements. Under our implementation, we are able to reach full branch coverage of this section of code.

## 3 Incremental Solving Mode of Z3

### 3.1 Theoretical Foundations

Symbolic Execution is a type of automated testing technique able to systematically explore all execution paths in the program. While it is a promising technique and has gained significant interest in the recent years [3], it remains computationally expensive. Despite various advancements in the constraint solving technique, it continues to be the bottleneck on the symbolic execution time. Two optimization approaches have been proposed to increase constraint solver performance:

1. **Eliminate irrelevant constraint:** An important characteristic of path condition constraint solving is that despite many variables and constraints in the program, the program branch generally only depends on a small fraction of variables. Therefore, constraint solver performance can be improved by removing the constraints that are irrelevant to the decision making of current branch flow.
2. **Incremental solving:** Constraints generated from symbolic execution share very similar constraints due to the program code contains fixed set of static branches. Therefore, it is possible to leverage caching to reuse the resolved results from previous constraint solving. For example, if a superset of constraints has been resolved and cached, the subset of constraint can be easily solved from cache because removing an constraint does not violate existing solution.

For implementing an advanced feature on the symbolic execution engine for wlang, incremental solving is a promising candidate for increasing constraint solver performance. In our implementation, we leverage both incremental solving interfaces “Scopes” and “Assumptions” provided by z3 solver [2]:

1. **Scopes:** The scopes z3 incremental solving interface works well with the push/pop strategy of incremental solving [1]. Push/pop strategy works by first pushing an assertion frame on the stack for solving the constraint. After the assertion, the frame is popped off the stack to reset the solver’s state. The push/pop strategy exploits the assumption that some constraints are shared between queries. The assumption holds particular true in the execution engine for wlang. In checking the satisfiability of if else branch, the continuity of while loop and results of assertion, a significant portion of constraints are shared between branches of execution. Therefore, this strategy can potentially result in significant performance improvement for the execution engine [4].

2. **Assumptions:** The assumptions z3 incremental solving interface works well for the check-sat-assuming strategy [1]. Check-sat-assuming strategy takes a list of boolean variables and use them to activate the assumptions. This way, the solver no longer needs to explore the search space that is not relevant to the current execution path. This incremental solving strategy works particular well with breath-search-first exploration strategy.

For the incremental solving feature implemented in this project, we leverage both solving interfaces in their respective strength areas. "Assumptions" interface is used as the backbone of incremental solving. Whenever a path is not shared by all paths, the execution of this path is governed by this interface. For example, all the code execution in the "then" block of if branch being true is added to this interface. The execution state maintains a set of activated assumption boolean variables for a particular execution path. "Scopes" interface is used in checking satisfiability of if else branch, entering while loop body and checking the assert results. In these cases, the branch condition adds a small number of constraints to the existing constraints. Reuse the cached results of solving existing constraints significantly increase execution speed.

### 3.2 Implementation Logic

The core of incremental solving is realized in IncState and IncExec classes. IncState inherits from SymState, it adds functionalities to track boolean variables for activating "Assumptions" solving interface, and provides functionalities to set backtracking point for push/pop strategy.

```
class IncState(SymState):
    def __init__(self, solver=None):
        ...
        # activation variables
        self.activationLiterals = set()
        # precondition activation literal
        self.precondition = None
        ...
```

In the instantiation of `IncState`, we create an empty set to track the boolean variables for activating assumptions for this particular path. This makes sure all assumptions relevant to this path is set to True, and none of the irrelevant assumptions are introduced to constraint solving. The variable precondition is of 'BooleanRef' type of z3. Every time it enters branch of if else or while loop, every code execution (constraints) of branch body is set under the assumption that this assumption variable is True. This makes sure the code executed under a particular path does not affect other paths.

The idea of how to use this 'precondition' variable is best illustrated by the code snippet below.

```
def add_pc(self, *exp):
```

```
        """Add constraints to the path condition"""
        if self.precondition != None:
            for e in exp:
                self.path.append
                    (z3.Implies(self.precondition, e))
                self._solver.append
                    (z3.Implies(self.precondition, e))
        else:
            self.path.extend(exp)
            self._solver.append(exp)
```

If the 'precondition' variable is not set, the constraint is added to all paths of execution. When the 'precondition' variable is set, the constraint only belongs to this particular path.

The `IncState` class also comes with helper functions to wrap around the Z3 solver incremental solving "Scopes" and "Assumptions" interfaces, illustrated below. This increases maintainability of code and decouples z3 solver to code logic.

```
def get_new_activation_literal(self):
        return z3.FreshBool()

    def add_activation_literal(self, activationLiteral):
        if activationLiteral in self.activationLiterals:
            print("Warning:activation␣variables"+
            "already␣defined.␣This␣shouldn't␣happen!")
        self.activationLiterals.add(activationLiteral)

    def add_state_backtrack_point(self):
        self._solver.push()

    def revert_state_backtrack(self):
        self._solver.pop(num=1)

    def is_empty_withActivation(self):
        """Check whether the current symbolic state has
        any concrete states, use assumption"""
        res = self._solver.check
            (*self.activationLiterals)
        return res == z3.unsat

    def path_activation_fork(self):
        """Fork the current state into two identical
        states that can evolve separately
           Retain the same solver
        """
```

```
        child = IncState(solver=self._solver)
        child.env = dict(self.env)
        child.activationLiterals = set
            (self.activationLiterals)
        child.precondition = self.precondition
        child.add_pc(*self.path)

        return (self, child)
```

