Concolic Execution Engine in EXE-Style using Z3 Incremental Solving Mode

Gabriel Olivotto and Leo Zhang

University of Waterloo, Waterloo, Ontario, Canada
{golivott, l536zhan}@uwaterloo.ca

Abstract. In this project, we developed a concolic execution engine in the style of EXE, designed specifically for the WHILE programming language. Our engine leverages the Z3 SMT solver's incremental solving capabilities to efficiently handle symbolic execution. By integrating Z3, we enabled the engine to perform path exploration and constraint solving dynamically, allowing for more comprehensive coverage of program paths and detection of potential errors. The incremental nature of Z3 facilitated the efficient handling of evolving constraints, thus optimizing the overall execution time. This work aims to improve the precision and performance of concolic testing, providing a valuable tool for developers in the verification and analysis of WHILE programs.

Keywords: Concolic Execution · EXE-style Engine · WHILE Programming Language · Z3 SMT Solver · Incremental Solving

1 Introduction

Software testing and verification are fundamental to ensuring the reliability and security of software systems. Traditional testing methods often struggle to achieve comprehensive coverage, potentially leaving critical issues undetected. Concolic execution, which systematically explores both concrete and symbolic program paths, has emerged as a promising technique to enhance test coverage.

This project focuses on implementing two key features to improve concolic execution for the WHILE programming language. The first feature involves developing an EXE-style concolic execution engine. This engine integrates concrete and symbolic execution, enabling the exploration of a broad spectrum of program behaviors and the identification of potential issues that standard testing might miss.

The second feature leverages the incremental solving capabilities of the Z3 SMT solver to optimize symbolic execution performance. Symbolic execution often requires solving similar queries repeatedly, which can be computationally expensive. By utilizing Z3's incremental solving mode, we can dynamically add and remove constraints between solver calls, significantly enhancing efficiency. We explored both available incremental interfaces—Scopes and Assumptions—and ultimately settled on using Assumptions for its ease of integration.

This report details the design and implementation of these features, highlighting the improvements and challenges encountered. We provide a comprehensive evaluation of the system's performance, demonstrating the effectiveness of our approach in enhancing test coverage and execution efficiency for WHILE programs. The subsequent sections outline the system design, implementation specifics, and evaluation results, culminating in a discussion of future directions and potential enhancements.

2 System Design

2.1 Architecture Overview

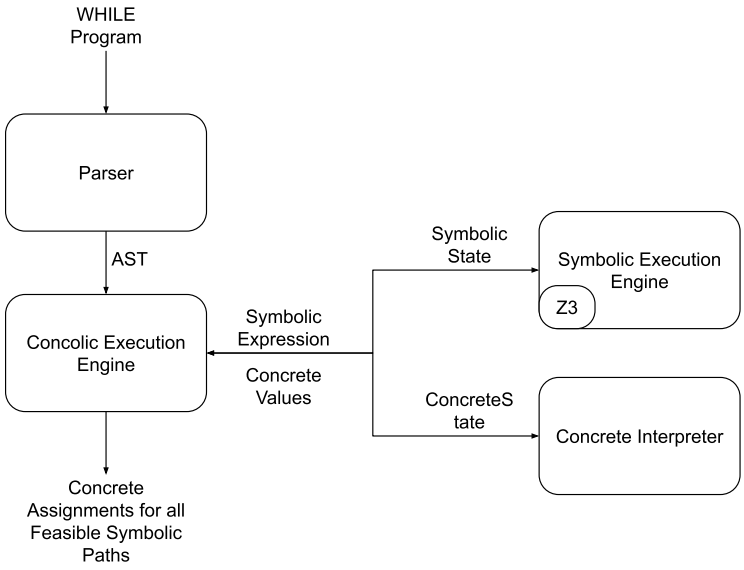

Fig. 1. Architecture diagram

The concolic execution engine developed in this project is designed in the style of EXE and integrates with the Z3 SMT solver to leverage its incremental solving capabilities. The system architecture comprises several key components: the parser, the interpreter, the symbolic executor, and the Z3 interface. The parser processes the WHILE programming language source code into an abstract syntax tree (AST). The interpreter executes the program concretely, while the symbolic executor explores symbolic paths by maintaining and manipulating symbolic variables. The Z3 interface handles the interaction with the SMT solver, using incremental solving techniques to manage path constraints efficiently.

