Two Features for Symbolic Execution Engine

Zhijun Wei and Mustafa Khan

University of Waterloo, Waterloo, Ontario, Canada
`z29wei@uwaterloo.ca`
`mm5khan@uwaterloo.ca`

Abstract. This report presents the enhancement of the `WHILE` language's symbolic execution engine with two significant features: Incremental Solving Mode and Concolic Execution in EXE-Style. Each feature is explored in terms of design decisions, theoretical foundations, and testing strategies, underlining their impact on the engine's efficiency and capability.
The first section focuses on the Incremental Solving Mode, utilizing `push` and `pop` operations in Z3 to dynamically manage the solver's state. This feature optimizes the handling of control structures, significantly reducing computational overhead and increasing execution speed. Testing confirms a threefold improvement in processing while-loops, maintaining functional integrity across various scenarios.
The second section details the integration of Concolic Execution in EXE-Style, blending concrete and symbolic execution to enhance path exploration, especially in complex environments. This hybrid approach improves path coverage and efficiency, as demonstrated in tests simulating real-world applications.
Collectively, these features mark substantial progress in program analysis tools for `wlang`, offering improved performance and expanded applicability in diverse programming challenges. The report showcases the enhancements not only in theoretical formulation but also in practical execution and testing, affirming the engine's advanced capabilities in handling intricate program constructs.

Keywords: Symbolic execution · Incremental solving mode · Concolic execution

1 Incremental Solving Mode of Z3

1.1 Design Decisions

Motivation In designing our symbolic execution engine, a key decision was the integration of Z3's incremental solving mode. This decision was driven by the need to optimize the performance of our engine, particularly in handling the numerous similar queries generated during symbolic execution.
Symbolic execution inherently involves exploring various program paths, each leading to potentially similar but slightly different queries to the SMT solver. Traditional approaches treat each query independently, often leading to redundant computations. By leveraging the incremental solving capabilities of Z3, we

aimed to address this inefficiency. This mode allows for the addition and withdrawal of constraints between multiple queries, maintaining the solver's state across these queries.

Implement The implementation of incremental solving was focused on three primary areas:

- `visit_IfStmt`,
- `visit_WhileStmt`,
- `visit_AssertStmt`,

along with the addition of `push` and `pop` operations in the `SymState` class.

SymState We introduced `push` and `pop` methods in the `SymState` class. These methods manage the creation and reversion of local scopes within the Z3 solver. This allows the solver to retain its state and efficiently handle additional constraints or rollback as needed. Here is what we added to `SymState` class:

```
def push(self):
    """Create a new local scope."""
    self._solver.push()

def pop(self):
    """Revert to the previous local scope."""
    self._solver.pop()
```

visit_IfStmt The `visit_IfStmt` method was modified to utilize `push` and `pop` for handling the branching logic of if-statements. This enables the solver to consider each branch in isolation, adding constraints relevant to that branch only, and then reverting back to the previous state before exploring the other branch. Here is an example of how we modify `visit_IfStmt` to incremental solving mode.
`visit_IfStmt` in original solving mode:

```
st = kwargs['state'].fork()
st[0].add_pc(cond)
if not st[0].is_empty():
    states.extend(self.visit(node.then_stmt, state=st[0]))
```

`visit_IfStmt` in incremental solving mode:

```
st = kwargs['state']
st.push()
st.add_pc(cond)
if not st.is_empty():
    states = self.visit(node.then_stmt, state=st)
st.pop()
```

visit_WhileStmt The optimization of `visit_WhileStmt` in our symbolic execution engine focuses on addressing the inefficiencies of handling while-loops, especially in worst-case scenarios with multiple iterations. Initially, the `fork` operation was used for each iteration, leading to exponential growth in resource consumption with loops containing multiple iterations, such as 10 in our worst-case scenario. We restructured `visit_WhileStmt` to utilize Z3's incremental solving capabilities, replacing the fork operation with `push` and `pop`. This change efficiently manages the solver's state within each loop iteration without duplicating it. The result is a substantial reduction in computational overhead for loops with numerous iterations, enhancing performance significantly.Here is an example of how we modify `visit_WhileStmt` to incremental solving mode.
`visit_WhileStmt` in original solving mode:

```
st_exit, st_continue = each_state.fork()
if i != 0:
    st_exit.add_pc(z3.Not(cond))
    if not st_exit.is_empty():
        states.extend([st_exit])
```

