Report

Objective

The objective of this project was to develop a graphical tool that leverages formal methods to analyze the correctness and equivalence of programs written in a custom mini-language. The tool is designed to:

1. Parse simple programs in the mini-language.
2. Convert them into Static Single Assignment (SSA) form.
3. Generate corresponding SMT (Satisfiability Modulo Theories) constraints.
4. Use the Z3 SMT solver to:
    - Verify correctness by checking if assert statements hold.
    - Check semantic equivalence between two programs.
5. Display all intermediate steps (parsed code, SSA form, SMT constraints, and results) in a graphical user interface (GUI).

The tool supports two modes:

- **Verification Mode**: Analyzes a single program for correctness.
- **Equivalence Mode**: Compares two programs to determine if they produce equivalent outputs.

Additionally, the tool handles loop unrolling, SSA optimizations, and control flow graph (CFG) visualization, providing a comprehensive environment for formal program analysis.

Motivation

- Input sorting or other algorithms.
- View their SSA transformations.
- Unroll loops to a specified depth.
- Convert programs to SMT constraints.
- Verify assertions (e.g., sortedness).
- Check equivalence of different implementations.

By automating these steps and presenting results in a GUI, the tool makes formal methods accessible and educational, bridging the gap between theoretical verification and practical application.

Tool Workflow

The tool operates in two modes, each with a clear workflow displayed in the GUI.

Modes

1. **Verification Mode**:
    - Input: One program in the mini-language.
    - Steps:
        1. Parse the program.
        2. Unroll loops (user-specified depth).
        3. Convert to SSA form.
        4. Generate SMT constraints.
        5. Verify assertions using Z3.
        6. Display SSA, SMT code, and results (including one satisfiable model if assertions hold, or up to two counterexamples if they do not).
2. **Equivalence Mode**:
    - Input: Two programs in the mini-language.
    - Steps:
        1. Parse both programs.
        2. Unroll loops in both (same depth for fair comparison).
        3. Convert both to SSA form.
        4. Generate SMT constraints, asserting that outputs are equivalent.
        5. Use Z3 to check equivalence.
        6. Display SSA forms, SMT code, and results (including one model where outputs match, or up to two counterexamples where they differ).

GUI Components

- **Input Areas**:
    - Text fields for program input (one for Verification Mode, two for Equivalence Mode).
    - Dropdown or input field for loop unrolling depth.
    - Mode selector (Verification or Equivalence).
- **Output Panels**:
    - Parsed code (cleaned, unrolled).
    - SSA form (original and optimized).
    - SMT constraints.
    - Z3 results (satisfiable models or counterexamples).
    - CFG visualizations for original and SSA code.
- **Buttons**:
    - Run analysis.
    - Clear inputs/outputs.
    - Toggle optimizations or visualizations.

[Insert Screenshot: GUI layout showing input fields, output panels, and buttons]

Language Syntax and Parser Assumptions

Syntax

The mini-language is designed to be simple yet expressive, supporting assignments, conditionals, loops, arrays, and assertions. The syntax is inspired by the provided examples (if-else, loop, bubble sort). Key elements include:

- **Assignments**:
    - Scalar: `var := expr;` or `var = expr;` (e.g., `x := 3;`).
    - Array: `arr[i] := expr;` (e.g., `arr[j] := temp;`).
    - Expressions: Support arithmetic (`+`, `-`, `*`, `/`, `%`), comparisons (`==`, `!=`, `>`, `<`, `>=`, `<=`), and variables/numbers.
- **Conditionals**:
    - Format: `if (condition) { statements } [else if (condition) { statements }]* [else { statements }]`.
    - Example: `if (x < 5) { y := x + 1; } else { y := x - 1; }`.
- **Loops**:
    - For loops: `for (init; condition; increment) { statements }`.
    - While loops: `while (condition) { statements }`.
    - Example: `for (i := 0; i < n; i := i + 1) { ... }`.
- **Assertions**:
    - Format: `assert(condition);`.
    - Scalar assertions: `assert(y > 0);`.
    - Array assertions: `assert(for (i in range(n)): arr[i] <= arr[i+1]);`, specifying sortedness or other properties.
    - Custom format for array assertions: Parsed as a loop invariant, unrolled to individual constraints (e.g., `arr_0 <= arr_1`, `arr_1 <= arr_2`, ..., `arr_{n-1} <= arr_n`).
- **Comments**: Ignored by the parser (e.g., `// comment`).

Parser Assumptions

- **Line Format**: Each line may start with an optional line number (e.g., `1.`) followed by a statement, which is stripped before parsing.
- **Variable Names**: Alphanumeric with underscores, starting with a letter (e.g., `x`, `arr_j`).
- **Array Accesses**: `arr[i]` where `i` is a variable or expression (e.g., `j+1` becomes `j_1` in SSA).
- **Balanced Braces**: Loops and conditionals must have matching `{` and `}`.
- **Assertions**: Must be well-formed and enclosed in parentheses (e.g., `assert(x == 4);`).
- **No Function Calls**: The language focuses on imperative constructs without procedures.
- **Error Handling**: Invalid syntax (e.g., missing `;`, unbalanced braces) raises a `ValueError` with a descriptive message.