2.2 Core Components

The system is composed of the following core components:

Parser: The parser reads the WHILE program and converts it into an AST, representing the program's structure. This AST serves as the basis for both concrete and symbolic execution.

Concrete Interpreter: The interpreter component, detailed in `int.py`, executes the AST in a concrete state. It tracks the concrete values of variables and the program's control flow. This component helps identify actual program paths that might be taken during execution.

Symbolic Executor: Located in `sym.py`. It maintains symbolic states, where variables can represent both concrete values and symbolic expressions. The symbolic executor explores different execution paths by forking the state at conditional branches, generating symbolic path conditions for each path. These path conditions are essential for checking the feasibility of the paths using the Z3 SMT solver.

Concolic Executor: Located in `exe.py`. It is the heart of the concolic execution engine and performs the execution of both the concrete interpreter and symbolic executor for each state in the style of EXE. The concolic executor uses the different execution paths discovered by the symbolic executor and uses Z3 to solve for possible concrete assignments that explore all paths through the program.

Z3 Interface: The Z3 interface manages the communication with the Z3 SMT solver. It is responsible for encoding path conditions as SMT problems and querying Z3 to determine their satisfiability. The interface leverages Z3's incremental solving features, particularly the Assumptions interface, to efficiently handle evolving constraints during symbolic execution.

2.3 Data Flow

The data flow in the system begins with the parsing of the WHILE program into an AST. The concolic execution engine extends the base `ast.Visitor` to execute each statement in the program. The interpreter runs this AST in a concrete state, while the symbolic executor operates in parallel, maintaining a symbolic state. At each branch in the program, the concolic executor forks the current state and generates path conditions based on the symbolic state to ensure both possible paths through the program are evaluated. These conditions are passed to the Z3 solver via the Z3 interface to check their feasibility.

If the Z3 solver finds both path conditions are feasible, the symbolic executor continues exploring each path, potentially generating new symbolic states and path conditions. Infeasible paths are pruned, focusing the exploration on viable execution paths. The use of Z3's incremental solving capabilities allows the system to efficiently manage constraints by adding and withdrawing assumptions as the execution progresses, rather than recomputing from scratch for each new path condition.

2.4 Design Choices

Several critical design choices were made during the development of this concolic execution engine:

Incremental Solving with Z3: The integration of Z3's incremental solving capabilities is a key feature of our concolic execution engine. Incremental solving allows the SMT solver to efficiently manage and manipulate a series of related constraints without the need to recompute from scratch for each new set of constraints. This approach significantly reduces the computational overhead associated with symbolic execution, particularly in complex programs with numerous branching paths. By maintaining a consistent set of assumptions and only adding new constraints as needed, we reduce the number of solver calls required to explore all paths. This approach avoids redundant computations and speeds up the overall process.

We explored two main incremental solving interfaces provided by Z3: Scopes and Assumptions[2]. The Assumptions interface was chosen for its flexibility and ease of use in managing evolving path conditions. Unlike the Scopes interface, which requires a stack-like mechanism for adding and removing constraints, the Assumptions interface allows each symbolic state to maintain its own set of constraints. This method simplifies the handling of multiple states, especially in scenarios involving nested conditional statements or loops.

In practice, each symbolic state in our engine maintains a separate solver instance. As the engine explores different execution paths, it uses the Assumptions interface to add constraints representing the current path conditions. When a new branch is encountered, the engine forks the current state and adds the appropriate constraint (or its negation) to each forked state. The solver then determines the feasibility of these states by checking the satisfiability of the constraints under the current assumptions.

Concretization of complex sections: A desired characteristic of any concolic execution engine is the ability to concretize complex code to aid in the symbolic execution. Due to the lack of functions in the WHILE programming language identifying complex sections of code to concretize is difficult. As such the only place where the symbolic state uses the concrete state to simplify execution is when while loops exceed 10 iterations.

