Improvement in Symbolic Execution Performance by Combining z3 Scopes and Assumptions with Incremental Solving Interfaces.

Devansh Kansara - 21112551
Harmanjot Singh - 21114621

July 30, 2024

ECE 653 - Software Testing, Quality Assurance and Maintenance
Spring 2024

1 Abstract

In this project, we selected choice 0 and implemented two important components for our existing symbolic execution engine. This project's main goal was to significantly improve symbolic execution performance, which is crucial for software system analysis. These improvements were created as an addition to the Python programming language with the goal of improving the execution engine's efficacy and efficiency. The project involved integrating the state-of-the-art Satisfiability Modulo Theories (SMT) solver z3 in a symbiotic manner with the pre-existing framework for symbolic execution.

The key to our invention lies in applying z3's assumption-based incremental solving and scope management, which results in an analysis process that is more reliable and effective.

2 Introduction

Symbolic execution is a way of executing a program abstractly, so that one abstract execution covers multiple possible inputs of the program that share a particular execution path through the code. The execution treats these inputs symbolically, “returning” a result that is expressed in terms of symbolic constants that represent those input values. By looking at the program's control flow graph, collecting path conditions, and identifying input sets that match particular execution paths, this technique abstracts away the concrete execution.

In this field, Microsoft Research's Z3 theorem prover's remarkable integration is essential. Z3 transcends its function as a mere tool for checking satisfiability; it is crucial for decision-making in symbolic execution. The prover assesses path condition feasibility, supporting the examination of potential execution paths that conform to the program’s logical parameters. With its adeptness in various theories, Z3 excels in evaluating intricate data structures and operations, essential for the nuanced demands of symbolic execution.

Our implementation depends on path conditions within the symbolic execution paradigm. At certain points in the program, they serve as logical limitations on symbolic variables. These changing conditions determine the direction of execution and whether particular paths are viable with regard to of particular inputs. The engine can carefully explore through paths and uncover behaviours that would be hidden during regular execution thanks to them, which are essential for tracking the program's progress.

To sum up, this study presents a new engine designed to analyse Python code through symbolic execution, utilizing the Z3 Satisfiability Modulo Theories (SMT) solver to improve state analysis. Z3 is strategically used to improve the engine's capacity to traverse program states under various assumptions, which significantly improves the engine's ability to find bugs and security problems.

3 Design and Implementation

3.1 Scope

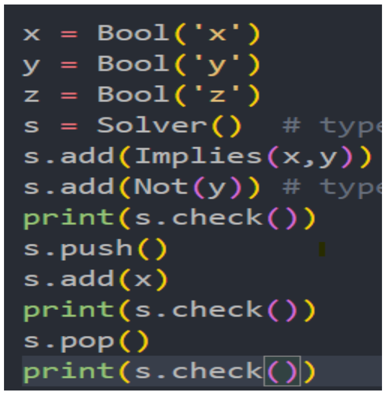
Scopes in z3 are pivotal for managing the solver's state in a dynamic and flexible manner. We implemented a stack-like structure for scope management, allowing constraints to be added, modified, or removed. This modular approach to problem-solving enables the solver to handle constraints in an isolated manner, within their respective scopes.

3.1.1 Implementation of Scope Incremental Interface

Z3 uses a stack-based structure for scope management. A fresh frame, or scope, is added to the top of the stack with the "Push" command. Every frame contains a distinct scope that preserves the solver's state as well as its own set of constraints. After a "Push," any constraints added to the solver stay inside the current top scope so as not to affect the scopes before it. This makes scopes loosely connected.

Additionally, scope management has a "Backtracking" function that allows you to go back in time. One important tool in this backtracking procedure is the "Pop" function. It removes the top frame from the stack and with it all the limitations connected to that specific scope. By doing this, the solver can get back to how it was before the related push. When branching logic is included, as in conditional if-else statements or while loops, "Pop" is useful because it allows the solver to explore alternative branches without keeping the constraints from earlier routes.

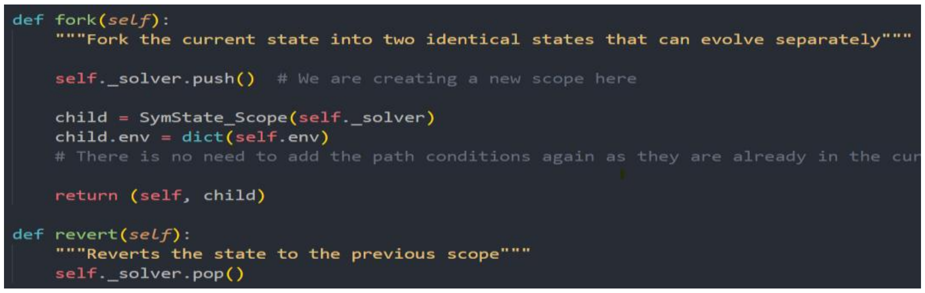
According to the above explanation, two states would have been created. However, by using this backtracking approach, we can easily and efficiently go back to a



```
x = Bool('x')
y = Bool('y')
z = Bool('z')
s = Solver()  # type
s.add(Implies(x,y))
s.add(Not(y)) # type
print(s.check())
s.push()
s.add(x)
print(s.check())
s.pop()
print(s.check())
```

Figure 1(a): Illustration of the scope stack with backtracking using Pop operation.

previous state. A path is popped off the stack, freeing up memory and requiring less space, once it has been thoroughly explored. In the previously given example, we add the constraint x and then verify satisfiability. The solver returns to its initial state when we pop that branch off the stack once we are done working with it.

