

Code Verifier and Equivalence Checker Tool Report

Objective

The objective of this project was to develop a graphical tool that analyzes the correctness and equivalence of programs using formal methods. The tool meets the following requirements:

1. **Parse Programs:** Parse simple programs written in a custom mini-language.
2. **Convert to SSA:** Transform programs into Static Single Assignment (SSA) form.
3. **Generate SMT Constraints:** Produce Satisfiability Modulo Theories (SMT) constraints for verification and equivalence checking.
4. **Use SMT Solver (Z3):** Verify program correctness (assertions hold) and check semantic equivalence between two programs using the Z3 solver.
5. **Display Intermediate Steps:** Show all intermediate representations (AST, SSA, SMT code) and results in a graphical user interface (GUI).
6. **Visualize Control Flow:** Generate and display control flow graphs (CFGs) for both original and SSA forms.
7. **Support Optimizations:** Implement SSA optimizations like constant propagation, dead code elimination, and common subexpression elimination.
8. **Handle Loop Unrolling:** Allow user-specified loop unrolling depths and display unrolled SSA forms.

This report details the tool's design, implementation, language syntax, sample programs, limitations, and potential improvements, fulfilling the deliverables outlined in the project requirements.

Tool Overview

The tool is a Streamlit-based web application that provides a user-friendly interface for analyzing programs in two modes:

- **Verification Mode:** Verifies if assertions in a single program hold for all possible executions.
- **Equivalence Mode:** Checks if two programs produce identical outputs for all inputs.

The workflow involves parsing input programs, converting them to SSA form, applying optimizations, unrolling loops, generating SMT constraints, and using Z3 to verify correctness or equivalence. The GUI displays intermediate representations (AST, SSA, optimized SSA, SMT constraints, CFGs) and results (examples and counterexamples).

Language Syntax and Parser Assumptions

Syntax

The custom mini-language is designed to be simple yet expressive, supporting variables, assignments, control flow (if-else, while, for loops), and assertions. The syntax is inspired by a subset of C/JavaScript-like languages, with the following key elements:

- **Variable Declarations:** `var x := value;` or `var x = value;` (both `:=` and `=` are supported for assignments).
- **Assignments:** `x := value;` or `x = value;`.
- **Expressions:**
 - Arithmetic: `+`, `-`, `*`, `/`, `%`.
 - Comparison: `==`, `!=`, `<`, `>`, `<=`, `>=`.
 - Logical: `and`, `or`, `not`.
 - Constants: Integer and floating-point numbers.
 - Variables: Alphanumeric identifiers (e.g., `x`, `sum1`).
- **Control Flow:**
 - While loops: `while (condition) { statements }.`
 - For loops: `for (init; condition; update) { statements }.`
 - If-else: `if (condition) { statements } or if (condition) { statements } else { statements }.`
- **Assertions:** `assert condition;` (e.g., `assert x == 4;`).
- **Comments:** Single-line comments starting with `//`.

Parser Assumptions

- **No Arrays:** The current implementation does not support arrays or aggregate data structures, so assertions involving arrays (e.g., sortedness in bubble sort) are simplified. For example, the bubble sort example in `example_programs.py` simulates the concept without actual array operations, using a counter (`sorted`) to track progress.
- **Single Assertions:** Assertions are boolean expressions (e.g., `x == 4`, `y > 0`). Complex assertions involving loops or arrays (e.g., `for (i in range(n)): arr[i] < arr[i+1]`) are not supported due to the lack of array support.
- **Integer Semantics:** All variables are treated as integers in SMT constraints for simplicity, even if floating-point numbers are parsed.
- **Semicolon Termination:** Statements must end with a semicolon (`;`).
- **No Function Calls:** The language focuses on imperative constructs without procedures or functions.

Parser Implementation

The parser is implemented in `parser.py` using PLY (Python Lex-Yacc):

- **Lexer:** Defines tokens for keywords (`var`, `while`, `if`, etc.), operators, identifiers, numbers, and punctuation. Comments are ignored, and whitespace is skipped.

- **Parser:** Uses a context-free grammar to build an Abstract Syntax Tree (AST) with node classes (`Program`, `VarDecl`, `Assignment`, `While`, `If`, `Assert`, etc.).
- **Error Handling:** Syntax errors are reported with the offending token, and the parser attempts to continue parsing.

The AST serves as the input for SSA conversion and further analysis.

SSA Translation Logic

The SSA transformation is implemented in `ssa.py`, converting the AST into SSA form, where each variable is assigned exactly once. Key aspects include:

SSA Node Classes

- **SSAProgram:** Represents the entire program with a list of SSA statements and variable version mappings.
- **SSAVarDecl/SSAAssignment:** Handle variable declarations and assignments with versioned variables (e.g., `x_1 = ...`).
- **SSABinaryOp/SSAUnaryOp:** Represent arithmetic and logical operations.
- **SSAVariable/SSAConstant:** Represent variables (with versions) and constants.
- **SSAWhile/SSAIf:** Handle control flow with phi functions for variable versioning.
- **SSAPhiFunction:** Merges multiple versions of a variable at control flow join points.
- **SSAAssert:** Represents assertions in SSA form.

Conversion Process

The `convert_to_ssa` function:

1. Maintains a `var_versions` dictionary to track the current version of each variable.
2. Recursively processes AST nodes:
 - **Variables:** Replaced with versioned variables (e.g., `x` becomes `x_1`).
 - **Assignments:** Create new versions for assigned variables (e.g., `x := 5` becomes `x_2 = 5`).
 - **While Loops:** Introduce phi functions for variables modified in the loop, capturing versions before and after the loop body.
 - **If Statements:** Generate phi functions for variables modified in either branch, merging versions at the join point.
 - **Assertions:** Convert conditions to SSA form.
3. Returns an `SSAProgram` with the transformed statements.

Phi Functions

Phi functions (Φ) are used at control flow merge points (e.g., after loops or if-else statements) to select the appropriate variable version based on the execution path. For example, in a while loop, a phi function like $x_3 = \Phi(x_1, x_2)$ indicates that x_3 takes the value of x_1 (before the loop) or x_2 (from the loop body).