3 Implementation

3.1 Class Diagrams

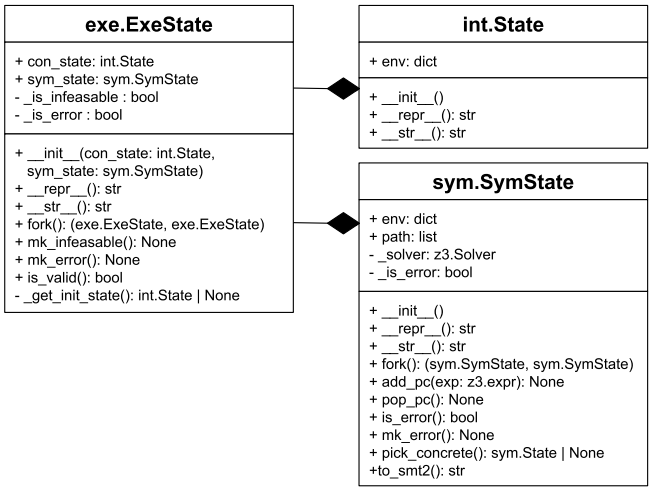

Fig. 2. ExeState Class Diagram

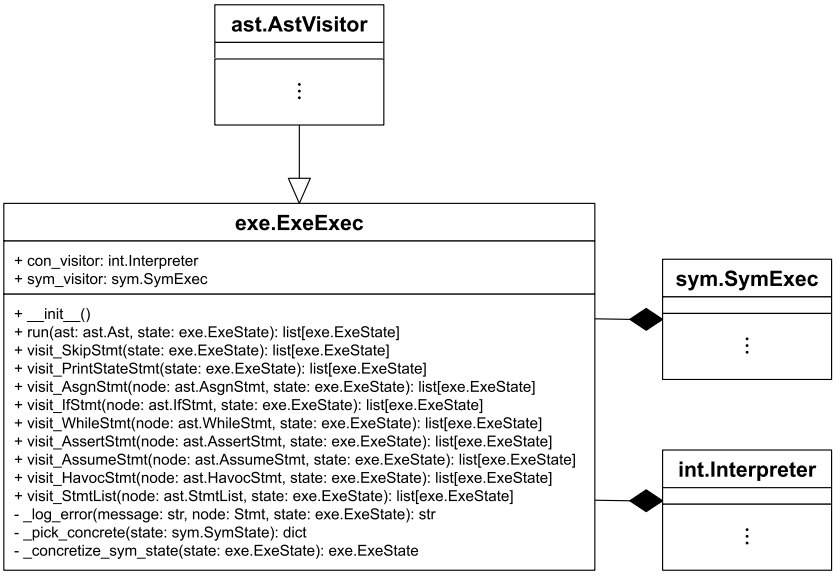

Fig. 3. ExeExec Class Diagram

3.2 Module Breakdown

ExeState: The ExeState class, detailed in figure 2, holds the current state of the execution, including both symbolic (`sym_state`) and concrete (`con_state`) states. This dual nature allows the system to handle concrete values for practical execution while also maintaining symbolic expressions to explore different execution paths.

Forking States: One of the essential features of the EXE algorithm is the ability to explore multiple paths simultaneously. The ExeState class supports this by providing the `fork()` method, which creates a copy of the current state. This new state can then diverge from the original, exploring a different path based on branch conditions. This is crucial for handling conditional statements and loops, where different paths need to be evaluated.

Error Management: The ExeState class includes methods like `mk_error()` to handle errors encountered during execution. This feature is vital for detecting potential issues in the program, such as assertion failures, and logging these errors for further analysis.

Path Condition Management: Path conditions are logical expressions that must hold true for a particular execution path to be feasible. The ExeState class manages these conditions as part of the symbolic state. When the execution reaches a conditional statement, the state can fork, with each new state adding a different constraint (representing the true or false condition of the branch) to its path conditions. States make use of the `is_empty()` method to check if the current symbolic state is satisfiable.

State Validity: The method `is_valid()` checks whether the current state is feasible, or has encountered some halting error. This function is used to prevent the execution of impossible or error-ed paths.

ExeExec: The ExeExec class serves as the core execution engine for the symbolic execution framework. It acts as an abstract syntax tree (AST) visitor, systematically traversing and executing the nodes of the WHILE program.

Visiting Statement Lists: This is done by iterating through the list of statements and invoking the corresponding visit methods for each statement type. This approach ensures that the program's flow is correctly managed and that each statement is executed in the proper sequence.

The ExeExec class was designed such that the visit methods always takes as input a single state and always returns a list of states. The unpacking and repacking of these state lists is also handled by the `visit_StmtList()` method, this helped to simplify the implementation of the other visit methods.