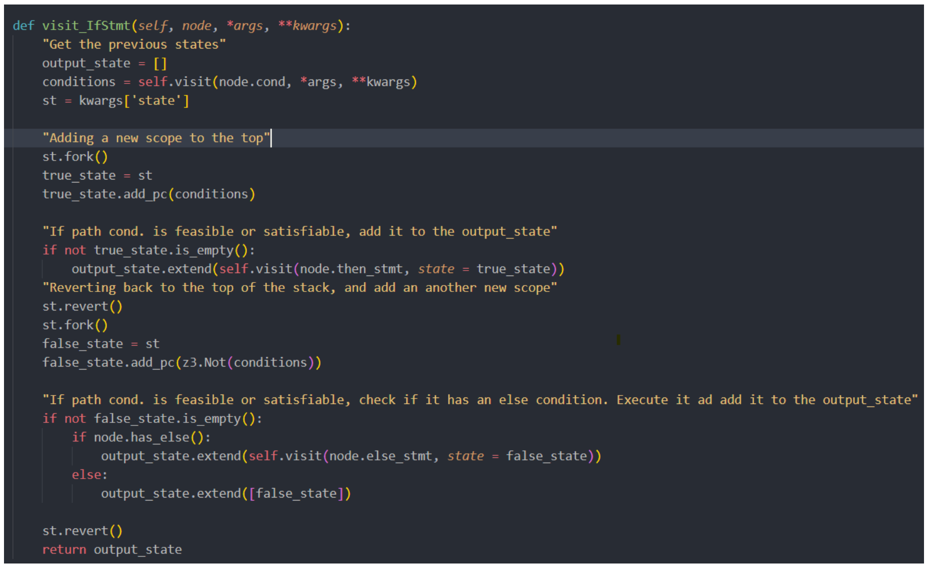


```python
def fork(self):
    """Fork the current state into two identical states that can evolve separately"""

    self._solver.push()  # We are creating a new scope here

    child = SymState_Scope(self._solver)
    child.env = dict(self.env)
    # There is no need to add the path conditions again as they are already in the cur

    return (self, child)

def revert(self):
    """Reverts the state to the previous scope"""
    self._solver.pop()
```

Figure 1(b): Illustration of Fork and Revert Method.

The fork() method simply pushes a new scope to the top of the stack, while the revert() method, returns the stack to the previous state by popping the added element to ensure it rolls back to the previous state, this step is very necessary as the path condition is loosely coupled from the Z3 symbolics.

3.1.2 IF Condition using Scopes(visit_IfStmt)



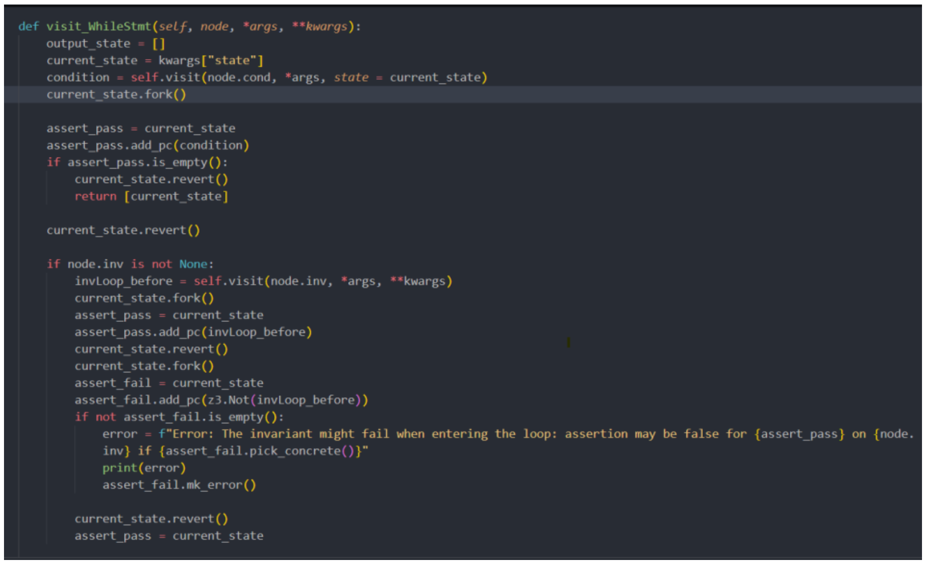
```python
def visit_IfStmt(self, node, *args, **kwargs):
    "Get the previous states"
    output_state = []
    conditions = self.visit(node.cond, *args, **kwargs)
    st = kwargs['state']

    "Adding a new scope to the top"
    st.fork()
    true_state = st
    true_state.add_pc(conditions)

    "If path cond. is feasible or satisfiable, add it to the output_state"
    if not true_state.is_empty():
        output_state.extend(self.visit(node.then_stmt, state = true_state))
    "Reverting back to the top of the stack, and add an another new scope"
    st.revert()
    st.fork()
    false_state = st
    false_state.add_pc(z3.Not(conditions))

    "If path cond. is feasible or satisfiable, check if it has an else condition. Execute it ad add it to the output_state"
    if not false_state.is_empty():
        if node.has_else():
            output_state.extend(self.visit(node.else_stmt, state = false_state))
        else:
            output_state.extend([false_state])

    st.revert()
    return output_state
```

Figure 2: Illustration of If statement.

Using scopes is essential to efficiently managing states in the "visit_IfStmt" function with the use of Z3's push and pop functionalities. In order to allow for the examination of the "true" branch without altering the initial state, the process first evaluates the condition of the if statement. This is followed by a state fork that adds a new scope. After processing the "true" branch, the state is rolled back to the previous scope and then processed again for the "false" branch. This approach ensures a productive and independent analysis of every branch, making use of Z3's scope management features to regulate and reverse states as needed.

It is crucial to remember that one should always include a replica of a state rather than the original when adding it to the output list. This is due to the fact that a large number of programming languages have inbuilt pass-by-reference semantics, and some may not be able to handle reference counting during heap management.

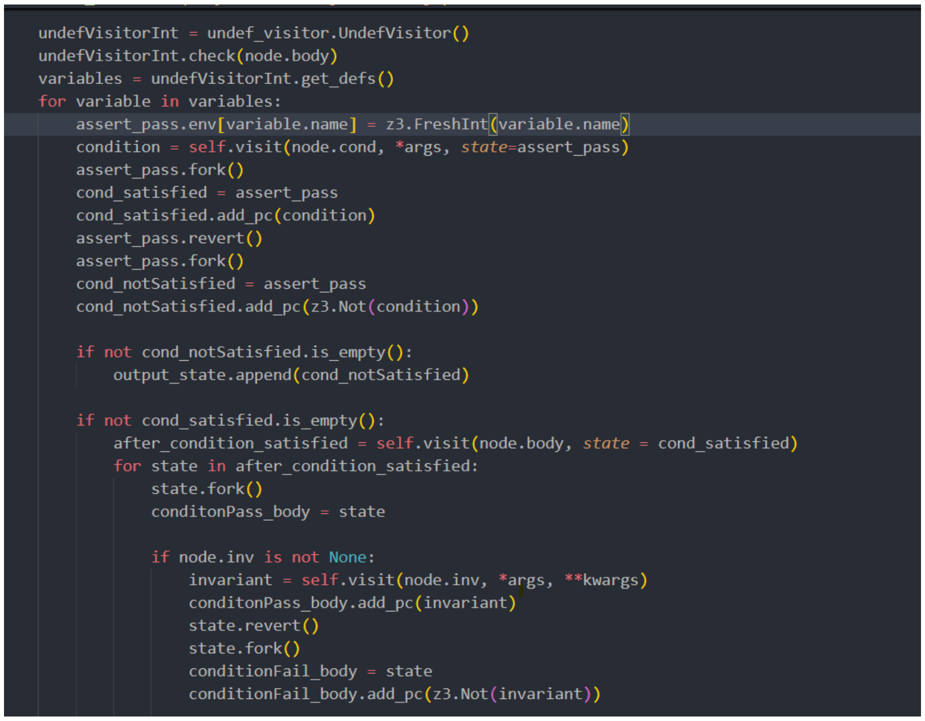
3.1.3 While Loop using Scopes (visit_WhileStmt)