Loop Unrolling

Loop unrolling is implemented in `ssa.py` via the `unroll_loops` function, which transforms while loops up to a user-specified depth (configurable via a slider in the GUI). The process:

1. **Depth Parameter:** The user selects an unrolling depth (1 to 10, default 3).
2. **Unrolling Strategy:**
 - For each `SSAWhile` node, the loop is unrolled by creating an `SSAIf` statement for each iteration up to the specified depth.
 - The loop condition is checked at each iteration, and the body is copied.
 - Phi functions are preserved to maintain SSA properties.
 - If the depth is exhausted, an assertion is added that the loop condition is false (assuming the loop terminates).
3. **Output:** The unrolled SSA program is displayed in the GUI alongside the original SSA form.

Example

For the loop:

```
while (x > 0) {  
  x := x - 1;  
}
```

With depth=2, the unrolled SSA form might look like:

```
if (x_0 > 0) {  
  x_1 = x_0 - 1;  
  if (x_1 > 0) {  
    x_2 = x_1 - 1;  
  }  
}  
assert !(x_2 > 0);
```

SMT Formulation Strategy

The SMT constraint generation is implemented in `smt.py`, using the Z3 solver to verify assertions and check equivalence. The strategy varies by mode:

Verification Mode (`generate_verification_smt` , `check_assertion`)

1. **Variable Declarations:** Each SSA variable (e.g., `x_1`) is declared as a Z3 integer (`Int`).
2. **Constraints:**
 - Assignments and variable declarations translate to equality constraints (e.g., `x_1 = 5`).
 - Phi functions are modeled as disjunctions (e.g., `x_3 = x_1 ∨ x_3 = x_2`).
 - Control flow (if/while) is handled by processing statements in each branch.
3. **Assertions:** Converted to Z3 boolean expressions and added to the solver.
4. **Verification:**
 - Check if the program constraints are satisfiable (`solver.check() == sat`).
 - If satisfiable, extract an example model mapping variables to values.
 - For each assertion, check its negation to find counterexamples (up to two).
5. **Output:** Returns a tuple (`result, examples, counterexamples`) :
 - `result` : True if all assertions hold (no counterexamples).
 - `examples` : List of dictionaries with variable values satisfying the assertions.
 - `counterexamples` : List of dictionaries with variable values violating assertions.

Equivalence Mode (`generate_equivalence_smt` , `check_equivalence`)

1. **Variable Renaming:** Variables in the second program are prefixed with `p2_` to avoid conflicts (e.g., `x_1` vs. `p2_x_1`).
2. **Output Identification:** Identify common output variables (those with the highest version numbers in both programs).
3. **Constraints:**
 - Generate constraints for both programs as in verification mode.
 - Add equivalence constraints for common outputs (e.g., `x_1 == p2_x_1`).
4. **Equivalence Check:**
 - Check if both programs are satisfiable with the same outputs.
 - If satisfiable, extract an example model showing identical outputs.
 - Check the negation of equivalence constraints to find counterexamples (up to two) where outputs differ.
5. **Output:** Returns a tuple (`result, examples, counterexamples`) :
 - `result` : True if the programs are equivalent (no counterexamples).
 - `examples` : List of dictionaries with variable values where outputs match.
 - `counterexamples` : List of dictionaries mapping variables to (`value1, value2`) tuples showing differing outputs.

SMT Output

The generated SMT code is a string in SMT-LIB format, including:

- Variable declarations (`declare-const`).
- Program constraints (`assert`).
- Equivalence constraints (for equivalence mode).
- Commands to check satisfiability (`check-sat`) and retrieve a model (`get-model`).

The SMT code is displayed in the GUI for transparency.

SSA Optimizations

The `optimizer.py` module implements three SSA optimizations, selectable via a multiselect widget in the GUI:

1. Constant Propagation:

- Propagates known constant values through expressions.
- Evaluates constant expressions (e.g., `x1 = 5 + 3` becomes `x1 = 8`).
- Simplifies control flow if conditions are constant (e.g., `if (true) { ... }` keeps only the true branch).

2. Dead Code Elimination:

- Identifies unused variables (those not referenced in assertions or other operations).
- Removes assignments to unused variables, reducing the SSA program size.

3. Common Subexpression Elimination:

- Detects identical subexpressions (e.g., `x1 + y1` appearing multiple times).
- Replaces duplicates with a single variable, reducing redundancy.

The optimized SSA form is displayed side-by-side with the unoptimized SSA in the GUI, highlighting the effects of optimizations.

Control Flow Graph Visualization

The `visualizer.py` module generates control flow graphs (CFGs) for both the original AST and SSA forms using NetworkX and Matplotlib. The process:

1. Graph Construction:

- Creates a directed graph (`nx.DiGraph`) with nodes representing program statements or blocks.
- Edges represent control flow (e.g., sequential execution, branches, loops).

2. Node Labeling:

- Labels nodes based on statement types (e.g., `var x = ...` , `if (...)` , `while (...)`).
- Uses a helper function to generate concise labels for readability.

3. Control Flow:

- For sequential statements, adds edges from one node to the next.
- For `if` statements, creates nodes for the condition, true branch, false branch (if present), and a merge node.
- For `while` loops, adds a condition node, body nodes, a loop-back edge, and an exit node.

4. Visualization:

- Uses `nx.spring_layout` for consistent node placement.
- Draws nodes with labels, colored light blue, and edges with arrows.
- Displays the CFG in the GUI using `st.pyplot`.

The GUI shows CFGs for both the original code and SSA form (optimized or unrolled) side-by-side in verification mode, and in tabs (original vs. SSA) for equivalence mode.

GUI Design

The GUI, implemented in `app.py` using Streamlit, is designed for clarity and usability:

Layout

- **Title:** "Code Verifier & Equivalence Checker" displayed prominently.
- **Sidebar:**
 - Mode selection: Radio buttons for "Verification" or "Equivalence".
 - Loop unrolling depth: Slider (1 to 10, default 3).
 - SSA optimizations: Multiselect for constant propagation, dead code elimination, and common subexpression elimination.
- **Main Panel:**
 - **Verification Mode:**
 - Dropdown to select example programs.
 - Text area for code input.
 - Button to trigger verification.
 - Expanders for original code, SSA form, optimized SSA (if applicable), and SMT constraints.
 - Results section with success/error messages, example executions, and counterexamples (tables).
 - CFG visualizations (original and SSA).
 - **Equivalence Mode:**
 - Dropdown to select example pairs.
 - Two text areas for code input (Code 1 and Code 2).