`visit_WhileStmt` in incremental solving mode:

```
if i != 0:
    each_state.push()
    each_state.add_pc(z3.Not(cond))
    if not each_state.is_empty():
        states.extend([each_state])
    each_state.pop()
```

visit_AssertStmt In `visit_AssertStmt`, the incremental solving mode is used to check for potential assertion violations. By pushing a negation of the assertion condition and then popping the state, the engine efficiently determines if an assertion might be violated under certain conditions.Here is an example of how we modify `visit_AssertStmt` to incremental solving mode.
`visit_AssertStmt` in original solving mode:

```
st = kwargs['state'].fork()
st[0].add_pc(z3.Not(cond))
if not st[0].is_empty():
    print("Assertion error")
    st[0].mk_error()
```

`visit_AssertStmt` in incremental solving mode:

```
st = kwargs['state']
st.push()
if not st.is_empty():
    print("Assertion error")
    st.mk_error()
st.pop()
```

1.2 Theoretical Foundations

Incrementality in SMT Solving The concept of incrementality plays a pivotal role in Satisfiability Modulo Theories (SMT) solving, especially as it applies to our symbolic execution engine. Incrementality in SMT solvers, such as Z3, refers to the ability to check the satisfiability of assertions in a step-by-step manner. This process begins with an initial set of assertions, which are checked for satisfiability. Subsequently, additional assertions can be added for further checks. Crucially, these assertions can be retracted using scopes that are defined by `push` and `pop` operations.
Z3 specifically employs a one-shot solver during the first check for satisfiability. Upon subsequent calls to the solver, the default behavior is to switch to an incremental solver. This incremental solver utilizes the SMT core, which is a fundamental component of Z3's architecture. It's important to note that for use-cases not requiring the full feature set provided by the SMT core, specialized solvers can be more beneficial. These specialized solvers, such as those for finite domains (bit-vectors, enumeration types, bounded integers, and Booleans), are specified using the `QF_FD` logic and can offer more efficient solutions in specific contexts.

Scopes and Their Management The management of local scopes through `push` and `pop` operations is another foundational aspect of our implementation. In the context of Z3, `push` and `pop` operations are used to create and revert local scopes, respectively. Assertions added within a scope created by a `push` are retracted upon a matching `pop`. This mechanism allows for a dynamic and flexible handling of the solver's state, enabling it to adapt to changing conditions and requirements in a symbolic execution context. For instance, consider a typical session in Z3:

```
p, q, r = Bools('p q r')
s = Solver()
s.add(Implies(p, q))
s.add(Not(q))
print(s.check())
s.push()
s.add(p)
print(s.check())
s.pop()
print(s.check())
```

In this example, the initial assertions lead to a satisfiable state (`sat`). Upon adding new assertions within a `push`, the state changes to unsatisfiable (`unsat`). Finally, retracting these additional assertions with a `pop` reverts the state back to satisfiable.

1.3 Testing Strategy

Verification of Correctness Post-Modification The primary step in our testing strategy is to ensure the correctness of the modified code which now implements incremental solving. This involves running the existing test cases and comparing the output length (len(out)) before and after the modifications, which represents the number of distinct symbolic states generated by the symbolic execution engine for a given program. A consistent len(out) across pre- and post-modification indicates that the modified engine maintains functional integrity.

Performance Efficiency The next step is to evaluate the efficiency gains brought about by the incremental solving mode. Incremental solving optimizes performance by maintaining the solver's state across queries, thus avoiding redundant computations, especially in complex control structures like loops.