Visiting Assume and Assert Statements: Assume and assert statements are critical for constraining the state space and verifying program properties. The ExeExec class handles these statements by adding constraints to the path conditions.

Assume statements add assumptions to the symbolic state, effectively restricting the paths that the execution can take. The class checks whether the current state satisfies the assumption and prunes paths that do not meet these conditions or generates new concrete assignments that satisfy the assumption.

Assert statements are used to verify conditions that must hold true. The ExeExec class checks the validity of these assertions in the current state by forking execution and evaluating the satisfiability of the result state that passes the assert and the state where it fails. If an assertion can fail, it indicates a potential bug or error in the program, which is logged for further analysis. In the event that both the fail state and the pass state are both satisfiable new concrete assignments are calculated and used for the rest of the execution.

Visiting If Statements: If statements introduce branching in the program, leading to multiple possible execution paths. The ExeExec class handles this by forking the current state into two: one for the true branch and one for the false branch. The true branch adds the condition of the if statement as a constraint, while the false branch adds the negation of that condition. These constraints are managed using the Z3 solver context stored in each symbolic state to determine the feasibility of each path. Since the newly added constraint includes all the previous constraints that allowed us to reach that state the new concrete assignments can be generated as needed to satisfy the new constraints.

Visiting While Statements: While statements introduce loops, which can create complex state spaces due to repeated execution. The ExeExec class manages while loops by evaluating the loop's condition and executing the body, maintaining both symbolic and concrete states.

To manage complexity, the ExeExec class symbolically unrolls loops up to 10 iterations forking the current state into an exit loop state and a continue loop state. The exit loop state add the constraint for exiting the loop after some number of iterations and the enter loop state adds the constraint for continuing the loop. New concrete assignment are generated using the updated solver context as needed. If the loop has not concluded after these 10 symbolic iterations, it is deemed too complex to handle purely symbolically.

After reaching the unrolling limit, the symbolic state is concretized. This means taking the concrete assignment of variables after the loop finishes executing and using these concrete values for the rest of the execution by adding them as constraints to the current Z3 solver context. This approach helps to prevent the state space from becoming unmanageable and allows the analysis to proceed efficiently.

4 Evaluation

4.1 Test Results

```
havoc x;
r := 0;
if x > 8 then r := x - 7;
if x < 5 then r := x - 2
```

```
[exec]: state reached
Init Concrete State:
x: 9
Concrete State:
x: 9
r: 2
Symbolic State:
x: x!0
r: x!0 - 7
pc: [8 < x!0, Not(5 > x!0)]

[exec]: state reached
Init Concrete State:
x: 0
Concrete State:
x: 0
r: -2
Symbolic State:
x: x!0
r: x!0 - 2
pc: [Not(8 < x!0), 5 > x!0]

[exec]: state reached
Init Concrete State:
x: 5
Concrete State:
x: 5
r: 0
Symbolic State:
x: x!0
r: 0
pc: [Not(8 < x!0), Not(5 > x!0)]

[exec]: found 3 valid states
```

Fig. 4. Sample WHILE program and output from concolic execution engine

In this section, we present the results from testing our concolic execution engine on a sample WHILE program seen in figure 4, taken from our lecture notes [1]. The program involves a series of conditional statements based on the variable x, which is initially assigned an arbitrary value through the havoc operation. The program then performs a series of assignments to the variable r based on the value of x.

The engine discovered same three valid execution paths that we derived in the lecture [1]. I will now walk through how the concolic execution engine generated these paths. First, it began executing with a single empty state after executing the havoc statement the variable x was assigned symbolically to x!0 and the concretely to 0. It would then proceed to the next statement assign the variable r to the integer value 0 concretely and symbolically. Next it would arrive at the first if statement. This would cause our current state to fork into 2 states adding the condition $8 < x$ to the solver context of state 1 and adding its negation $8 >= x$ to the state 2. After these constraints are added each solver is checked for satisfiability which in this case is true since there are no other constraints on x at this time. Since the current concrete assignment for x is 0 we generate a new concrete assignment for x in state 1 using state 1's solver context. After completing the execution of the first if statement we have 2 states each with its own solver context that will be used to evaluate the satisfiability going forward. Upon reaching the next if statement first state 1 is forked into another 2 states each of theses states preserve state 1's solver context add a new constraint for this if statements condition. This creates state 1a which has $8 < x$ and $5 > x$ but after adding this new constraint this state is no longer satisfiable and therefore is dropped from the execution. The other state that was created is state 1b which has the constraints $8 < x$ and $5 <= x$ which is satisfiable. Depending on what the solver choose to be the new concrete assignment of x after the first fork 1b may have to reassign the concrete value of x before continue. The same process happens to state 2 creating state 2a with the context $8 >= x$ and $5 > x$ and state 2b with the context $8 >= x$ and $5 <= x$. It then generates new concrete assignments as needed to statisify the updated constraints.