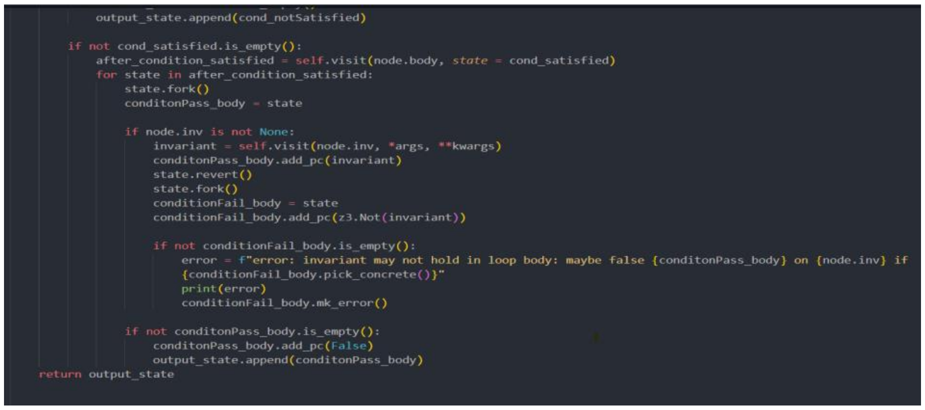
```python
def visit_WhileStmt(self, node, *args, **kwargs):
    output_state = []
    current_state = kwargs["state"]
    condition = self.visit(node.cond, *args, state = current_state)
    current_state.fork()

    assert_pass = current_state
    assert_pass.add_pc(condition)
    if assert_pass.is_empty():
        current_state.revert()
        return [current_state]

    current_state.revert()

    if node.inv is not None:
        invLoop_before = self.visit(node.inv, *args, **kwargs)
        current_state.fork()
        assert_pass = current_state
        assert_pass.add_pc(invLoop_before)
        current_state.revert()
        current_state.fork()
        assert_fail = current_state
        assert_fail.add_pc(z3.Not(invLoop_before))
        if not assert_fail.is_empty():
            error = f"Error: The invariant might fail when entering the loop: assertion may be false for {assert_pass} on {node.
            inv} if {assert_fail.pick_concrete()}"
            print(error)
            assert_fail.mk_error()

        current_state.revert()
        assert_pass = current_state
```

Figure 3(a): While at Initiation statement.

Fork the state, add the path condition that the invariant fails, and determine whether the path is feasible. If it is, we print that the invariant might fail under the state. This process ensures that, if the while loop has an invariant, it is true at the initiation stage (before the loop runs). Then, if it is possible, we add the path requirement that the invariant passes to our output state, reverse the state, fork the state once more, and then reverse the state.



```python
undefVisitorInt = undef_visitor.UndefVisitor()
undefVisitorInt.check(node.body)
variables = undefVisitorInt.get_defs()
for variable in variables:
    assert_pass.env[variable.name] = z3.FreshInt(variable.name)
    condition = self.visit(node.cond, *args, state=assert_pass)
    assert_pass.fork()
    cond_satisfied = assert_pass
    cond_satisfied.add_pc(condition)
    assert_pass.revert()
    assert_pass.fork()
    cond_notSatisfied = assert_pass
    cond_notSatisfied.add_pc(z3.Not(condition))

    if not cond_notSatisfied.is_empty():
        output_state.append(cond_notSatisfied)

    if not cond_satisfied.is_empty():
        after_condition_satisfied = self.visit(node.body, state = cond_satisfied)
        for state in after_condition_satisfied:
            state.fork()
            conditonPass_body = state

            if node.inv is not None:
                invariant = self.visit(node.inv, *args, **kwargs)
                conditonPass_body.add_pc(invariant)
                state.revert()
                state.fork()
                conditionFail_body = state
                conditionFail_body.add_pc(z3.Not(invariant))
```



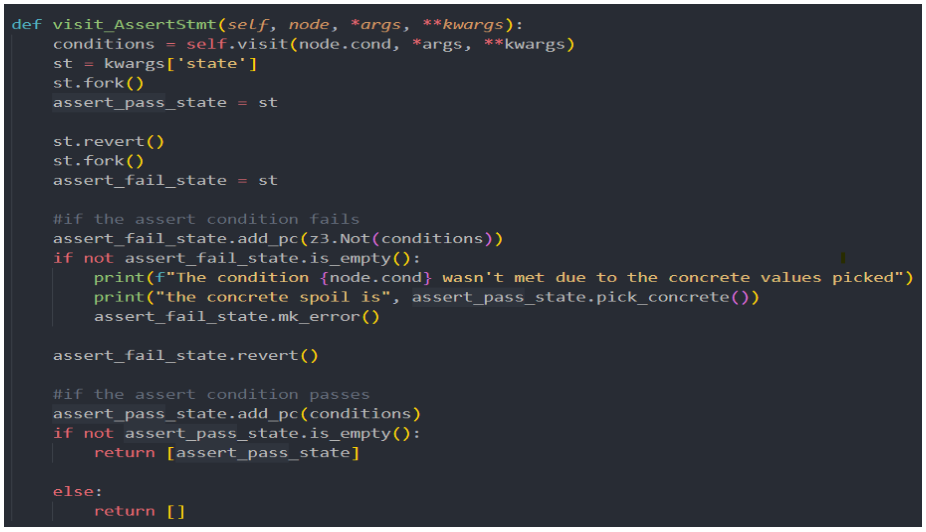
```python
        output_state.append(cond_notSatisfied)

    if not cond_satisfied.is_empty():
        after_condition_satisfied = self.visit(node.body, state = cond_satisfied)
        for state in after_condition_satisfied:
            state.fork()
            conditonPass_body = state

            if node.inv is not None:
                invariant = self.visit(node.inv, *args, **kwargs)
                conditonPass_body.add_pc(invariant)
                state.revert()
                state.fork()
                conditionFail_body = state
                conditionFail_body.add_pc(z3.Not(invariant))

                if not conditionFail_body.is_empty():
                    error = f"error: invariant may not hold in loop body: maybe false {conditonPass_body} on {node.inv} if
                    {conditionFail_body.pick_concrete()}"
                    print(error)
                    conditionFail_body.mk_error()

            if not conditonPass_body.is_empty():
                conditonPass_body.add_pc(False)
                output_state.append(conditonPass_body)
return output_state
```

Figure 3(b): While at Maintenance statement.

