Formal Methods in Software Engineering

Project

Course Instructor

Madam Laiba Imran

Group Members

Abdullah Daoud……(22I-2626)

Usman Ali……(22I-2725)

Faizan Rasheed……(22I-2734)

Section

SE-E

Date

Sunday, May 11th, 2025

Spring 2025

Department of Software Engineering

FAST – National University of Computer & Emerging Sciences

Islamabad Campus

Table of Contents

MiniLang Tool Project Report

This report provides an in-depth overview of the MiniLang Tool, a software solution designed to verify program correctness and equivalence using Static Single Assignment (SSA) form, loop unrolling, and SMT-based reasoning. The report is structured into sections detailing the language syntax and parser assumptions, SSA translation logic, loop unrolling mechanisms, SMT formulation strategy, GUI features with test results, and concludes with an analysis of limitations and proposed improvements.

1. Language Syntax and Parser Assumptions

1.1 MiniLang Syntax

MiniLang is a minimalistic programming language tailored for formal verification tasks. Its syntax supports a variety of constructs essential for expressing basic algorithms and their properties:

- **Assignments**:
  - **Scalar Assignments**: Written as variable := expression; (e.g., x := 5 + 3;), where expression can include arithmetic operators (+, -, *, /) and variables.
  - **Array Assignments**: Expressed as array[index] := expression; (e.g., arr[i] := x * 2;), supporting single-dimensional arrays with integer indices.
- **Control Flow Constructs**:
  - **If-Else Statements**: Structured as if (condition) { statements } else { statements }, where condition is a boolean expression (e.g., x < y, a == b).
  - **While Loops**: Defined as while (condition) { statements }, allowing repeated execution based on a dynamic condition.
  - **For Loops**: Written as for (init; condition; update) { statements }, mimicking traditional C-style loops (e.g., for (i := 0; i < n; i := i + 1) { arr[i] := 0; }).
- **Assertions**: Formatted as assert(condition);, used to specify properties that must hold post-execution (e.g., assert(x > 0);).

1.2 Parser Assumptions

The parser, implemented in core/parser.py, processes MiniLang code into an Abstract Syntax Tree (AST) with the following assumptions:

- **Well-Formed Input**: The parser expects syntactically correct code, free of unmatched braces, missing semicolons, or invalid operators. Errors like if (x < 5 {} result in parsing failures.
- **Array Handling**: Arrays are parsed as ArrayAccess or ArrayAssign nodes. Indices can be variables or expressions (e.g., arr[j + 1]), but multi-dimensional arrays (e.g., arr[i][j]) are unsupported.

Assertion Format: Assertions must be standalone statements (assert(condition);). Universal quantifiers like forall are not supported; instead, array properties are verified using explicit loops. For instance, to assert an array is sorted:

```
for (k := 0; k < n - 1; k := k + 1) {
    assert(arr[k] <= arr[k + 1]);
}
```

- **Expression Parsing**: Arithmetic and comparison operators are fully supported, with operator precedence following standard mathematical rules (e.g., * before +).

2. SSA Translation Logic

The SSA conversion, handled by SSAConverter in core/ssa.py, transforms MiniLang code into SSA form, ensuring each variable is assigned exactly once. Key components include:

- **Variable Versioning**: Each assignment generates a new version of a variable (e.g., x_1, x_2). A dictionary tracks versions, incrementing counters per variable assignment.

Phi Functions: Used at control flow merge points to reconcile variable versions. A phi function φ(condition, value1, value2) selects between values based on the preceding condition. For example:

```
if (x_1 > 0) {
   y_1 := x_1;
} else {
   y_2 := 0;
}
y_3 := φ(x_1 > 0, y_1, y_2);
```

- **Array Representation**: Array elements are treated as distinct variables in SSA. An access like arr[j] becomes arr_j_1, where the index is symbolically incorporated into the variable name. Assignments update versions (e.g., arr_j_2 := x_1).
- **Loop Processing**: Loops introduce phi functions at their headers to merge initial and iterative values. For a for loop:

```
for (i := 0; i < n; i := i + 1) {
   x := x + 1;
}
```

SSA form:

```
i_1 := 0
x_1 := initial_x
i_2 := φ(is_first_iter, i_1, i_3)
x_2 := φ(is_first_iter, x_1, x_3)
for_cond := i_2 < n
x_3 := x_2 + 1
```

i_3 := i_2 + 1

2.1. Example Translation

Input Code:

```
x := 0;
while (x < 5) {
    x := x + 1;
}
assert(x == 5);
```

SSA Output:

```
x_1 := 0
x_2 := φ(is_first_iter, x_1, x_3)
while_cond := x_2 < 5
x_3 := x_2 + 1
x_4 := φ(while_cond, x_3, x_2)
assert := x_4 == 5
```