Efficiency Testing Comparing the average execution time of all test cases, we observed that the original code took 0.76 seconds, while the modified code with incremental solving averaged at 0.26 seconds. This implies a significant speedup, almost threefold.
A dedicated test case for a while-loop requiring 11 iterations demonstrates the efficiency of incremental solving in handling loops:

```
def test_5(self):
    prg1 = "havoc x; y:=2; while (not (x=10)) do {x:=x-1; y:=y-1}"
    ast1 = ast.parse_string(prg1)
    engine = sym.SymExec()
    st = sym.SymState()
    out = [s for s in engine.run(ast1, st)]
    self.assertEquals(len(out), 11)
```

Here is a diagram showing the performance comparison of the test case above in 100 runs: Fig. 1).

The execution time for this test case dropped from 0.276 seconds to 0.059 seconds post-modification, showcasing a notable performance improvement.

Test Coverage Our test suite achieves complete statement and branch coverage for the modified incremental solving code. This comprehensive coverage ensures that all new paths and branches introduced by the modifications are thoroughly tested, thereby guaranteeing the reliability and robustness of the updated engine.
The high coverage affirms that the modified engine behaves as expected across a wide range of scenarios, including complex control structures and edge cases. It also ensures that potential issues such as incorrect state management or unhandled cases in the incremental solving logic are identified and addressed, thereby enhancing the overall quality of the engine.

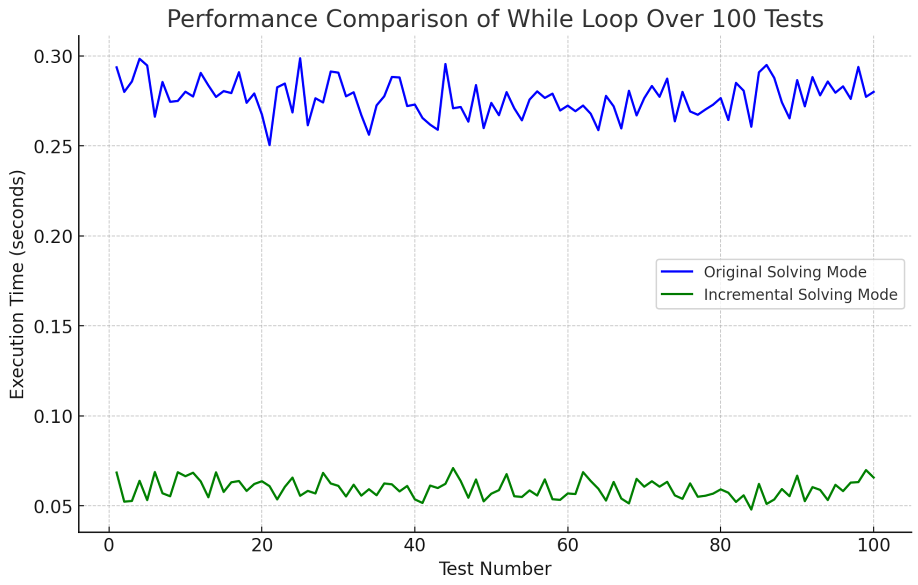

Fig. 1. Performance Comparison of While Loop Over 100 Tests

2 Concolic execution in EXE-style

2.1 Design Decisions

Motivation DSE, or Dynamic Symbolic Execution, is grounded in concolic execution, where a combination of symbolic and concrete execution is employed. In symbolic execution (SE), branches are navigated using symbolic values, and at times, certain paths prove challenging to traverse. This is where DSE comes in, executing those problematic paths concretely. When encountering a branch condition, the DSE approach involves executing one of the branches with concrete values to verify if the path condition holds. This technique was developed to overcome challenges faced by SE. DSE aims to circumvent the need for calling the solver for every branch condition. Various execution engines adopt dynamic strategies with diverse search approaches.

Implement We have developed a new program named dyn.py, incorporating Symbolic Execution (SE) and Dynamic Symbolic Execution (DSE) features. This program is built upon a pre-existing sym.py file. Changes were made in the following methods:

- `visit_IfStmt`,
- `visit_IntVar`,
- `visit_BoolConst`,
- `visit_IntConst`,

– `visit_AsgnStmt`,
– `visit_AssertStmt`,
– `visit_WhileStmt`,
– `visit_HavocStmt`,
– `visit_StmtList`,

visit_IfStmt The `visit_IfStmt` is one of the most important method in the Dynamic Symbolic Execution engine. It is designed to handle conditional branching represented by IfStmt nodes in the AST. First, it evaluates the condition by calling visit method in ast.py.

```
cond = self.visit(node.cond, *args, **kwargs)
```