The invariant may be false throughout execution, but it must return to true after the body executes. This is the maintenance stage, where we must now make sure the invariant stays true after each loop. To add the path condition of the condition failing, fork the state, verify if it is viable, and then add the state to our output state in the code above.

Following this, we go back, fork once more, and add the condition passing. If this is feasible, we then execute the body and obtain a new state. At this point, we need to make sure that the new states have the invariant set to true, so we go back to the initiation stage and repeat the previous procedure, adding the invariant pass and fail condition, verifying its feasibility, and appending it to the output state once it is completed.

3.1.4 Assertion using Scopes (visit_AssertStmt)



```python
def visit_AssertStmt(self, node, *args, **kwargs):
    conditions = self.visit(node.cond, *args, **kwargs)
    st = kwargs['state']
    st.fork()
    assert_pass_state = st

    st.revert()
    st.fork()
    assert_fail_state = st

    #if the assert condition fails
    assert_fail_state.add_pc(z3.Not(conditions))
    if not assert_fail_state.is_empty():
        print(f"The condition {node.cond} wasn't met due to the concrete values picked")
        print("the concrete spoil is", assert_pass_state.pick_concrete())
        assert_fail_state.mk_error()

    assert_fail_state.revert()

    #if the assert condition passes
    assert_pass_state.add_pc(conditions)
    if not assert_pass_state.is_empty():
        return [assert_pass_state]

    else:
        return []
```

Figure 4: Assert statement using scopes.

The assert statement works in a methodical manner, retrieving the previous state first and using forking to recreate it. The path condition for a failed assertion is introduced once the assertion has branched. If it is decided that this scenario is possible, error messages appear.

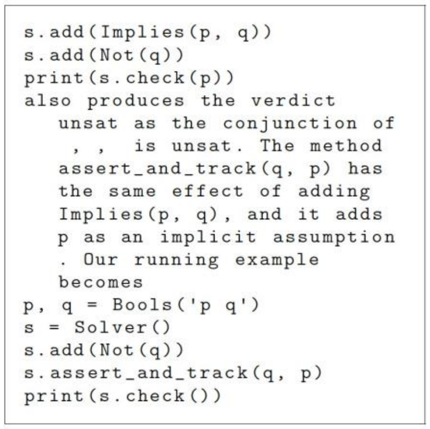
Also, we are not using the original “st” variable for checking the path condition. Instead, we are storing one condition in “assert_pass_state”, then we are reverting back in the stack which will pop off the top of the stack and again we are forking the “st” which gives us the “assert_fail_state”. Conversely, should the assertion’s success be infeasible, an empty list is returned to signify this outcome.

3.2 Assumption

The fork() method is called with an assumption or a constraint as its parameter when Assumptions are included in the code, rather than just copying the existing state. By doing this, the solver is reset to its initial state and all path conditions are applied again together with the new assumption. This process creates a new path in the symbolic execution by supposing that this specific condition is true.

A new symbolic state is produced by each conditional branch that this procedure spawns. These states retain the current path circumstances and surrounding settings, but they are additionally incorporated with new assumptions that correspond with the conditions of their individual branches.

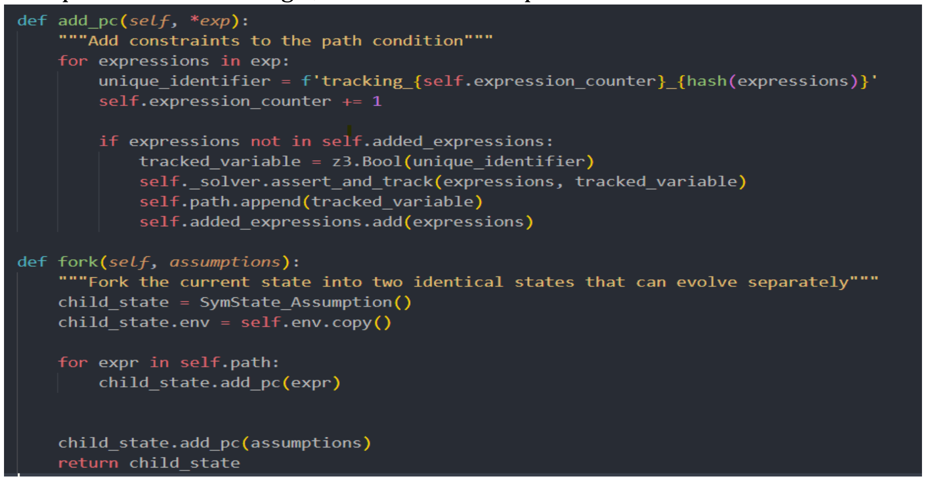
3.2.1 Implementation of Scope Incremental Interface



```
s.add(Implies(p, q))
s.add(Not(q))
print(s.check(p))
also produces the verdict
   unsat as the conjunction of
    , ,  is unsat. The method
   assert_and_track(q, p) has
   the same effect of adding
   Implies(p, q), and it adds
   p as an implicit assumption
   . Our running example
   becomes
p, q = Bools('p q')
s = Solver()
s.add(Not(q))
s.assert_and_track(q, p)
print(s.check())
```

Figure 5(a): How Assumption works.