3. How Unrolling is Handled

Loop unrolling, implemented in core/unroller.py via the LoopUnroller class, expands loop iterations to facilitate verification:

- **Unrolling Depth**: Users specify a depth (e.g., 3) through the GUI, controlling how many iterations are unrolled.
- **For Loop Unrolling**: The loop body and update statements are replicated within conditional blocks. For depth 2:

```
for (i := 0; i < n; i := i + 1) {
    x := x + i;
}
```

Unrolled Code:

```
i := 0;
x_0 := initial_x;
if (i < n) {
    x_1 := x_0 + i;
    i := i + 1;
}
if (i < n) {
    x_2 := x_1 + i;
    i := i + 1;
}
```

- **While Loop Unrolling**: Similar to `for` loops, the body is repeated with `if` statements checking the loop condition:

```
x := 0;
while (x < 3) {
    x := x + 1;
}
```

Unrolled with Depth 2:

```
x := 0;

if (x < 3) {

   x := x + 1;

}

if (x < 3) {

   x := x + 1;

}
```

- **Symbolic Updates**: Variables and conditions remain symbolic (e.g., `i < n`), enabling the SMT solver to reason about general cases rather than concrete values.

The unrolled code is then converted to SSA form, displayed in the GUI alongside the original and intermediate representations.

4. SMT Formulation Strategy

The SMTGenerator class in core/smt_generator.py converts SSA instructions into SMT-LIB format for the Z3 solver:

- **Variable Declarations**: Variables are declared based on their SSA context:
  - Integers: (declare-const x_1 Int)
  - Booleans: (declare-const cond_1 Bool)
  - Input variables are identified as those without prior assignments.
- **Instruction Translation**:
  - Assignments: x_1 := 5 becomes (assert (= x_1 5)).
  - Phi Functions: y_3 := φ(cond_1, y_1, y_2) translates to (assert (ite cond_1 (= y_3 y_1) (= y_3 y_2))).
  - Conditions: x_1 < y_2 becomes (assert (< x_1 y_2)).
- **Assertions**: Collected as separate constraints (e.g., (assert (> y_3 0))). Z3 checks satisfiability; if satisfiable, it provides a model; if unsatisfiable, it offers counterexamples (up to two).

- **Equivalence Checking**: For two programs, their SMT constraints are merged, and output variables' final versions are compared (e.g., (assert (not (= x_5 y_3)))). If unsatisfiable, the programs are equivalent.

4.1. Example SMT Code

SSA Input:

x_1 := 2

if_cond_1 := x_1 < 4

y_1 := x_1 + 1

y_2 := x_1 - 1

y_3 := φ(if_cond_1, y_1, y_2)

assert := y_3 > 0

SMT Output:

(set-logic QF_NIA)

(declare-const x_1 Int)

(declare-const y_1 Int)

(declare-const y_2 Int)

(declare-const y_3 Int)

(declare-const if_cond_1 Bool)

(assert (= x_1 2))

(assert (= if_cond_1 (< x_1 4)))

(assert (= y_1 (+ x_1 1)))

(assert (= y_2 (- x_1 1)))

(assert (ite if_cond_1 (= y_3 y_1) (= y_3 y_2)))

(assert (> y_3 0))

(check-sat)

(get-model)