It then proceeds to create two new states state1 and state2 (using fork()).

```
 for state in states:
     state.execType="sym"
     conds.append(self.visit(node.cond, *args, state=state))
     state.execType="con"
     conds.append(self.visit(node.cond, *args, state=state))
     state1,state2=state.fork()
```

For each state in the input list, the method checks the satisfiability of the condition in both the symbolic and concrete execution modes. If the condition is satisfiable, it updates the state's path condition with the condition expression.

```
if self.isSAT(conds[1],state,0)==True:
     state1.add_pc(conds[0])
     state1.execType="con"
     fst.append(self.visit(node.then_stmt, *args, state=[state1]))
     ....

 else:
     state1.add_pc(z3.Not(conds[0]))
     if node.has_else==True:
         state1.execType="sym"
         fst.append(self.visit(node.else_stmt, *args, state=[state1]))
     else:
         fst+=[state1]

     if self.isSAT(conds[0],state):
                 ....
```

Subsequently, the method explores the true branch for states where the condition is satisfied and the false branch for states where the negation of the condition is satisfied. This involves forking the symbolic states and updating their execution types accordingly.

visit_IntVar, visit_BoolConst, visit_IntConst All other methods like visit_IntVar(), visit_BoolConst(), visit_IntConst(), visit_RelExp, visit_BExp, and others had little changes done to them to facilitate both symbolic and dynamic execution.

```
def visit_IntVar(self, node, *args, **kwargs):
    st=kwargs["state"]
    match st.execType:
        case "con":
            return kwargs['state'].conEnv[node.name]
        case _:
            return kwargs['state'].symEnv[node.name]

def visit_BoolConst(self, node, *args, **kwargs):
    st=kwargs["state"]
    match st.execType:
        case "con":
            return node.val
        case _:
            return z3.BoolVal(node.val)

    return z3.BoolVal(node.val)

def visit_IntConst(self, node, *args, **kwargs):
    st=kwargs["state"]
    match st.execType:
        case "con":
            return node.val
        case _:
            return z3.IntVal(node.val)
```

visit_AsgnStmt In `visit_AsgnStmt`, method takes the state from kwargs and perform symbolic and concrete execution for assignment statements by iterating over states and updating their symbolic and concrete environments based on the assignment's left and right-hand sides.

```
def visit_AsgnStmt(self, node, *args, **kwargs):
    st = kwargs['state']
    i=0
    while i<len(st):
        st[i].execType="sym"
        st[i].symEnv[node.lhs.name] = self.visit(node.rhs, *args, state=st[i])

        st[i].execType="con"
        st.conEnv[node.lhs.name] = self.visit(node.rhs, *args, state=st[i])
        i+=1
    return st
```

visit_AssertStmt In `visit_AssertStmt`checks the satisfiability of the assertion condition in each state and collects the states where the assertion condition might be violated. If the assertion condition is unsatisfiable in a state, that state is not included in the result.

```
def visit_AssertStmt(self, node, *args, **kwargs):
    # Don't forget to print an error message if an assertion might be violated

    st = kwargs['state']
    fst=list()
    zero=0
    i=0
    while i<len(st):
        cond = self.visit(node.cond, *args, state=st[i])
        if self.isSAT(cond,st[i],0)==True:
            st[i].add_pc(cond)
            fst.append(st[i])
            i+=1

    return fst
```

visit_AssumeStmt In `visit_AssumeStmt` method considers the assumption condition and includes the states where the assumption condition is satisfiable, updating the path condition accordingly.

```
def visit_AssumeStmt(self, node, *args, **kwargs):
    cond = self.visit(node.cond, *args, **kwargs)
    st = kwargs["state"]
    fst=list()
    i=0
    while i<len(st):
        cond = self.visit(node.cond, *args, state=st[i])
        if self.isSAT(cond,st[i],0):
            st[i].add_pc(cond)
            fst.append(st[i])
        i+=1
    return fst
```