To replicate the above logic, I have created helper functions as shown below:

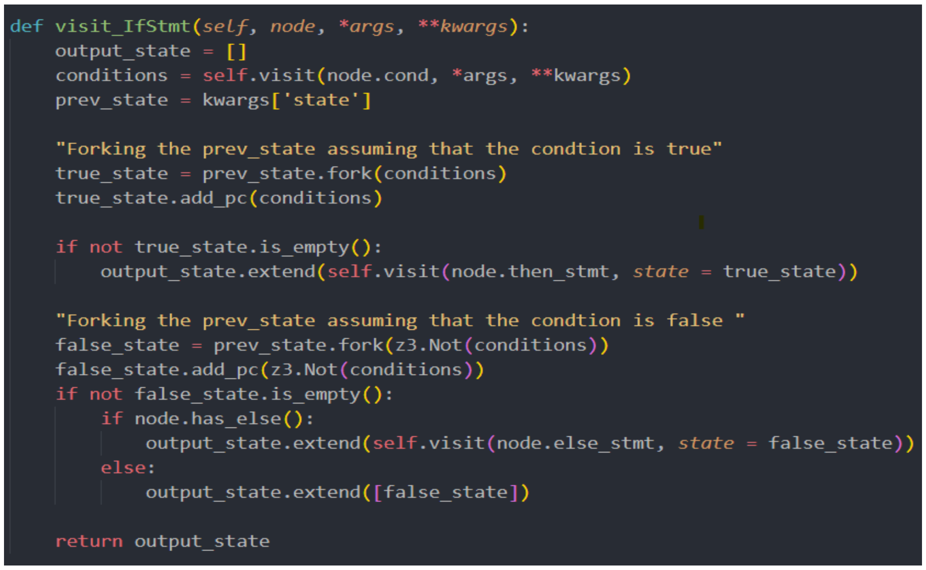


```python
def add_pc(self, *exp):
    """Add constraints to the path condition"""
    for expressions in exp:
        unique_identifier = f'tracking_{self.expression_counter}_{hash(expressions)}'
        self.expression_counter += 1

        if expressions not in self.added_expressions:
            tracked_variable = z3.Bool(unique_identifier)
            self._solver.assert_and_track(expressions, tracked_variable)
            self.path.append(tracked_variable)
            self.added_expressions.add(expressions)

def fork(self, assumptions):
    """Fork the current state into two identical states that can evolve separately"""
    child_state = SymState_Assumption()
    child_state.env = self.env.copy()

    for expr in self.path:
        child_state.add_pc(expr)


    child_state.add_pc(assumptions)
    return child_state
```

Figure 5(b): Replication in Python.

The assert_and_rack() method is used by the add_pc() function to define a constraint and associate it with a Boolean variable for tracking. The solver can identify the precise restrictions that result in unsatisfiability thanks to this relationship. The execution continues along the corresponding path when a set of constraints is found to be satisfiable. On the other hand, the path is deemed unfeasible and is no longer pursued by the engine if the constraints make it unsatisfactory. The distinct identification of the tracking variable allows one to follow back which specific assumption gave rise to the unsatisfactory circumstance.

3.2.2 IF Condition using Assumptions(visit_IfStmt)



```
def visit_IfStmt(self, node, *args, **kwargs):
    output_state = []
    conditions = self.visit(node.cond, *args, **kwargs)
    prev_state = kwargs['state']

    "Forking the prev_state assuming that the condtion is true"
    true_state = prev_state.fork(conditions)
    true_state.add_pc(conditions)

    if not true_state.is_empty():
        output_state.extend(self.visit(node.then_stmt, state = true_state))

    "Forking the prev_state assuming that the condtion is false "
    false_state = prev_state.fork(z3.Not(conditions))
    false_state.add_pc(z3.Not(conditions))
    if not false_state.is_empty():
        if node.has_else():
            output_state.extend(self.visit(node.else_stmt, state = false_state))
        else:
            output_state.extend([false_state])

    return output_state
```

Figure 6: If Stmt with Assumptions.

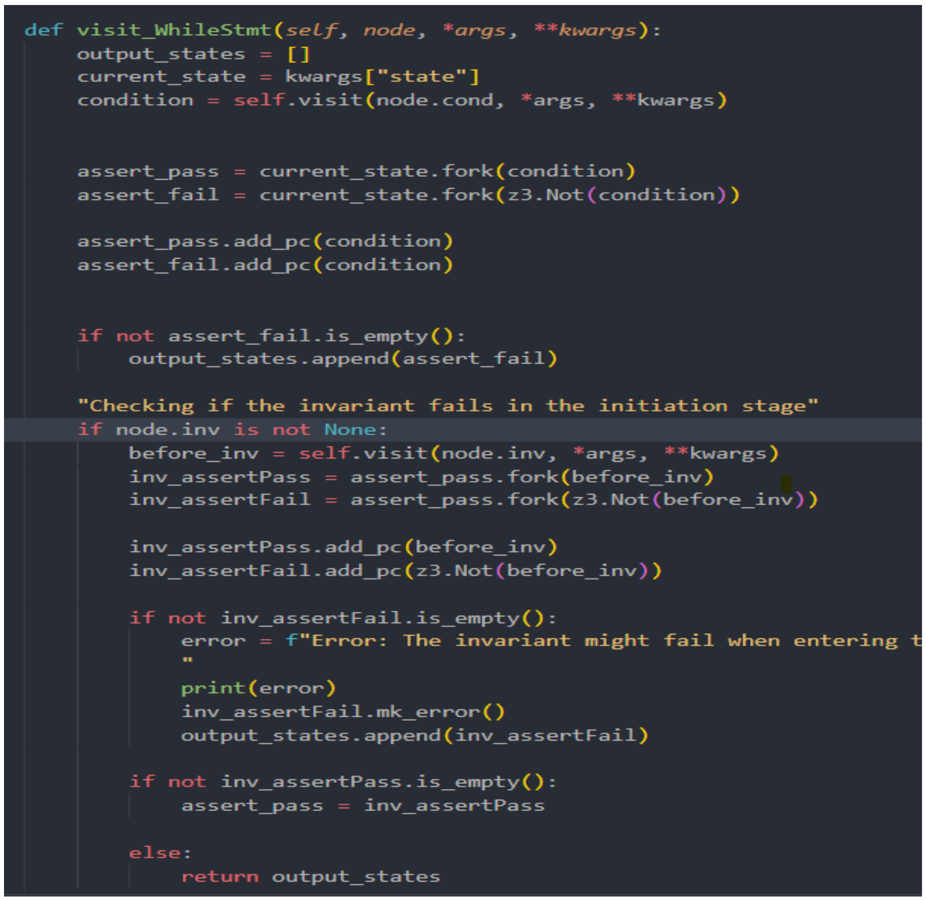
The visit_IfStmt function initiates the branching process by examining the condition of the statement in conjunction with the current symbolic state upon detecting a "if" statement in the code. Two distinct symbolic states that branch off from the present one are created by invoking the fork() function. Assuming that the condition in the "if" statement is true, the state indicated as true_state moves forward. The false_state state, on the other hand, functions on the grounds that the condition is false. This bifurcation represents the possible split in execution paths that might happen according on how the "if" condition is evaluated. In order to precisely track these varying routes, the add_pc() method is utilized.