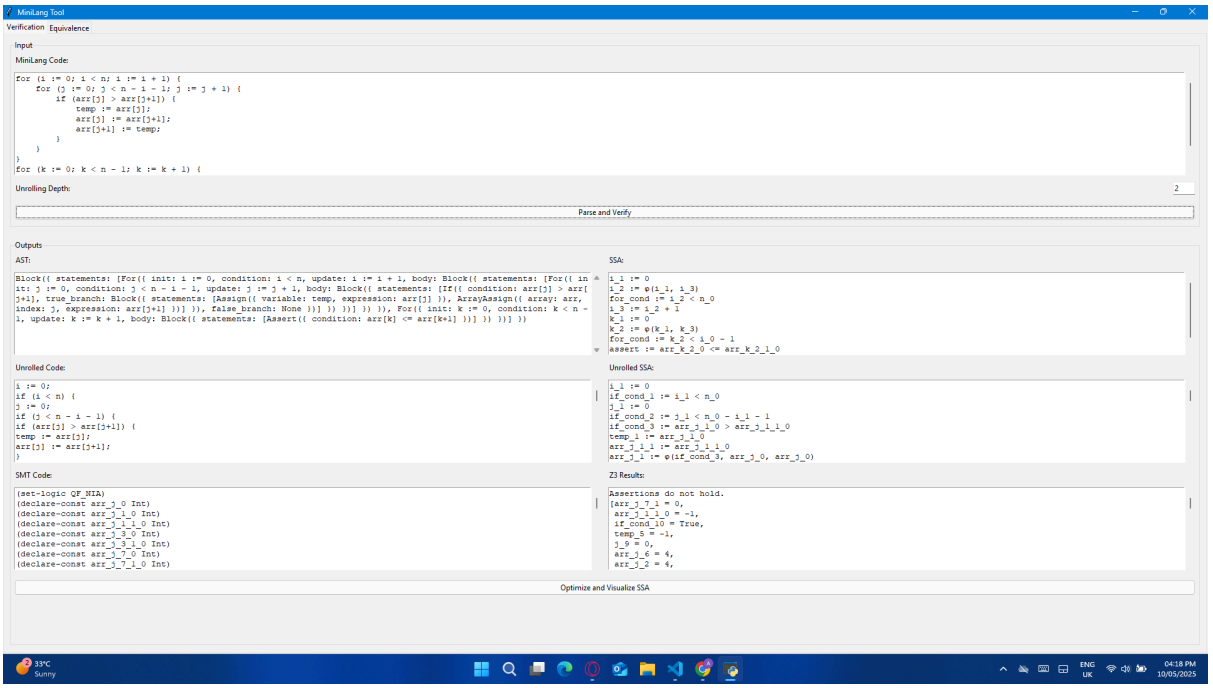
5. Screenshots of GUI and Test Results

5.1 GUI Overview

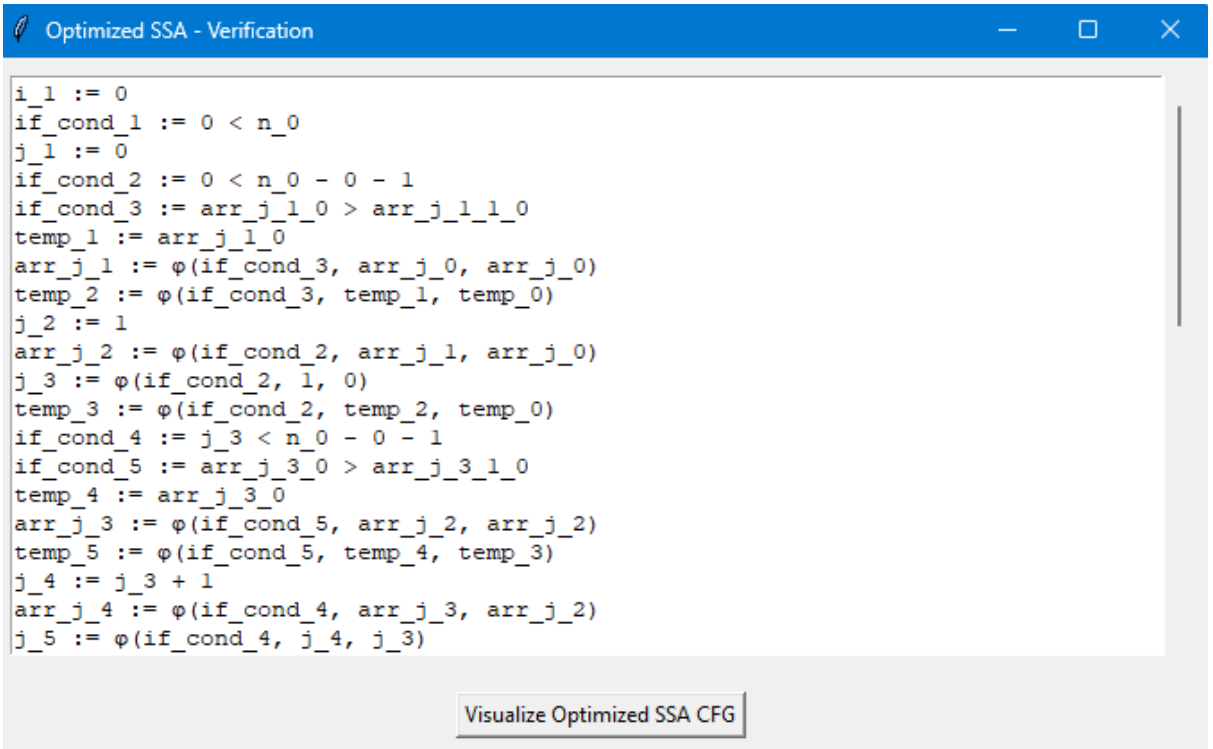
The GUI, built with Tkinter, provides an interactive interface for program analysis:

- **Verification Mode**:
  - **Layout**: A grid displays input code, AST (as a tree structure), SSA code, unrolled code, unrolled SSA, SMT code, and Z3 results.
  - **Features**: Users enter MiniLang code, set unrolling depth via a text field, and click “Verify” to process and display all steps.
- **Equivalence Mode**:
  - **Layout**: Two input panels for programs, with SSA, SMT, and equivalence results shown below. Users specify output variables in a separate field.
  - **Features**: “Check Equivalence” button triggers comparison, displaying whether programs are equivalent with supporting models or counterexamples.
- **CFG Visualization**:
  - Accessed via “Optimize and Visualize SSA” buttons, rendering control flow graphs using Graphviz integration.
  - Shows basic blocks and edges for both original and optimized SSA forms.

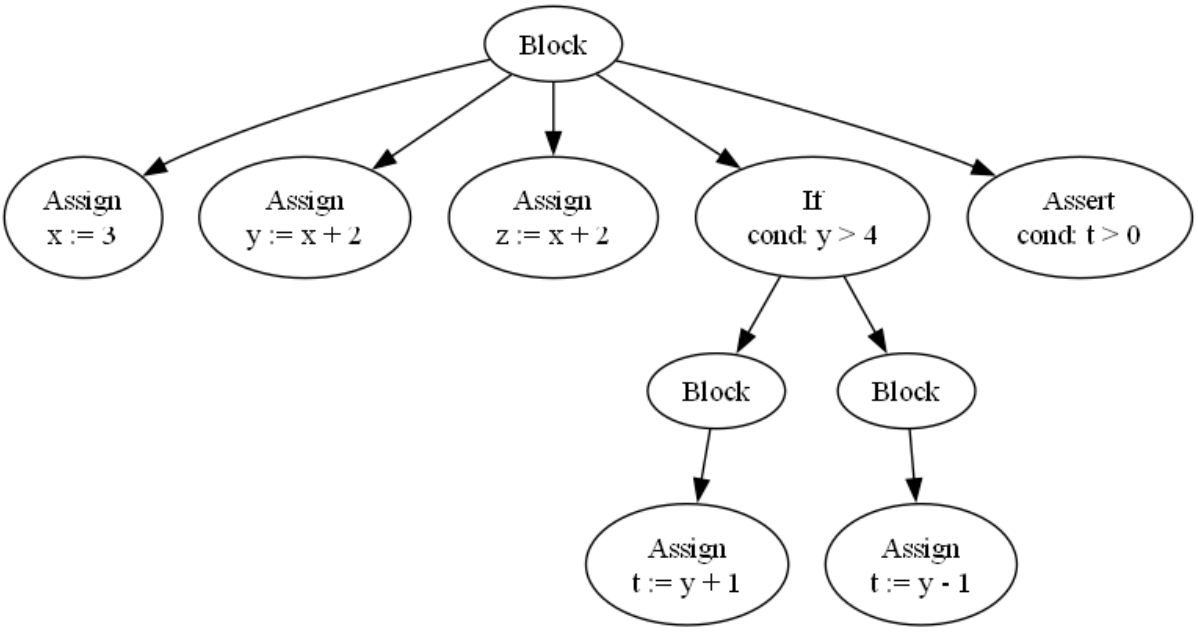
5.1.1. Verification Tab



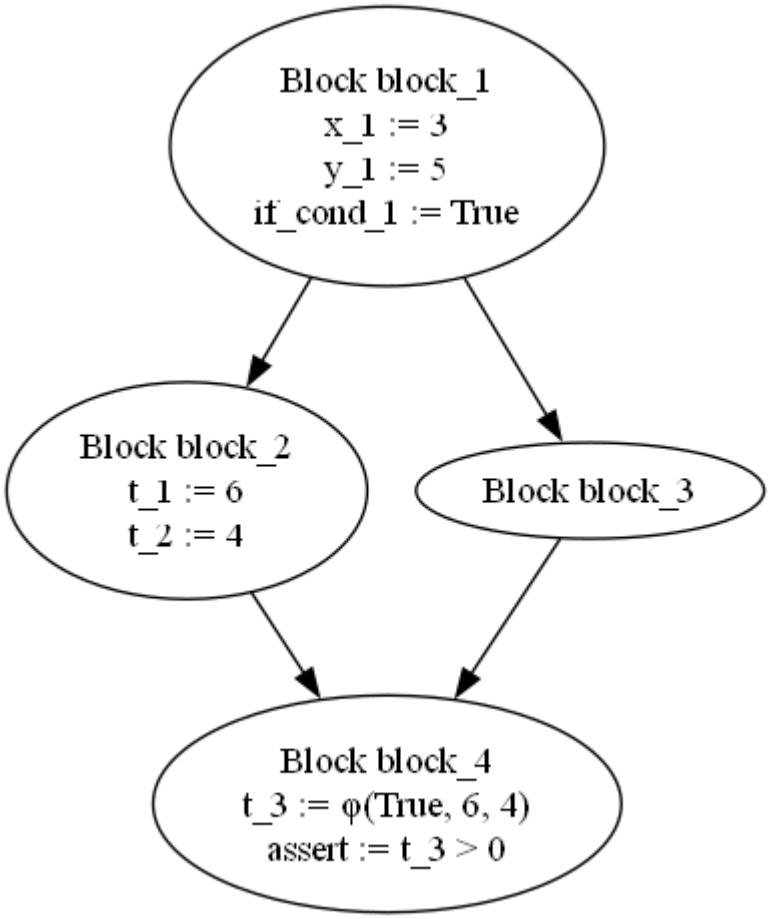
5.1.2. Optimized SSA - Verification