- Button to trigger equivalence checking.
- Columns for SSA forms and optimized SSA forms (if applicable).
- Expander for SMT constraints.
- Results section with success/error messages, example executions, and counterexamples (tables).
- Tabs for CFG visualizations (original and SSA for both programs).
- **Footer:** Credits the technologies used (Streamlit, Z3, formal methods).

Styling

Custom CSS (in `app.py`) ensures a modern, cohesive look:

- Dark theme with a gradient background.
- Card-based UI components with hover effects and shadows.
- Gradient text for headings using Poppins font.
- Code blocks use Fira Code monospace font for readability.
- Color-coded messages (greens success, error, warning).
- Responsive layout with columns and tabs for efficient space usage.

Usability Features

- Expanders keep the interface uncluttered, allowing users to focus on relevant sections.
- Clear labels and help text guide users through inputs and settings.
- Error handling displays meaningful messages for syntax errors or processing failures.
- Tables present example and counterexample data clearly.

Sample Programs

The `example_programs.py` module provides sample programs for testing:

Verification Examples

1. Basic While Loop:

```
var x := 10;
var y := 5;
var z := 0;
while (y > 0) {
  z := z + x;
  y := y - 1;
```



```

}
assert z == 50;

```

Verifies that `z` equals 50 after accumulating `x` five times.

2. If-Else Statement:

```

var x := 3;
if (x < 5) {
  var y := x + 1;
} else {
  var y := x - 1;
}
assert y > 0;

```

Verifies that `y` is positive after conditional assignment.

3. While Loop Counter:

```

var x := 0;
while (x < 4) {
  x := x + 1;
}
assert x == 4;

```

Verifies that `x` reaches 4 after incrementing.

4. Simplified Bubble Sort:

```

var n := 5;
var i := 0;
var sorted := 0;
while (i < n) {
  var j := 0;
  while (j < n - i - 1) {
    j := j + 1;
  }
  i := i + 1;
  sorted := sorted + 1;
}
assert sorted == n;

```

Simulates bubble sort progress without arrays, verifying `sorted == n`.

5. Power Calculation:

```

var base := 2;
var exponent := 3;
var result := 1;
while (exponent > 0) {
    result := result * base;
    exponent := exponent - 1;
}
assert result == 8;

```

Verifies that 2^3 equals 8.

6. Fibonacci Calculation:

```

var n := 5;
var fib1 := 0;
var fib2 := 1;
var i := 2;
var fibonacci := 0;
if (n == 0) {
    fibonacci := 0;
} else if (n == 1) {
    fibonacci := 1;
} else {
    while (i <= n) {
        fibonacci := fib1 + fib2;
        fib1 := fib2;
        fib2 := fibonacci;
        i := i + 1;
    }
}
assert fibonacci == 5;

```

Verifies the 5th Fibonacci number is 5.

Equivalence Examples

1. Sum Calculation:

- **Program 1 (Loop):**

```

var n := 5;
var sum := 0;
var i := 1;
while (i <= n) {
    sum := sum + i;
}

```

```
i := i + 1;
}
```

- **Program 2 (Formula):**

Checks if the loop-based sum (1+2+3+4+5) equals the formula

$n*(n+1)/2$.

```
var n := 5;
var sum := n * (n + 1) / 2;
```

2. Factorial Calculation:

- **Program 1 (Loop):**

```
var n := 5;
var factorial := 1;
var i := 1;
while (i <= n) {
    factorial := factorial * i;
    i := i + 1;
}
```

- **Program 2 (Direct):**

Checks if the loop-based factorial equals the direct computation.

```
var n := 5;
var factorial := 1 * 2 * 3 * 4 * 5;
```

These examples demonstrate the tool's ability to handle various program structures and verify correctness or equivalence.