It leverages Z3's assert and track feature to append the given conditions to the path conditions of each forked state while also assigning unique identifiers for tracking purposes.

The procedure determines whether the newly formed states are feasible after forking and condition assertion. For any state considered possible, it then carries out the associated branch of the "if" expression. The function explores the 'then' clause of the 'if' statement if true_state is true. Likewise, this alternative branch is investigated in the event that the false_state is true and a "else" clause is present.

The results of these investigations are combined into a coherent set of new states in the last phase, which shows the possible outcomes of the program after the 'if' statement. This thorough analysis of every possible execution path that arises from conditional branches is essential to the main goals of symbolic execution, which are to thoroughly investigate and examine every possible path a program could take in light of its underlying logic and conditional statements.

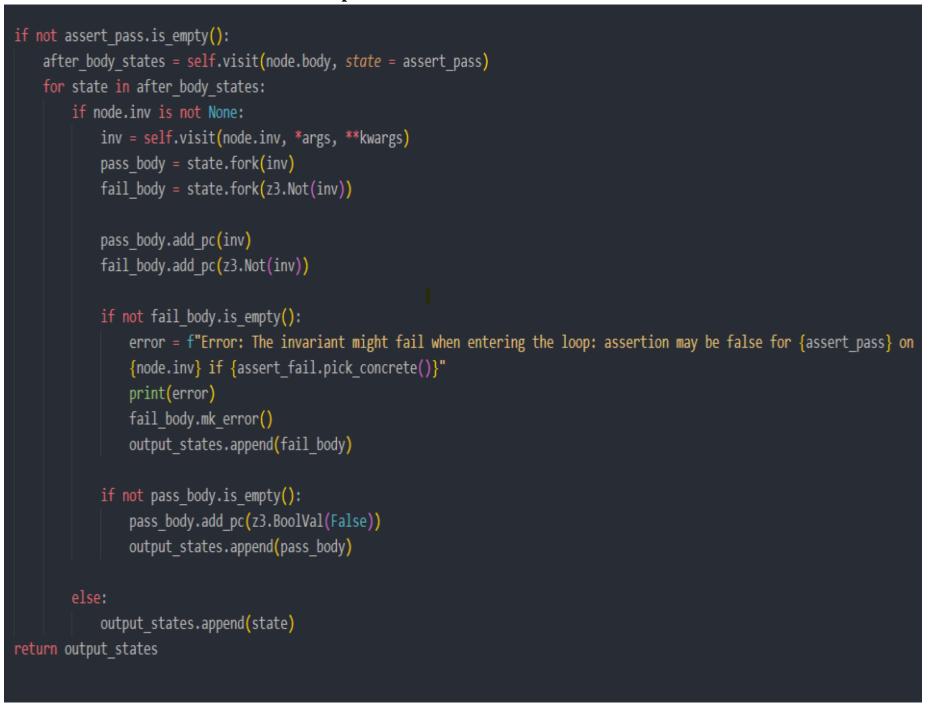
3.2.3 While Loop using Assumptions (visit_WhileStmt):



```
def visit_WhileStmt(self, node, *args, **kwargs):
    output_states = []
    current_state = kwargs["state"]
    condition = self.visit(node.cond, *args, **kwargs)


    assert_pass = current_state.fork(condition)
    assert_fail = current_state.fork(z3.Not(condition))

    assert_pass.add_pc(condition)
    assert_fail.add_pc(condition)


    if not assert_fail.is_empty():
        output_states.append(assert_fail)

    "Checking if the invariant fails in the initiation stage"
    if node.inv is not None:
        before_inv = self.visit(node.inv, *args, **kwargs)
        inv_assertPass = assert_pass.fork(before_inv)
        inv_assertFail = assert_pass.fork(z3.Not(before_inv))

        inv_assertPass.add_pc(before_inv)
        inv_assertFail.add_pc(z3.Not(before_inv))

        if not inv_assertFail.is_empty():
            error = f"Error: The invariant might fail when entering t
            "
            print(error)
            inv_assertFail.mk_error()
            output_states.append(inv_assertFail)

        if not inv_assertPass.is_empty():
            assert_pass = inv_assertPass

        else:
            return output_states
```

Figure 7(a): While Stmt at Initiation with Assumptions.

Similarly, as we did with scopes, we need to make sure the invariant holds true if there is a loop invariant supplied in the program. We produce two distinct symbolic states, one where the invariant holds true, and the other where it is false, we then add those conditions with the fork() method to ensure accurate path tracking. The next step is checking for feasibility by using the z3 solver, if any path is feasible, we add it to the output state.



```
if not assert_pass.is_empty():
    after_body_states = self.visit(node.body, state = assert_pass)
    for state in after_body_states:
        if node.inv is not None:
            inv = self.visit(node.inv, *args, **kwargs)
            pass_body = state.fork(inv)
            fail_body = state.fork(z3.Not(inv))

            pass_body.add_pc(inv)
            fail_body.add_pc(z3.Not(inv))

            if not fail_body.is_empty():
                error = f"Error: The invariant might fail when entering the loop: assertion may be false for {assert_pass} on
                {node.inv} if {assert_fail.pick_concrete()}"
                print(error)
                fail_body.mk_error()
                output_states.append(fail_body)

            if not pass_body.is_empty():
                pass_body.add_pc(z3.BoolVal(False))
                output_states.append(pass_body)

        else:
            output_states.append(state)
return output_states
```