Path 1: When $8 < x \wedge 5 <= x$: This path can be executed by setting the initial value of x to 9 and results in r equals 2.

Path 2: When $8 >= x \wedge 5 > x$: This path can be executed by setting the initial value of x to 0 and results in r equals -2.

Path 3: When $8 >= x \wedge 5 <= x$: This path can be executed by setting the initial value of x to 5 and results in r equals 0.

4.2 Test Coverage

All tests are written in `test_exe.py` and `test_sym.py`. To maximize the program's correctness, the team aimed to create a test suite using the **unittest**

framework, that covers 100 % statement and branch coverage. We thoroughly examined our symbolic engine with some edge cases to ensure comprehensive coverage. The team eventually reached 99 % coverage of branches and statements. The detailed reasons of why this line and branch can't be tested is included at the end of the section

The team first aim to test for the visiting methods within the class ExeExec. This class is the core execution engine for symbolic execution framework. It acts as an abstract syntax tree (AST) visitor, systematically traversing and executing the nodes of the WHILE program. This means the coverage of this class will cover methods for each state, meaning most class attributes from ExeState will be covered as well. Therefore, the optimal strategy is to cover all branches and states within ExeExec first, and then aim to cover all other parts.

Within the class ExeExec the team aims to cover `visit_AssertStmt`, `visit_AssumeStmt`, and `visit_HavocStmt` first. The first 3 tests in the below image covers all branches and lines for these 3 visitors.

```
class TestExe (unittest.TestCase):
    def test_two(self):
        prg1 = "havoc x; assume x > 10;assume false"
        ast1 = ast.parse_string(prg1)
        engine = exe.ExeExec()
        st = exe.ExeState()
        out = [s for s in engine.run(ast1, st)]
        self.assertEquals(len(out), 1)
    def test_one(self):
        prg1 = "havoc x; assume x>10; assert x<9; assert x>10"
        ast1 = ast.parse_string(prg1)
        engine = exe.ExeExec()
        st = exe.ExeState()
        out = [s for s in engine.run(ast1, st)]
        self.assertEquals(len(out), 1)
    def test_three(self):
        prg1 = "havoc x; assert x<9"
        ast1 = ast.parse_string(prg1)
        engine = exe.ExeExec()
        st = exe.ExeState()
        out = [s for s in engine.run(ast1, st)]
        self.assertEquals(len(out), 2)
```

Fig. 5. Tests for havoc, assert, and assume

Within the class ExeExec the team aims to cover `visit_AssertStmt`, `visit_AssumeStmt`, and `visit_HavocStmt` first. The first 3 tests in the below image cover all branches and lines for these 3 visitors.In the above image, `test_two` and `test_one` are able to cover `visit_AssumeStmt` by exploring whether we can generate a concrete assignment. The code in test 2 and 3 explores

whether both branches are SAT for the AssertStmt, and in the case of only one branch is SAT, whether the concrete condition is met. The havoc statements can be covered if we only call the visitor itself.

The team now aim to cover `visit_SkipStmt`, `visit_PrintStateStmt` and `visit_AsgnStmt`

```
def test_four(self):
    prg1 = "x := 10;print_state;skip"
    ast1 = ast.parse_string(prg1)
    engine = exe.ExeExec()
    st = exe.ExeState()
    out = [s for s in engine.run(ast1, st)]
    self.assertEquals(len(out), 1)
```

Fig. 6. Tests for skip, print, and assignment

The above tests covers those 3 methods.They can be covered by simplying executing each statement within our engine, the team creates these 3 statements in asts and execute them