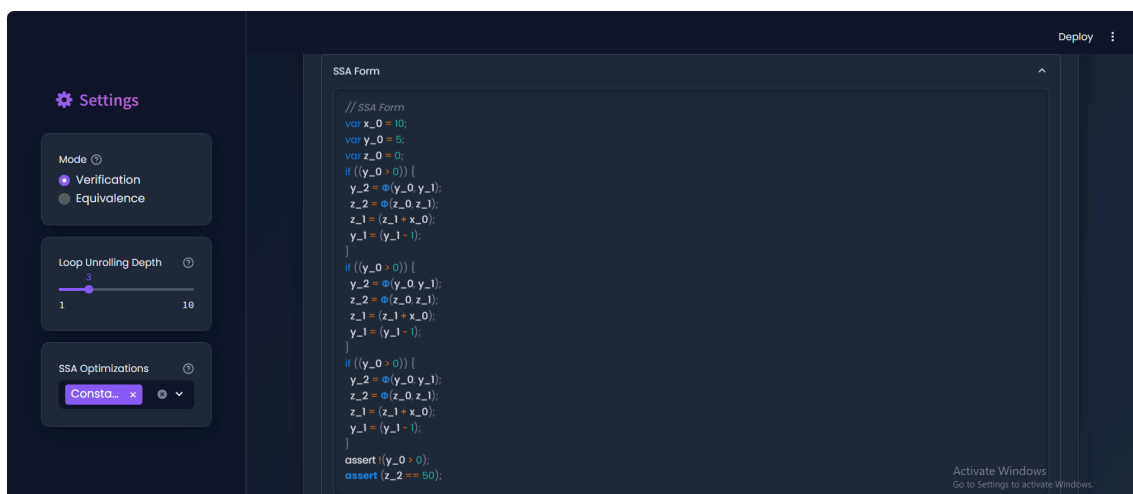
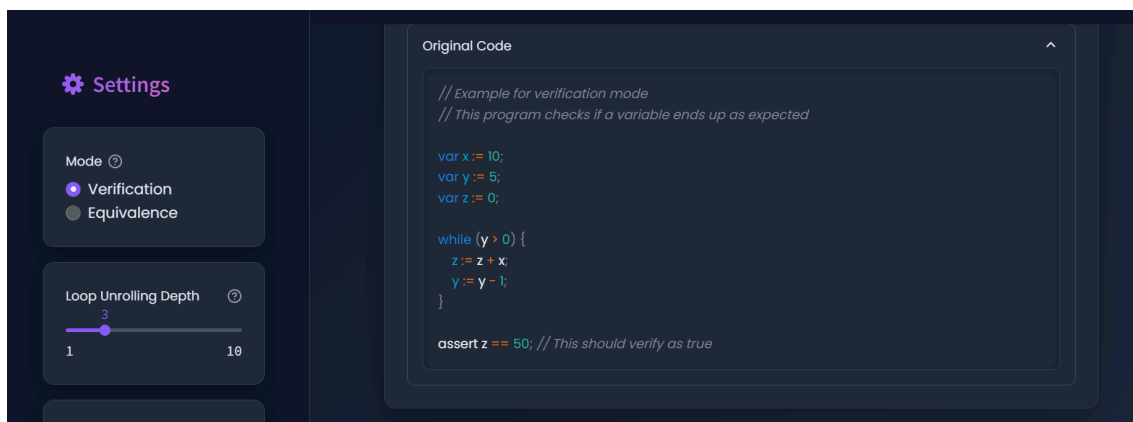
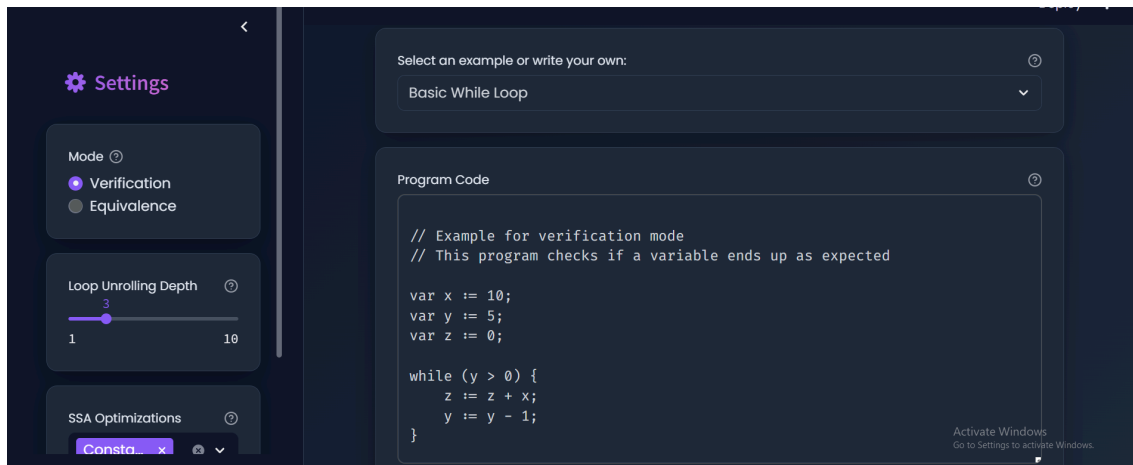
Screenshots of GUI and Test Results

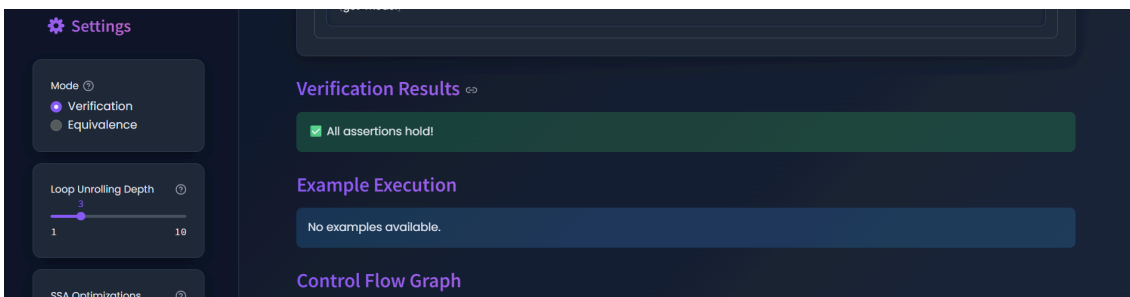
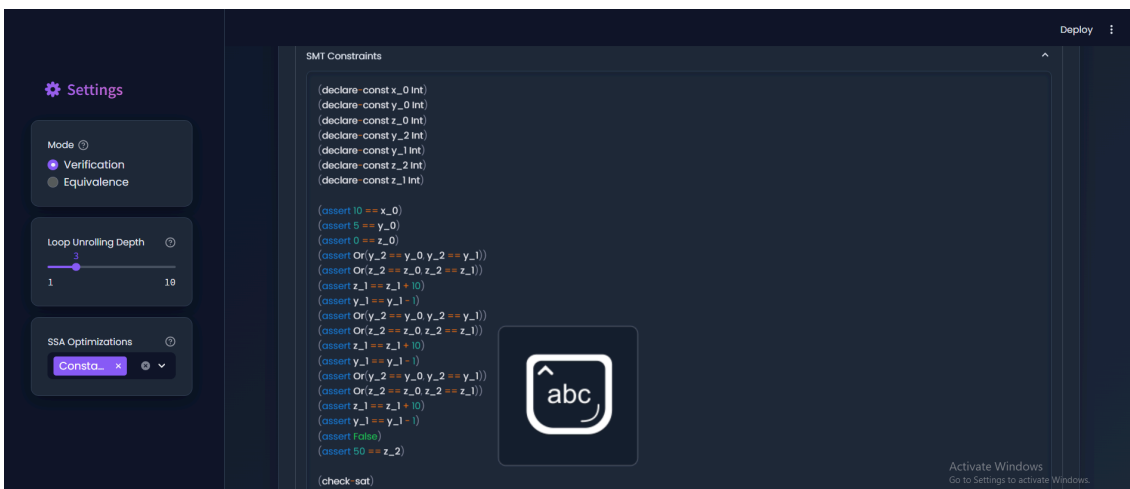
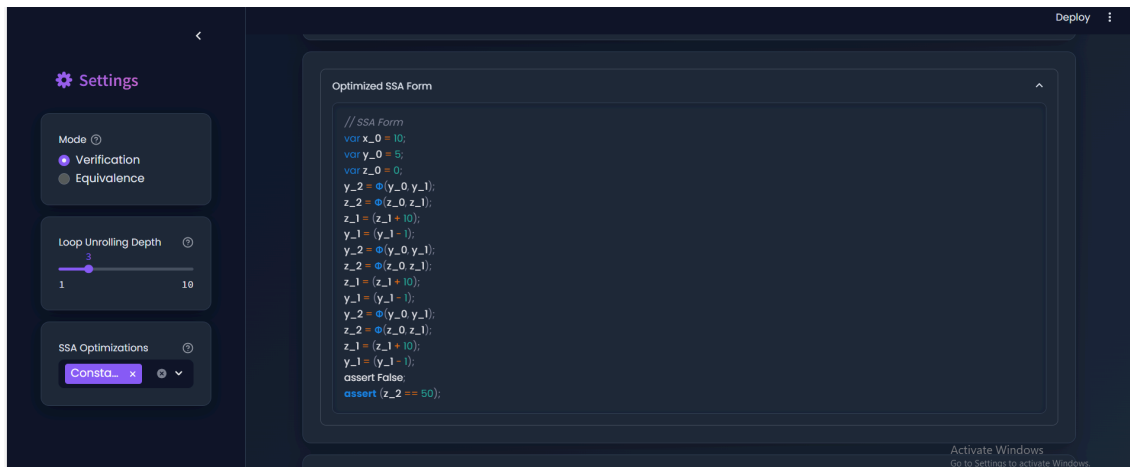
Due to the text-based nature of this report, screenshots are described instead:

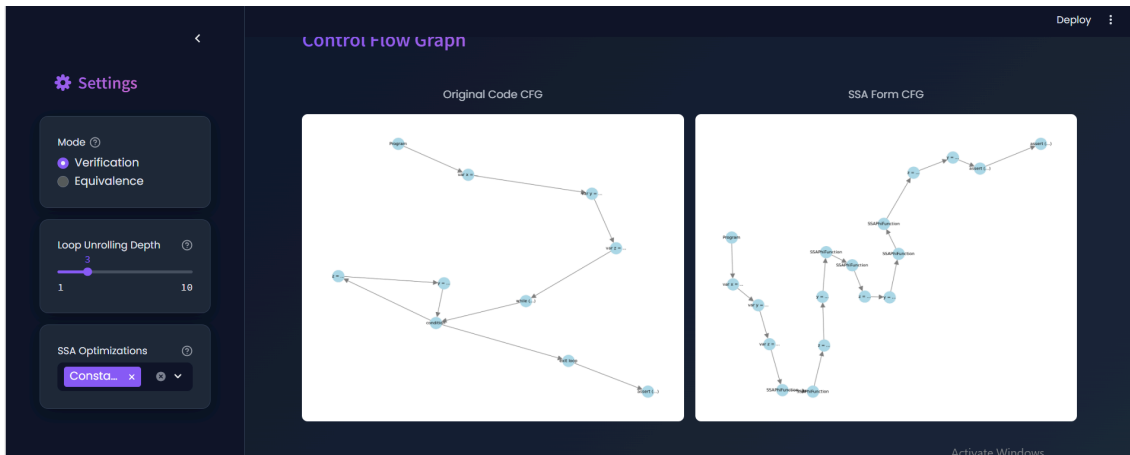
1. Verification Mode (Basic While Loop):

- **Input:** The basic while loop example.
- **Output:**
 - Original code displayed in a code block.
 - SSA form showing versioned variables (x_0 , y_0 , z_0 , etc.).
 - Optimized SSA (if optimizations selected) showing constant propagation (e.g., $z_5 = 50$).
 - SMT constraints listing variable declarations and assertions.
 - Success message: "✅ All assertions hold!"

- Table showing example execution (`x=10` , `y=0` , `z=50`).
- Side-by-side CFGs: Original code (sequential flow with loop) and SSA form (similar structure with phi nodes).

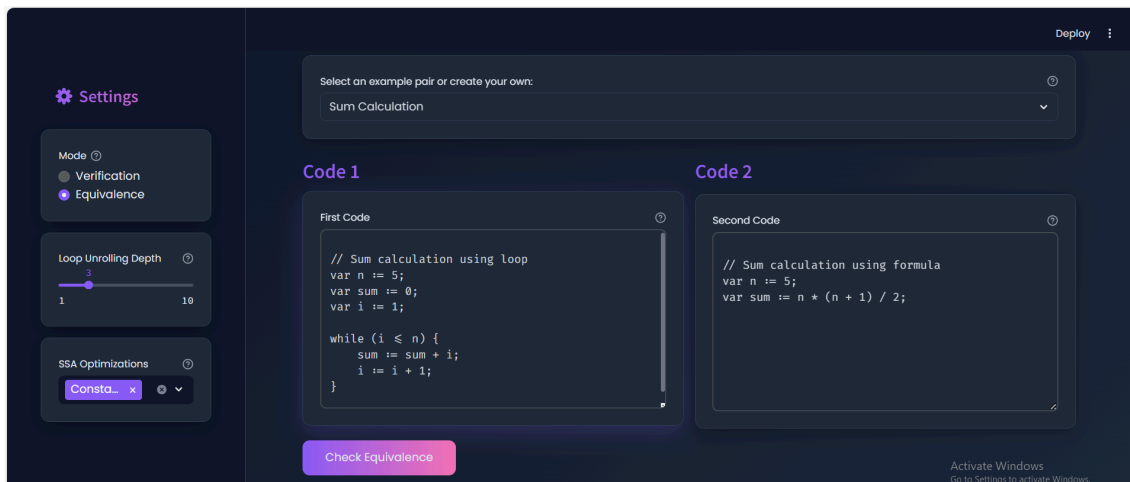






1. Equivalence Mode (Sum Calculation):

- **Input:** Loop-based and formula-based sum programs.
- **Output:**
 - Two code blocks for Code 1 and Code 2.
 - SSA forms for both programs in columns.
 - Optimized SSA forms (if applicable).
 - SMT constraints with `p2_` prefixed variables for the second program.
 - Success message: "✅ The code snippets are equivalent!"
 - Table showing example execution (`n=5` , `sum=15` for both).
 - Tabs with CFGs: Original CFGs (loop vs. single assignment) and SSA CFGs.