visit_HavocStmt In `visit_HavocStmt` handles the symbolic execution of Havoc statement nodes, which represent statements that introduce non-deterministic values into the program. The method takes havocstmt node as input with additional arguments and keyword arguments. It is responsible for handling non-deterministic assignments. It introduces fresh symbolic values and corresponding concrete values for each variable specified in the HavocStmt, allowing for exploration of multiple program paths during symbolic execution.

```
def visit_HavocStmt(self, node, *args, **kwargs):
    one=1
    st = kwargs['state']
    i=0
    j=0
    while i<len(st):
        while j<len(node.vars):
            st[i].symEnv[node.vars[j].name]=z3.FreshInt("R")
            st[i].conEnv[node.vars[j].name]=randomizer()
            j+=1
        i+=1
    st1=kwargs["state"]
    return st1
```

visit_WhileStmt The `visit_WhileStmt` responsible for handling the symbolic execution of while loops represented by WhileStmt nodes in the Abstract Syntax Tree (AST). The method systematically explores the possible execution paths within the loop, considering both the loop condition and the body of the loop. the method first evaluates the loop condition by invoking the symbolic execution visitor on the condition expression.

```
cond = self.visit(node.cond, *args, **kwargs)
```

It then pushes a new local scope to allow for changes in the path condition during the symbolic execution of the loop. The method iteratively explores the loop's body by repeatedly forking and updating symbolic states based on the loop condition. It manages different scenarios, including the initial state, states where the loop condition is satisfied, and states where the loop condition is not satisfied.

```
for i in range(0, 11):
    finish_states = []
    for each_state in start_states:
        print(f"Current state:{each_state}")
        cond = self.visit(node.cond, state=each_state)
. . .
```

For each iteration, the method considers the satisfiability of the loop condition and updates the state accordingly. It then explores the loop body using the symbolic execution visitor, incorporating the effects of the loop body on the symbolic state. The process continues until a predefined number of iterations or until the loop condition becomes unsatisfiable in a symbolic state.

visit_StmtList In `visit_StmtList` method initializes a list of symbolic states with the initial state obtained from the keyword arguments. It then iterates through each statement in the StmtList, invoking the symbolic visitor in ast.py for each statement. The method is recursive, and for each statement, it passes the current list of states as input to the next statement's execution. In conclusion, it serves as a mechanism for sequentially executing a series of statements.

```
ddef visit_StmtList(self, node, *args, **kwargs):
    states = [kwargs['state']]
    print(kwargs["state"])
    temp=dict(kwargs)
    for stmt in node.stmts:

        temp["state"] = states
        states = self.visit(stmt, *args, **temp)
    return states
```

2.2 Theoretical Foundations

Abstract Domain The abstract domain is the space in which abstract values (representing program states) reside. The code assumes an abstract domain where conditions and states are abstractly represented.

Galois Connection Abstract interpretation is often based on the concept of a Galois connection between concrete and abstract domains. The method visit_AssertStmt establishes a connection between the concrete program assertions and their abstract counterparts

Fixpoint Iteration Abstract interpretation often involves fixpoint computations, where the analysis iteratively refines abstract states until a fixpoint is reached. The loop in the code suggests an iterative process over the abstract states.

2.3 Testing Strategy

Unit Testing Unit testing is a software testing technique where individual units or components of a software application are tested independently to ensure that they perform as designed. A "unit" refers to the smallest testable part of the software, typically a function, method, or class.It Focuses on testing individual components rigorously, Verifying correctness of methods in DynState and DynExec and test methods like push, pop, add_pc, etc. We have to ensure unit tests cover a wide range of scenarios for robustness.

Integration Testing Integration testing is a software testing technique that focuses on verifying the interactions and interfaces between different components or modules of a software application. It ensures that integrated components work together as intended and identifies any issues that may arise when they are combined.it includes the following

- Component Interaction: Integration testing for the code which involves verifying how different components, such as DynState and DynExec classes, interact with each other.
- Interface Validation: It ensures that the interfaces between various modules or classes are correctly implemented and that the components communicate effectively.
- Dependencies Handling: The testing verifies that dependencies between components are managed appropriately and that changes in one component do not adversely affect others.