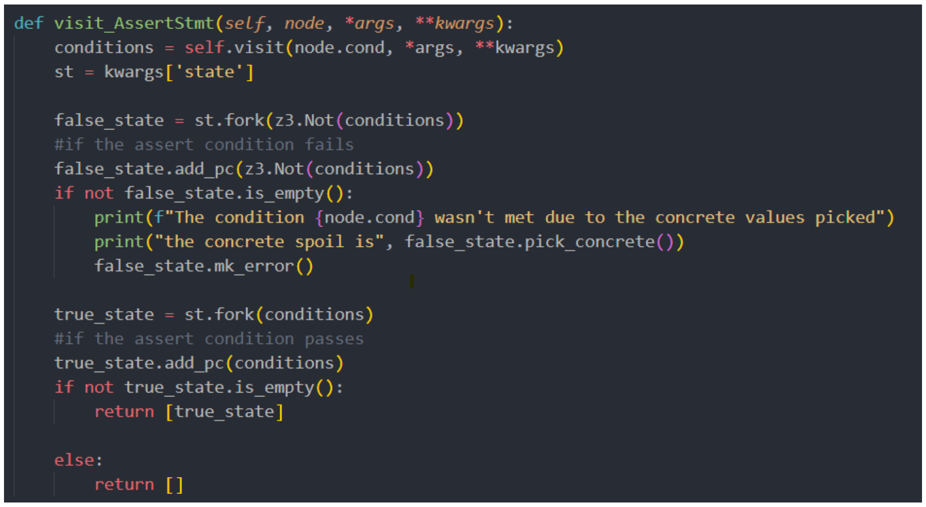
Figure 7(b): While Stmt at Maintenance with Assumptions.

Now in the maintenance stage, the visit_WhileStmt method starts by evaluating the loop's condition, then branches the current symbolic state into two separate paths. The first path, named pass_body, is predicated on the loop condition being true, suggesting that the loop will proceed. The second path, termed fail_body, is based on the assumption that the loop condition is false, indicative of the loop's termination. After establishing these states, the pass_body state incorporates the loop's condition as its path condition to indicate the loop's progression. On the other hand, fail_body adopts the inverse of the loop condition as its path condition, representing the path of loop exit. This differentiation is key for maintaining an accurate representation of the possible execution paths. Should the loop include an invariant, the method evaluates this invariant at both the start and during the execution of the loop. At each iteration of the loop, it is being checked if the invariant holds. This crucial step verifies the consistency of the invariant at the loop's inception and with each subsequent iteration.

The method checks the invariant right at the loop's entrance point. If the fail_body is not empty, this means that the invariant did not hold and so the method logs an error message in the console and adds that state to the output_ states. Then, the method proceeds to process the next iteration and reevaluates the invariant after the loop body has been executed. If the invariant is still valid after running the loop body, the loop's condition is reviewed once more, preparing for the possibility of another iteration of the loop. However, if the invariant is compromised at this point, the method logs an error, signaling a potential defect in either the loop's logic or the formulation of the invariant itself.

The various states that result from each pass through the loop, which represent the different scenarios after the execution of the loop body, are gathered together into a collection named 'output_states'. This collective array captures all the possible outcomes, encompassing the continuation of the loop in 'inv_assertPass ', exiting the loop in 'inv_assertFail ', or errors arising from invariant breaches in 'fail_body '.

3.2.4 Assertion using Scopes (visit_AssertStmt)



```python
def visit_AssertStmt(self, node, *args, **kwargs):
    conditions = self.visit(node.cond, *args, **kwargs)
    st = kwargs['state']

    false_state = st.fork(z3.Not(conditions))
    #if the assert condition fails
    false_state.add_pc(z3.Not(conditions))
    if not false_state.is_empty():
        print(f"The condition {node.cond} wasn't met due to the concrete values picked")
        print("the concrete spoil is", false_state.pick_concrete())
        false_state.mk_error()

    true_state = st.fork(conditions)
    #if the assert condition passes
    true_state.add_pc(conditions)
    if not true_state.is_empty():
        return [true_state]

    else:
        return []
```

Figure 8: Assert Stmt with Assumptions.

Within the "visit_AssertStmt" function, the method evaluates the assertion's condition by recursively translating the condition's expression into its symbolic form. This is accomplished through the 'self.visit(node.cond, *args, **kwargs)' invocation. Following this, the method splits the execution into two distinct pathways by creating branched states from the original state. These branched states, termed “false_state” and “true_state”, correspond to the outcomes of the assertion being either false or true, respectively. The ‘fork‘ function aids this branching, employing both the assertion condition and its inverse to create the diverged states.

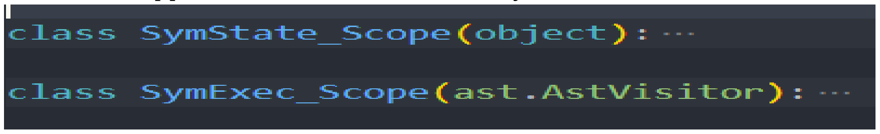
As the path condition for the "false_state" pathway, which indicates a situation in which the assertion may not hold, the negated assertion condition is integrated. Should this state turn out to be non-empty, this suggests that the assertion might fail inside the program, which would cause an error message and designate the state as incorrect. This procedure is necessary to find situations in which the claims made by the program might not be true. The assertion condition, however, is incorporated as the path condition in the "true_state" pathway. A non-empty "true_state" state permits more investigation via symbolic execution since it suggests that the statement might actually be true and does not contradict the logic of the program as a whole.

A list of these branched states is put together and returned at the end of the procedure. The array's "true_state" entry indicates that there may be execution paths where the assertion can be true. On the other hand, in the event if the assertion's validation produces a contradiction and an empty "true_state" state, the method will return an empty array, indicating that the program is unable to satisfy the assertion in any scenario.

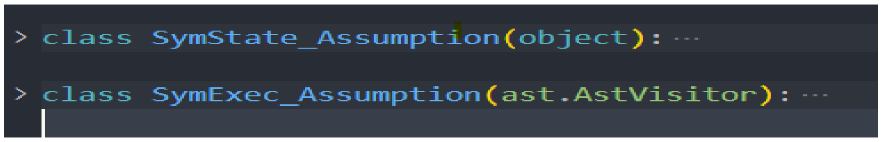
3.3 Running the code

To simplify things, we have the four classes SymState_Scope(), SymExec_Scope(), SymState_Assumption() and SymExec_Assumption() in the files sym_scope.py and sym_assumption.py respectively, and each class can be triggered by using test cases given for both the files.