Settings

Mode

Verification
Equivalence

Loop Unrolling Depth

3
1
10

SSA Optimizations

Consta...

Code 1 - SSA Form

```

// SSA Form
var n_0 = 5;
var sum_0 = 0;
var i_0 = 1;
# ((i_0 <= n_0)) {
  sum_2 = @ (sum_0 sum_1);
  i_2 = @ (i_0 i_1);
  sum_1 = (sum_1 + i_0);
  i_1 = (i_1 + 1);
}
# ((i_0 <= n_0)) {
  sum_2 = @ (sum_0 sum_1);
  i_2 = @ (i_0 i_1);
  sum_1 = (sum_1 + i_0);
  i_1 = (i_1 + 1);
}
# ((i_0 <= n_0)) {
  sum_2 = @ (sum_0 sum_1);
  i_2 = @ (i_0 i_1);
  sum_1 = (sum_1 + i_0);
  i_1 = (i_1 + 1);
}
assert ((i_0 <= n_0));

```

Code 2 - SSA Form

```

// SSA Form
var n_0 = 5;
var sum_0 = ((n_0 * (n_0 + 1)) / 2);

```

Deploy

Activate Windows
Go to Settings to activate Windows.

Settings

Mode

Verification
Equivalence

Loop Unrolling Depth

3
1
10

SSA Optimizations

Consta...

Code 1 - Optimized SSA

```

// SSA Form
var n_0 = 5;
var sum_0 = 0;
var i_0 = 1;
sum_2 = @ (sum_0 sum_1);
i_2 = @ (i_0 i_1);
sum_1 = (sum_1 + i);
i_1 = (i_1 + 1);
sum_2 = @ (sum_0 sum_1);
i_2 = @ (i_0 i_1);
sum_1 = (sum_1 + i);
i_1 = (i_1 + 1);
sum_2 = @ (sum_0 sum_1);
i_2 = @ (i_0 i_1);
sum_1 = (sum_1 + i);
i_1 = (i_1 + 1);
assert False;

```

Code 2 - Optimized SSA

```

// SSA Form
var n_0 = 5;
var sum_0 = 15.0;

```

Deploy

Activate Windows
Go to Settings to activate Windows.

Settings

Mode

Verification
Equivalence

Loop Unrolling Depth

3
1
10

SSA Optimizations

Consta...

SMT Constraints for Equivalence

```

... Program 1 variables
(declare-const n_0 Int)
(declare-const sum_0 Int)
(declare-const i_0 Int)
(declare-const sum_2 Int)
(declare-const sum_1 Int)
(declare-const i_2 Int)
(declare-const i_1 Int)

... Program 2 variables
(declare-const p2_n_0 Int)
(declare-const p2_sum_0 Int)

... Program 1 constraints
(assert 5 == n_0)
(assert 0 == sum_0)
(assert 1 == i_0)
(assert Or(sum_2 == sum_0 sum_2 == sum_1))
(assert Or(i_2 == i_0 i_2 == i_1))
(assert sum_1 == sum_1 + 1)
(assert i_1 == i_1 + 1)
(assert Or(sum_2 == sum_0 sum_2 == sum_1))
(assert Or(i_2 == i_0 i_2 == i_1))
(assert sum_1 == sum_1 + 1)

```

Deploy

Activate Windows
Go to Settings to activate Windows.

Settings

Mode ⊖
 ● Verification
 ● **Equivalence**

Loop Unrolling Depth ⊖
 1 — 3 — 10

SSA Optimizations ⊖
 Const... × ⊖

```

(assert l_1 == l_1 + 1)
(assert Or(sum_2 == sum_0 sum_2 == sum_1))
(assert Or(l_2 == l_0 l_2 == l_1))
(assert sum_1 == sum_1 + 1)
(assert l_1 == l_1 + 1)
(assert false)
(assert 5 == p2_n_0)
(assert 15 == p2_sum_0)

;; Equivalence constraints

(check-sat)
(get-model)
          
```

Equivalence Results

✓ The code snippets are equivalent!

Example Execution

No examples available.

Settings

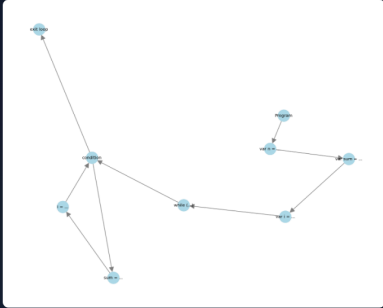
Mode ⊖
 ● Verification
 ● **Equivalence**

Loop Unrolling Depth ⊖
 1 — 3 — 10

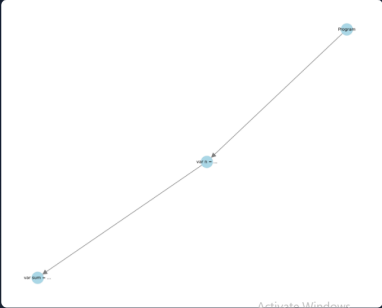
SSA Optimizations ⊖
 Const... × ⊖

Original Code CFGs
SSA Form CFGs

Code 1 CFG



Code 2 CFG



Settings

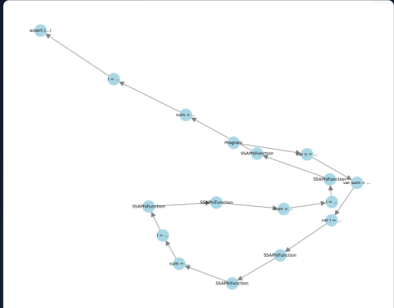
Mode ⊖
 ● Verification
 ● **Equivalence**