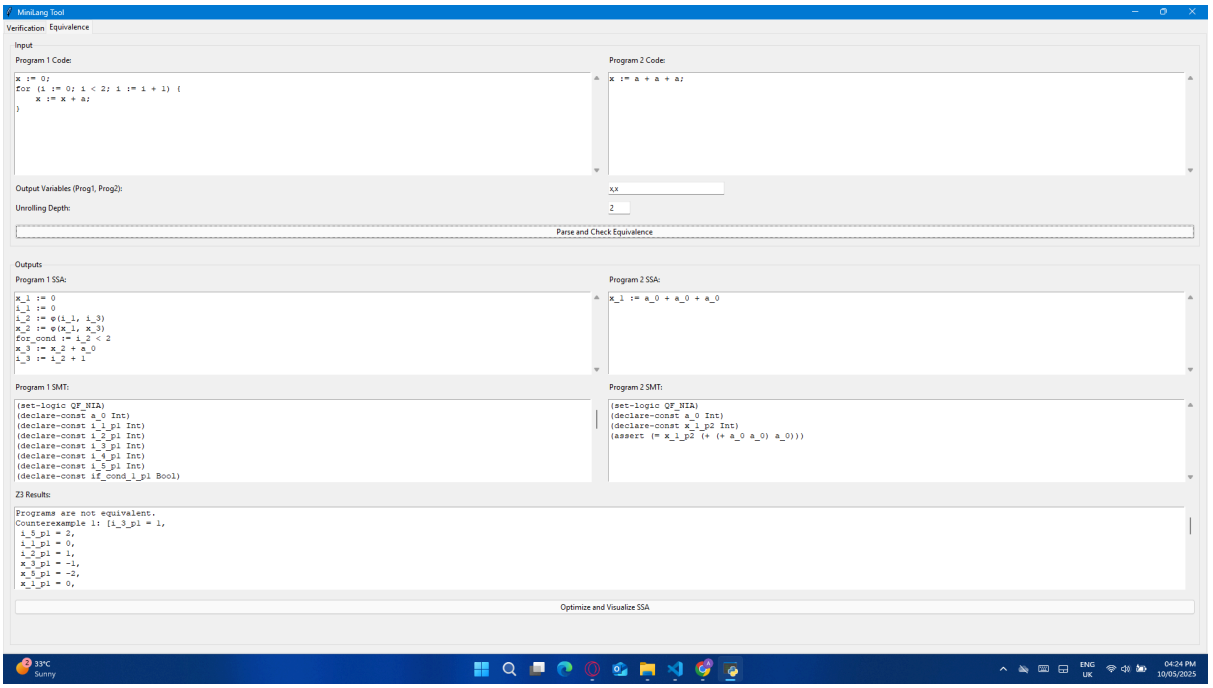
5.1.3. Original CFG



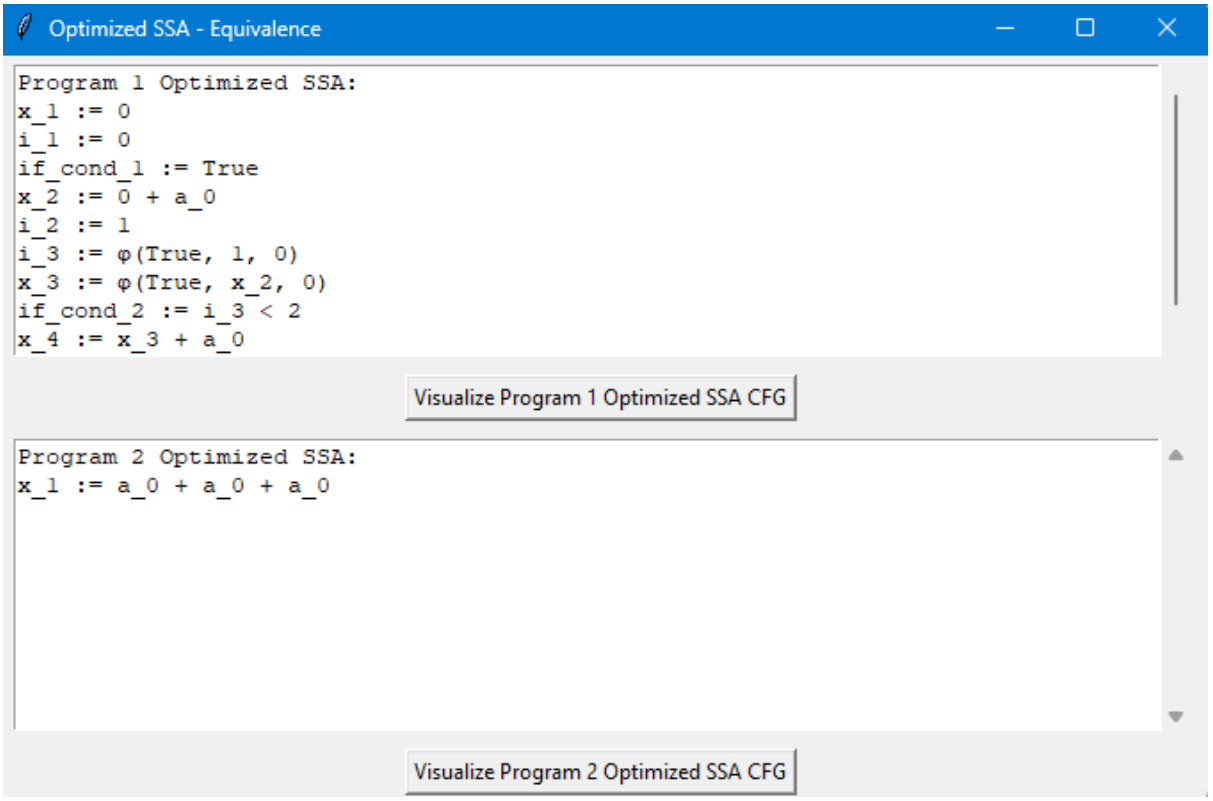
5.1.4. Optimized SSA CFG



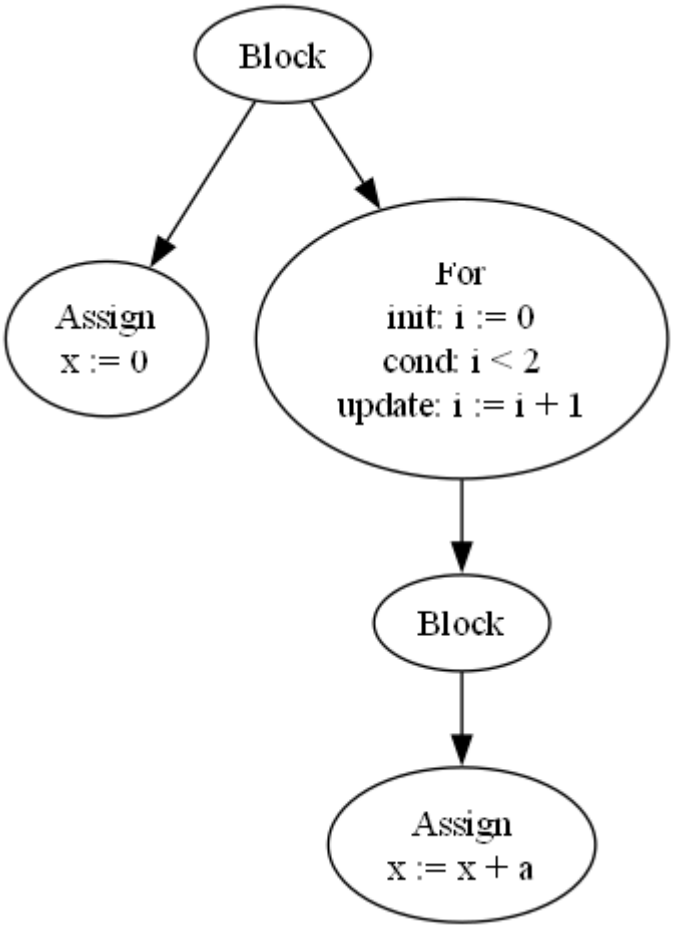
5.1.5. Equivalence Tab



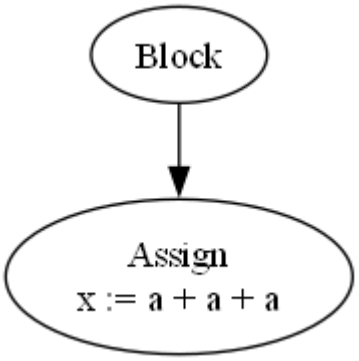
5.1.6. Optimized SSA - Equivalence



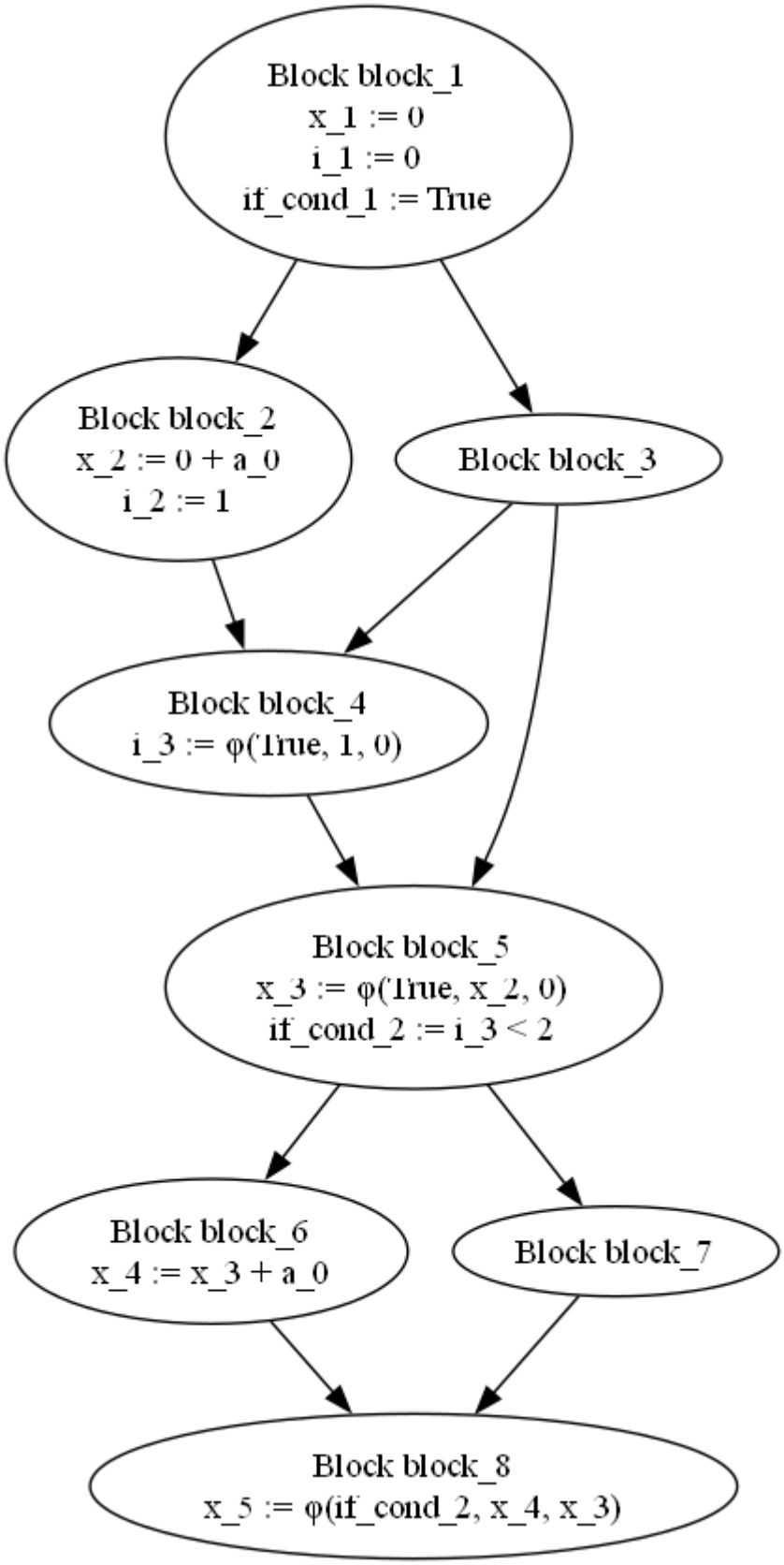
5.1.7. Original CFG - Program 1



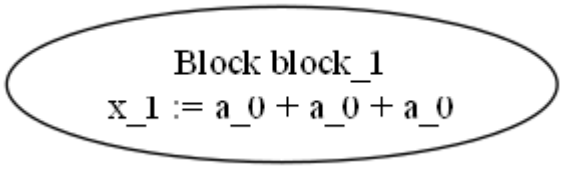
5.1.8. Original CFG - Program 2



5.1.9. Optimized SSA CFG - Program 1



5.1.10. Optimized SSA CFG - Program 2



5.2 Test Results

Test cases from example files validate the tool's functionality:

- **Verification1.txt**: A simple loop incrementing a variable. Result: Assertions hold, model [t_3 = 6, n_1 = 5, …].
- **Verification2.txt**: Array initialization with depth 2 unrolling. Result: Assertions hold, model [a_9 = 3, n_1 = 2, …].
- **Verification3.txt**: Faulty logic causing assertion failure. Result: Counterexamples [y_3 = 9] and [y_3 = 10].