Below is a snippet of what each class currently looks like



```
class SymState_Scope(object): ···

class SymExec_Scope(ast.AstVisitor): ···
```



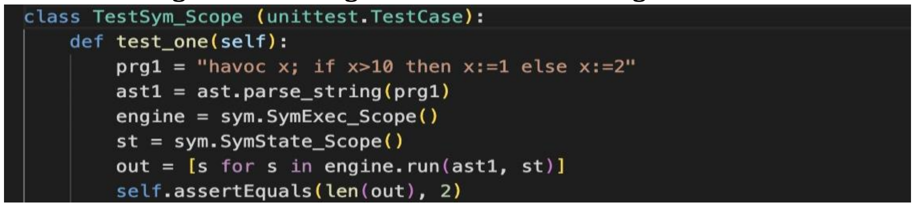
```
> class SymState_Assumption(object): ···

> class SymExec_Assumption(ast.AstVisitor): ···
```

Figure 9: Four Classes corresponding to Scopes and Assumptions.

Each of these classes have the same method names as the SymState() and SymExec(), they just add some new functions like copy(), and change the function definition of others like fork(), add pc() and others.

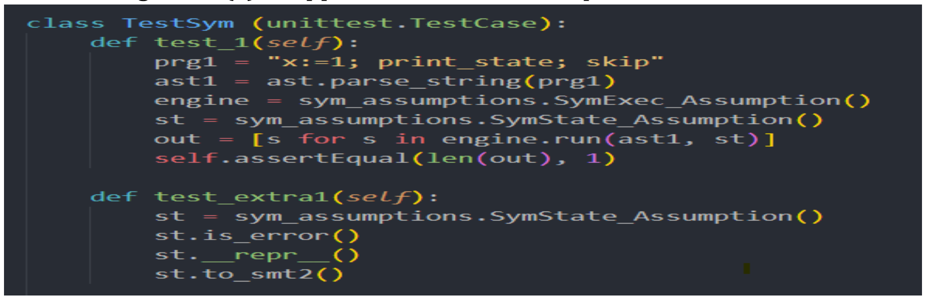
We created test suites to run both sym_scope and sym_assumptions. The tests can be run with **python -m wlang.test**

Node coverage can be done with **coverage run -m wlang.test**
Branch coverage with **coverage run –branch -m wlang.test**



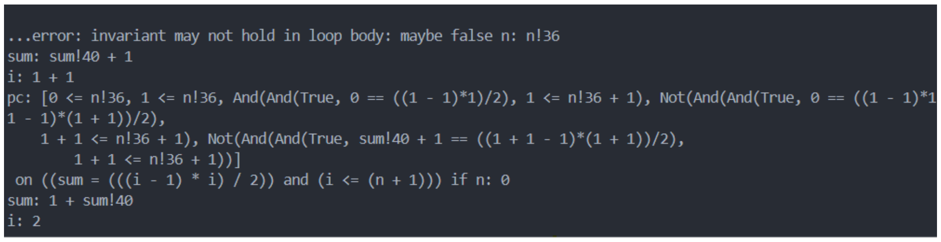
```
class TestSym_Scope (unittest.TestCase):
    def test_one(self):
        prg1 = "havoc x; if x>10 then x:=1 else x:=2"
        ast1 = ast.parse_string(prg1)
        engine = sym.SymExec_Scope()
        st = sym.SymState_Scope()
        out = [s for s in engine.run(ast1, st)]
        self.assertEquals(len(out), 2)
```

Figure 17(a): Snippet of Test class for Scope under Test Suite



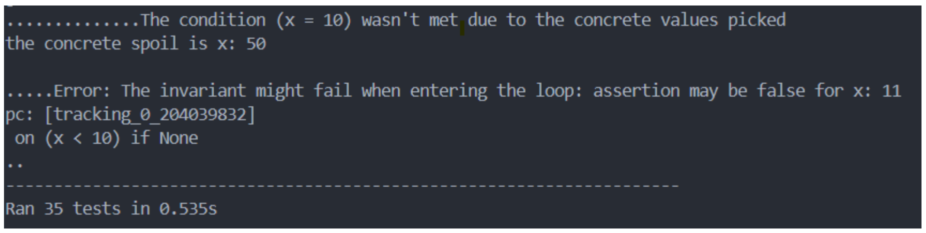
```
class TestSym (unittest.TestCase):
    def test_1(self):
        prg1 = "x:=1; print_state; skip"
        ast1 = ast.parse_string(prg1)
        engine = sym_assumptions.SymExec_Assumption()
        st = sym_assumptions.SymState_Assumption()
        out = [s for s in engine.run(ast1, st)]
        self.assertEqual(len(out), 1)

    def test_extra1(self):
        st = sym_assumptions.SymState_Assumption()
        st.is_error()
        st.__repr__()
        st.to_smt2()
```

Figure 18(a): Snippet of Test class for Assumption under Test Suite



```
...error: invariant may not hold in loop body: maybe false n: n!36
sum: sum!40 + 1
i: 1 + 1
pc: [0 <= n!36, 1 <= n!36, And(And(True, 0 == ((1 - 1)*1)/2), 1 <= n!36 + 1), Not(And(And(True, 0 == ((1 - 1)*1
1 - 1)*(1 + 1))/2),
    1 + 1 <= n!36 + 1), Not(And(And(True, sum!40 + 1 == ((1 + 1 - 1)*(1 + 1))/2),
        1 + 1 <= n!36 + 1))]
 on ((sum = (((i - 1) * i) / 2)) and (i <= (n + 1))) if n: 0
sum: 1 + sum!40
i: 2
```

Figure 17(b): Snippet of the output while running test cases for sym_scope



```
.............The condition (x = 10) wasn't met due to the concrete values picked
the concrete spoil is x: 50

.....Error: The invariant might fail when entering the loop: assertion may be false for x: 11
pc: [tracking_0_204039832]
 on (x < 10) if None
..
----------------------------------------------------------------------
Ran 35 tests in 0.535s
```

Figure 18(b): Snippet of the output while running test cases for sym_assumptions

4 Conclusion

When handling complex program states, the method of symbolic execution has changed with the advent of scope management and incremental solving based on assumptions. These developments provide a strong foundation for reversing and traveling various execution routes, allowing for a more efficient management of symbolic states. These improvements significantly increase the effectiveness of software system analysis. Consequently, there has been a reduction in computational needs, which has enabled faster and more accurate evaluations.

The testing techniques incorporated into this framework provide as additional evidence for the improvements in dependability and performance ascribed to these innovations. Extensive testing methodologies customized for breadth and assumed features have assured broad coverage and dependability in addition to confirming the modified engine's resilience.

In summary, Z3 Scopes and incremental solution through assumptions have significantly improved symbolic execution's ability to handle complicated code and state situations. This development opens up new avenues for the analysis and understanding of complex software systems, paving the way for more efficient and effective methods of software testing and quality control.