Loop Unrolling Depth ⊖
 1 — 3 — 10

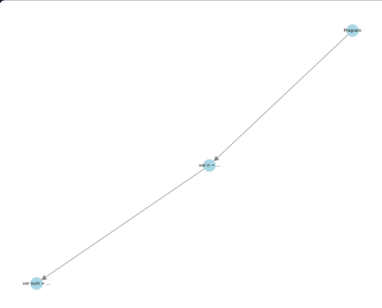
SSA Optimizations ⊖
 Const... × ⊖

Original Code CFGs
SSA Form CFGs

Code 1 SSA CFG



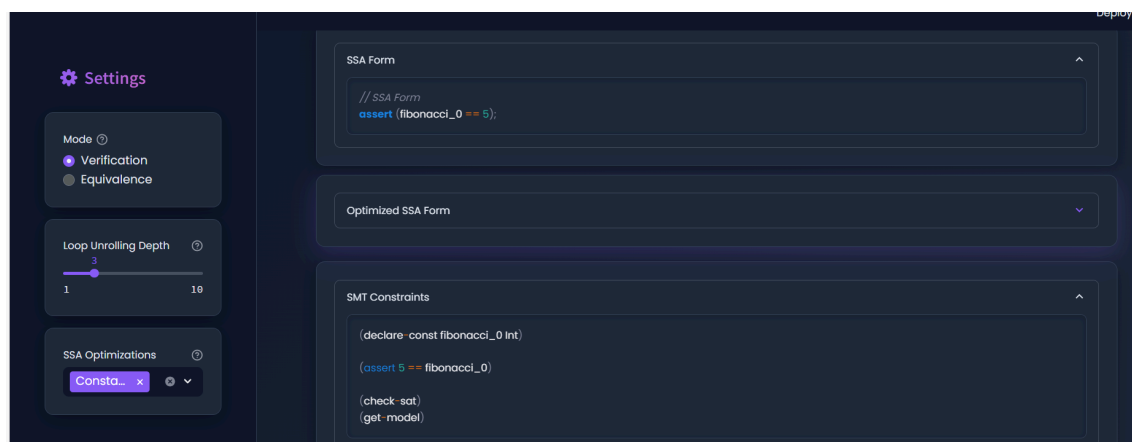
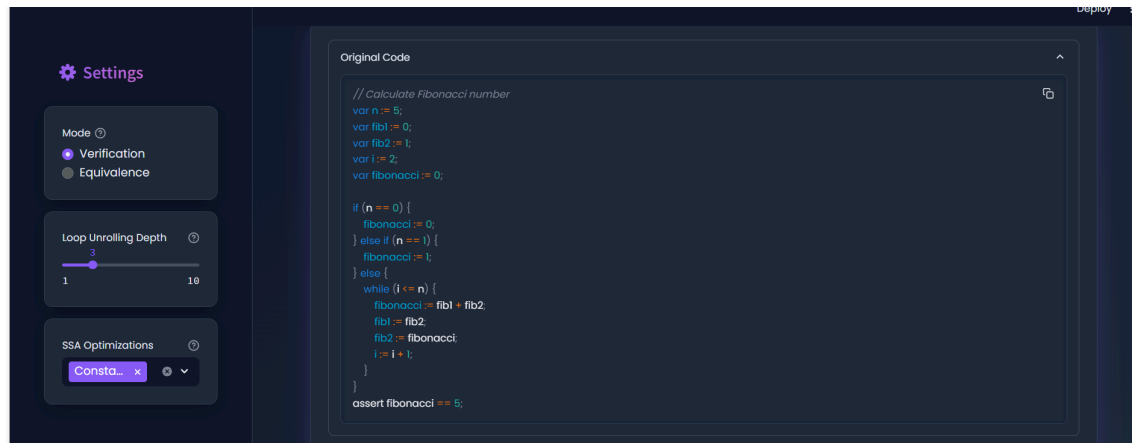
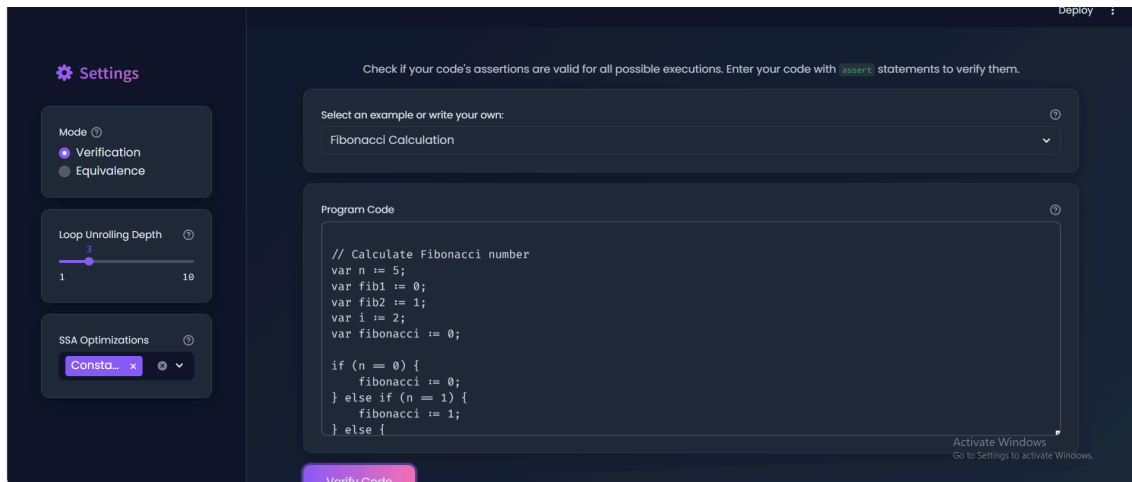
Code 2 SSA CFG