To cover the `visit_IfStmt`, The team used similar statements explained in 4.1. An additional else statement is added to cover the else branch

```
def test_five(self):
    prg1 = "havoc x;r:=0;if x>8 then r:=x-7 else r:=x-9; if x<5 then r:=x-2"
    ast1 = ast.parse_string(prg1)
    engine = exe.ExeExec()
    st = exe.ExeState()
    out = [s for s in engine.run(ast1, st)]
    self.assertEquals(len(out), 3)
```

Fig. 7. Test if both branches are SAT

In the above image, the team is only able to cover branches with the both branches are SAT. To cover the scenario with only one branch is SAT. The team uses the below test.

```
def test_six(self):
    prg1 = "x:=5;r:=0;if x>8 then r:=x-7 else r:=x-9; if x<=5 then r:=x-2"
    ast1 = ast.parse_string(prg1)
    engine = exe.ExeExec()
    st = exe.ExeState()
    out = [s for s in engine.run(ast1, st)]
    self.assertEquals(len(out), 1)
```

Fig. 8. Test if only 1 branch is SAT

In the above image, by setting x to 5, the team can have one less symbolic state, by using the condition x $>$ 5 or x $<$=5.

Lastly, the team has to verify a nest if condition would yield the correct output. There are a total of 3 states from the below test image, from 3 paths

```
def test_seven(self):
    prg1 = "havoc x;if x>0 then if x>5 then a := 1 else b := 1 else c := 1"
    ast1 = ast.parse_string(prg1)
    engine = exe.ExeExec()
    st = exe.ExeState()
    out = [s for s in engine.run(ast1, st)]
    self.assertEquals(len(out), 3)
```

Fig. 9. Nested If

Path 1: When $5 < x \wedge 0 < x$: This path can be executed by setting the initial value of x to 6 and results in a equals 1.

Path 2: When $5 > x \wedge 0 < x$: This path can be executed by setting the initial value of x to 1 and results in b equals 1.

Path 3: When $0 >= x$: This path can be executed by setting the initial value of x to 0 and results in c equals 1

From the below output, we can see the output of the program includes 3 states. This verifies our test is correct

```
[exec]: found 3 valid states
(venv)
```

Fig. 10. Lines and branches that cannot be covered

To test the while statement, the team created a strategy to test the while with both branches are SAT. The team also created a strategy with only one branch is SAT. Test eight, nine, ten and eleven covers all branches and lines of `visit_WhileStmt`.

```
def test_nine(self):
    prg1 = "havoc x;while x > 2 do x:=x-1"
    ast1 = ast.parse_string(prg1)
    engine = exe.ExeExec()
    st = exe.ExeState()
    out = [s for s in engine.run(ast1, st)]
    self.assertEquals(len(out), 12)
```

Fig. 11. general while loop

```
[exec]: state reached
Init Concrete State:
x: 13

Concrete State:
x: 2

Symbolic State:
x: 2
pc: [2 < x!0, 2 < x!0 - 1, 2 < x!0 - 1 - 1, 2 < x!0 - 1 - 1 - 1, 2 < x!0 - 1 - 1 - 1 - 1, 2 < x!0 - 1 - 1 - 1 - 1 - 1, 2 < x!0 - 1 - 1 - 1 - 1 -
 1, 2 < x!0 - 1 - 1 - 1 - 1 - 1 - 1, 2 < x!0 - 1 - 1 - 1 - 1 - 1 - 1 - 1, 2 < x!0 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1, 2 < x!0 - 1 - 1 - 1 -
 1 - 1 - 1 - 1 - 1 - 1 - 1, 2 == 2]
```

Fig. 12. output of figure 11

In test 9, we demonstrate that with the use of 10 unrolling limit, we are able to achieve an output of 12 states. That is because the highest initial concretization state possible is 13, with 11 path constraints and an additional concretization constraint, both in the highest concretization constraint.