- **Equivalence1_pair.txt**: Two programs with differing outputs. Result: Not equivalent, counterexamples [x_5 = 2] vs. [x_1 = 3].
- **Equivalence2_pair.txt**: Identical logic in different forms. Result: Equivalent, model [e_1 = 3, r_1 = 3].

These outcomes showcase the tool's ability to detect correctness and equivalence issues effectively.

6. Limitations and Improvements

6.1 Current Limitations

- **Array Handling**:
    - Limited to single-dimensional arrays with basic indexing.
    - Versioning (e.g., arr_j_1) oversimplifies dependencies, missing potential aliasing or out-of-bounds errors.
    - SMT translation lacks full array theory support, causing errors in complex array operations.

- **Loop Unrolling**:
    - Fixed-depth unrolling limits verification to the specified iterations, potentially missing deeper bugs.
    - No support for partial unrolling with invariants for remaining iterations.
- **Optimizations**:
    - Basic optimizations (constant propagation, dead code elimination) are implemented but rudimentary.
    - Advanced techniques like loop-invariant code motion or strength reduction are absent.
    - Optimizer falters with nested loops or intricate control flows.
- **GUI**:
    - CFG visualization is static, lacking interactivity (e.g., zoom, node details).
    - Output panels are cluttered, with minimal formatting or highlighting for readability.
    - No export functionality for intermediate results.

6.2 Proposed Improvements

- **Enhanced Array Support**:
    - Extend parser and SSA converter to handle multi-dimensional arrays and dynamic indices (e.g., arr[i][j]).
    - Integrate SMT array theories (e.g., (declare-fun arr ((Int) Int))) for robust verification of array properties.
- **Dynamic Loop Analysis**:
    - Implement bounded model checking or symbolic execution to analyze loops beyond fixed depths.
    - Add invariant inference to verify properties across all iterations symbolically.
- **Advanced Optimizations**:
    - Incorporate loop-invariant code motion to hoist invariant computations outside loops.
    - Enhance dead code elimination with data-flow analysis for complex cases.
    - Add common subexpression elimination and algebraic simplifications (e.g., $x + 0 \rightarrow x$).
- **GUI Enhancements**:

- Upgrade CFG visualization to an interactive canvas with tooltips and zoom capabilities.
- Introduce syntax highlighting in code input fields and collapsible sections for output panels.
- Enable exporting SSA, SMT, and Z3 results to files (e.g., .txt, .smt2) for external use.

Conclusion

The MiniLang Tool effectively verifies program correctness and equivalence through a pipeline of parsing, SSA transformation, loop unrolling, and SMT-based analysis, all accessible via a user-friendly GUI. While it excels with simple programs, its limitations in array handling, loop analysis, and optimizations highlight areas for growth. The proposed improvements aim to elevate its capability, making it a more versatile tool for formal verification tasks.