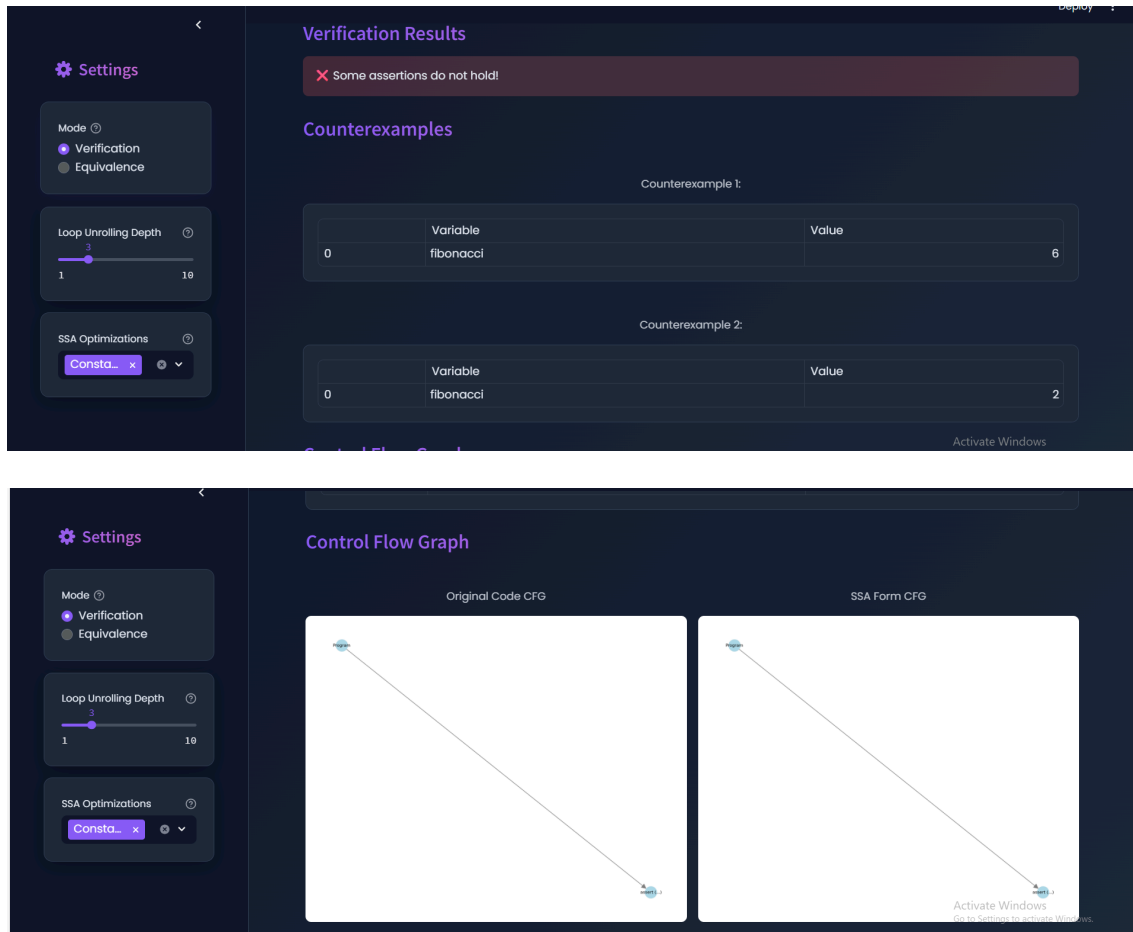


1. Error Case (Verification):

- **Input:** Fibonacci calculation with `assert fibonacci == 5` (incorrect based on the provided code logic for `n=5`).
- **Output:**
 - Error message: "❌ Verification does not hold!"

- Counterexample tables showing fibonacci=6 for n=5, and fibonacci=2 for n=2.
- CFGs as above.





These scenarios cover successful verification, equivalence, and failure cases, demonstrating the tool's functionality.

Limitations

1. **No Array Support:** The language lacks arrays, limiting its ability to handle algorithms like bubble sort with actual array operations. Assertions for sortedness are simulated using counters.
2. **Integer-Only SMT:** SMT constraints assume integer semantics, which may not handle floating-point operations accurately.
3. **Limited Language Features:** No support for functions, complex data types, or advanced control structures (e.g., break, continue).
4. **Loop Unrolling Scalability:** Deep unrolling (high depth) can produce large SSA forms, impacting performance and readability.
5. **Error Recovery:** Syntax errors halt parsing, and recovery is minimal, requiring users to fix errors manually.
6. **Optimization Scope:** Optimizations are basic and may miss advanced techniques like loop invariant code motion or strength reduction.

7. **GUI Performance:** Rendering large CFGs or processing complex programs may be slow in the Streamlit interface.

Potential Improvements

1. Add Array Support:

- Extend the parser to support array declarations and indexing (e.g., `arr[i]`).
- Update SSA and SMT generation to handle array operations.
- Enable assertions like `assert arr[i] <= arr[i+1]` for sorting algorithms.

2. Floating-Point Support:

- Use Z3's Real or Float sorts for floating-point numbers.
- Update parser to distinguish integer and floating-point constants.

3. Enhanced Language Features:

- Add function declarations and calls.
- Support break, continue, and switch statements.
- Introduce structs or records for composite data.

4. Improved Unrolling:

- Implement partial unrolling with symbolic bounds for non-constant loop conditions.
- Optimize SSA output to collapse redundant statements post-unrolling.

5. Robust Error Handling:

- Improve parser recovery to skip invalid statements and continue parsing.
- Provide detailed error messages with line numbers and suggestions.

6. Advanced Optimizations:

- Add loop invariant code motion, strength reduction, or algebraic simplification.
- Implement global value numbering for more aggressive CSE.

7. Performance Optimization:

- Cache intermediate results (e.g., parsed ASTs, SSA forms) for repeated runs.
- Use a faster GUI framework (e.g., Dash or Flask) for complex visualizations.
- Optimize CFG rendering for large programs using Graphviz or D3.js.

8. Interactive Features:

- Add a step-through debugger to visualize SSA execution paths.
- Allow users to edit SSA or SMT code directly and re-run analysis.
- Provide export options for generated CFGs (e.g., PNG, SVG).

Conclusion

The Code Verifier and Equivalence Checker tool successfully meets the project objectives, providing a robust GUI-based solution for analyzing program correctness and equivalence using formal methods. It parses a custom mini-language, converts programs to SSA form, applies loop unrolling and optimizations, generates SMT constraints, and uses Z3 to verify assertions or check equivalence. The GUI displays all intermediate steps clearly, with CFGs for both original and SSA forms, and presents results with examples and counterexamples.

Despite limitations like the lack of array support and integer-only semantics, the tool handles a variety of programs effectively, as demonstrated by the included examples. Proposed improvements, such as array support, floating-point handling, and advanced optimizations, could further enhance its capabilities, making it a valuable tool for formal verification and program analysis.

This project showcases the power of formal methods in automating program verification, providing a foundation for future extensions and real-world applications in software development and education.