To further demonstrate the program satisfy the concretization requirement, we demonstrate that the concretization of expressions within while loops are added to the path condition. We mentioned previously that when program sets an unrolling limit of 10, after reaching the unrolling limit, the symbolic state is concretized. This means for the program above only one state remains which is the state that loops 10 times and is concretized. This is because we assumed $x > 20$ before executing the look so this is the only possible execution path. This is demonstrated in the below figure 13.

```
def test_twelve(self):
    prg1 = "havoc x;; assume x>20;while x > 10 do x:=x-1"
    ast1 = ast.parse_string(prg1)
    engine = exe.ExeExec()
    st = exe.ExeState()
    out = engine.run(ast1, st)
    self.assertEquals(len(out), 1)
```

Fig. 13. concretization of while loops when reaching the limit of 10

```
leozl@DESKTOP-8DA7IOV MINGW64 /c/ece653backup/prj-golivott-l536zhan (leo)
$ python -m wlang.exe wlang/test2.prg
[exec]: state reached
Init Concrete State:
x: 21

Concrete State:
x: 10

Symbolic State:
x: 10
pc: [20 < x!0, 10 < x!0, 10 < x!0 - 1, 10 < x!0 - 1 - 1, 10 < x!0 - 1 - 1 - 1, 10 < x!0 - 1 - 1 - 1 - 1, 10 < x!0 - 1 - 1 - 1 - 1 - 1, 10 < x!0 - 1 - 1 - 1 - 1 - 1 - 1, 10 < x!0 - 1 - 1 - 1 - 1 - 1 - 1 - 1, 10 < x!0 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1, 10 < x!0 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1, 10 < x!0 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1, 10 < x!0 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1, 10 == 10]

[exec]: found 1 valid states
```

Fig. 14. output of figure 13

In figure 14, we see that the output of our program in `test_12` included the concretization of x=10 into the path constraint. Also, there is only 1 state. This proves our engine is correct.

wlang\exe.py: 99% 229 1 0 1

```
339     invalid_states = [s for s in states if not s.is_valid()]
340
341     count = 0
342     for out in invalid_states:
343         count = count + 1
344         print('[exec]: invalid state reached')
345         print(out)
346         print('[exec]: found', count, 'invalid states\n')
347
348     if len(valid_states) == 0:
349         print('[exec]: no valid output states')
350     else:
351         count = 0
352         for out in valid_states:
353
354             count = count + 1
355             print('[exec]: state reached')
356             print(out)
357         print('[exec]: found', count, 'valid states')
358     return 0
359
360
361 if __name__ == '__main__':                                  361 ↛362
362     sys.exit(main())
```

Fig. 15. Lines and branches that cannot be covered

Eventually, the team reached 99 % coverage of branches and statements.The only line and branch that can't be covered are included in the image above. The above program can not be covered because line 361 to line 362 needs to called externally. It can not be called within the test program. The image above also shows all other lines and branches are covered.

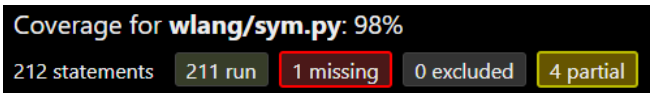

Fig. 16. Coverage for sym.py

The team also tested `sym.py` in figure 16 and achieved 98 % branch coverage. The only branches missing are either impossible to test under normal operation due to the semantics of the WHILE programming language or lines the cannot be run using the test program.

5 Conclusion

This project successfully developed a concolic execution engine in the style of EXE, tailored for the WHILE programming language and leveraging the Z3 SMT solver's incremental solving capabilities. Our engine's main contributions include the integration of symbolic and concrete execution to explore program paths comprehensively, the effective use of Z3's Assumptions interface to manage evolving constraints efficiently, and the implementation of techniques to handle complex program structures, such as loops.

Despite the progress made, the current implementation has limitations. One notable challenge is the handling of highly complex symbolic constraints, particularly in nested conditional and loop structures, which can result in state space explosion. Additionally, the lack of function support in the WHILE language restricts the system's ability to concretize complex code sections effectively.

Future work could focus on optimizing the use of the Assumptions interface to further enhance the efficiency of constraint management. Expanding support for more complex features in the WHILE language, such as functions and more sophisticated data structures, would also broaden the applicability of the engine. Integrating additional solving techniques or exploring hybrid approaches that combine concolic execution with other verification methods could improve path exploration and coverage.

Overall, the concolic execution engine developed in this project provides a valuable tool for the verification and analysis of WHILE programs, with potential applications in software testing and verification. It demonstrates the feasibility and benefits of using concolic execution combined with advanced SMT solving techniques to achieve comprehensive test coverage and identify potential software errors.

References

1. Werner Dietl: Dynamic Symbolic Execution. Slides, 8-21 (2024)
2. Programming Z3, https://theory.stanford.edu/ nikolaj/programmingz3.html#sec-incrementality, last accessed 2024/07/29