Building on top of functionalities from `IncState`, the push/pop strategy is used to check the branch condition satisfiability. For example, in checking the satisfiability of if else branch, we use the exactly same solver with different added constraints. First a backtracking point is set, then the condition of if branch is pushed to solver to check satisfiability. After this, solver is reverted back to original state, then the else branch condition is added to the solver. This way the solver reuses the constraints solved in previous code execution.

```
# Test satisfiability of both branches
        originalState, testBranch =
            st.path_activation_fork()
        # trueState
        testBranch.add_state_backtrack_point()
        testBranch.add_pc(cond)
        trueStateSatisfy = not testBranch.
            is_empty_withActivation()
        testBranch.revert_state_backtrack()
        # falseState
        testBranch.add_state_backtrack_point()
        testBranch.add_pc(z3.Not(cond))
        falseStateSatisfy = not testBranch.
            is_empty_withActivation()
        testBranch.revert_state_backtrack()

        # Create two potential states here,
        # if statement being True
        # and statement being False
        # trueState enters then block,
        # falseState enters else block
        trueState, falseState =
            originalState.path_activation_fork()

        newStates = []

        # Add conditions
        if trueStateSatisfy:
            ...
```

The use of "Assumptions" is best shown on code snippet below. The execution of "then" body is under the assumption that the path satisfies if condition. To isolate this code execution (constraints) from other paths, we first set the "preconodition" variable.

```
if trueStateSatisfy:
    trueStateActivatorSymbol =
        trueState.get_new_activation_literal()
    trueState.set_precondition
        (trueStateActivatorSymbol)
    trueState.add_pc(cond)
    trueState.add_activation_literal
    (trueStateActivatorSymbol)

    # Enter then state
    trueNewStates = self.visit
        (node.then_stmt, *args, state=trueState)
    trueNewStates = self._object_tolist(trueNewStates)
    newStates = newStates+trueNewStates
```

The "precondition" variable is removed at exiting the if else block. This way later code execution does not depend on this "precondition".

```
# Clear all precondition symbols as the
# entire if else block exit
    for state in newStates:
        state.clear_precondition()
```

### 3.3 Performance Comparison

To evaluating the effectiveness of implemented incremental solving mode, we build two sets of testing cases in "test execution speed.py" and measure its execution time using coverage program. The first set of test cases consist of `assert`, `if else` and `while` without invariant, in addition to all the basic operations such as value assignment. The second set of test cases consist of `while` loops with invariants, in addition to basic operations. We separate the test cases to different categories in hope to better understand if a particular category benefits more from incremental solving. To better illustrate the execution time differences, each test is running 50 times for difference to accumulate. To minimize external environment affect on the execution time, all background software programs were terminated and each test suit was run 3 times to calculate average execution time.

The results of test suit execution time are collected in table below[1]:

[1] Collected result screenshots are stored in "Performance Evaluation" directory in the root repository

**Table 1.** Execution of First Test Suite

| Test Run | SymExec | Incremental Solving |
|---|---|---|
| Run 1 | 21.244s | 7.426s |
| Run 2 | 20.891s | 7.598s |
| Run 3 | 21.235s | 7.558s |
| Average | 21.123s | 7.527s |

With implemented incremental solving, the execution time shows an impressive 64.4% reduction. It's almost three times faster.

**Table 2.** Execution of Second Test Suite

| Test Run | SymExec | Incremental Solving |
|---|---|---|
| Run 1 | 13.320s | 5.188s |
| Run 2 | 13.040s | 5.110s |
| Run 3 | 13.134s | 5.124s |
| Average | 13.164s | 5.140s |

This time the incremental solving achieves a consistent 61.0% execution time reduction.

From the collected results, we are confident to say incremental solving has consistently and significantly improved the execution time of symbolic execution engine. The combination of both "Scopes" (push/pop strategy) and "Assumptions" (check-sat-assuming strategy) leverage previous solved constraints while maintaining execution correctness.

## 4 Conclusion

Throug this course project and previous assignments on developing symbolic execution engine, we have gained valuable insights on how symbolic execution works and how more advanced form such as dynamic symbolic execution(DSE) solves the challenges faced by symbolic execution. In addition to this, we had hands-on-experience on how the performance of constraint solver can be improved, and our implementation of incremental solving symbolic execution engine fully supports the theory of incremental solving can improve performance. Overall, this course project highlights many important aspects of software testing and connects the dots between theory and practice. We appreciate Professor. Arie and TAs' efforts to deliver this valuable course.

## References

1. Bembenek, A., Ballantyne, M., Greenberg, M., Amin, N.: Making incremental smt solving work for logic programming systems (2020), https://api.semanticscholar.org/CorpusID:219937303

2. Bjørner, N.: https://theory.stanford.edu/ nikolaj/programmingz3.html
3. Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. Commun. ACM **56**(2), 82–90 (feb 2013). https://doi.org/10.1145/2408776.2408795, https://doi.org/10.1145/2408776.2408795
4. Liu, T., Araújo, M., d'Amorim, M., Taghdiri, M.: A comparative study of incremental constraint solving approaches in symbolic execution. In: Yahav, E. (ed.) Hardware and Software: Verification and Testing. pp. 284–299. Springer International Publishing, Cham (2014)