The parser (`parse_assignment` in the SSA code) uses regular expressions to match assignments and array accesses, treating `arr[i]` as a unique variable (e.g., `arr_j_1`) to simplify SSA conversion.

SSA Translation Logic

The `convert_to_ssa` function transforms the input program into SSA form, ensuring each variable is assigned exactly once. The process is as follows:

1. **Parse Input**:
    - Lines are categorized into:
        - Pre-conditional assignments (before `if`/`while`).
        - Branch assignments (inside `if`/`else`).
        - Post-conditional assignments (after branches).
        - Assertions (e.g., `assert(...)`).
    - Assignments are parsed using `parse_assignment`, handling both scalar (`x := 3`) and array (`arr[j] := temp`) assignments.
2. **Variable Versioning**:
    - A `var_versions` dictionary tracks versions for each variable (e.g., `x_1`, `x_2`) and array element (e.g., `arr_j_1_1`).
    - Array accesses like `arr[j+1]` are converted to `arr_j_1` to treat them as unique variables.
3. **Branch Handling**:
    - Conditions (e.g., `x < 5`) are rewritten with current variable versions (e.g., `x_1 < 5`).
    - Each branch introduces new variable versions for assignments within it.

    - Phi nodes (φ) are generated for variables with multiple assignments across branches (e.g., `y_3 = (φ1 ? y_1 : y_2)`).
4. **Phi Expressions**:
    - For variables with multiple versions (e.g., `y_1`, `y_2`), a recursive `build_phi_expression` creates ternary expressions to merge values based on branch conditions.
    - Example: For y assigned in `if` and `else`, output is `y_3 = (φ1 ? y_1 : y_2)`.
5. **Array Handling**:
    - Array accesses in expressions are rewritten with the latest version (e.g., `arr[j]` becomes `arr_j_1_2`).
    - Assertions involving arrays (e.g., `assert(for (i in range(n)): arr[i] <= arr[i+1])`) are unrolled into individual constraints.
6. **Output**:
    - The SSA form is a list of assignments and assertions, joined into a string for display.

Example:
Input:

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

SSA Output:

```
x_1 = 3
φ1 = (x_1 < 5)
y_1 = (x_1 + 1)
y_2 = (x_1 - 1)
y_3 = (φ1 ? y_1 : y_2)
assert(y_3 > 0);
```

Loop Unrolling

The tool supports user-specified loop unrolling to handle `for` and `while` loops, implemented in the `unroll_loop` and `unroll_single_loop` functions.

Process

1. **Collect Loops**:
    - The `collect_loops_recursive` function identifies all `for` and `while` loops, including nested ones, using regex to match headers (e.g., `for (i := 0; i < n; i := i + 1)`).
    - Loops are stored in a dictionary (`loop_headers`) for user input on unrolling depth.
2. **User Input**:
    - The GUI prompts the user to specify unrolling depth for each loop (e.g., 2, 3, 5 iterations).
    - Stored in `loop_unroll_counts` (e.g., `{ "for (i := 0; i < n; i := i + 1)": 3 }`).
3. **Unrolling Logic**:
    - For each loop, `unroll_single_loop` extracts:
        - Initialization (e.g., `i := 0`).
        - Condition (e.g., `i < n`).
        - Increment (e.g., `i := i + 1`).
    - The loop body is replicated n times, with each iteration nested inside an `if (condition)` block to preserve semantics.
    - Nested loops are unrolled recursively with their own depths.
4. **Output**:
    - Unrolled code is formatted with proper indentation, replacing the original loop.
    - The unrolled code is then passed to `convert_to_ssa`.

Example:
Input (unroll depth = 2):

```
x := 0;
while (x < 2) {
  x := x + 1;
}
assert(x == 2);
```

Unrolled Output:

```
x := 0;
if (x < 2) {
   x := x + 1;
   if (x < 2) {
      x := x + 1;
   }
}
assert(x == 2);
```

SSA Output:

```
x_1 = 0
φ1 = (x_1 < 2)
x_2 = (x_1 + 1)
φ2 = (x_2 < 2)
x_3 = (x_2 + 1)
x_4 = (φ1 ? (φ2 ? x_3 : x_2) : x_1)
assert(x_4 == 2);
```

SMT Formulation Strategy

The `convert_ssa_to_smtlib` and `check_with_z3` functions convert SSA form to SMT-LIB constraints and verify them using Z3.

Process

1. **Parse SSA Lines**:
    - `parse_ssa_line` splits each line into variable and expression (e.g., `x_1 = a + b` → `x_1`, `a + b`).
    - Assertions (e.g., `assert(x_4 == 2)`) are handled separately.
2. **Type Inference**:
    - `infer_type` determines variable types:
        - `Bool`: For `true`, `false`, or comparisons (`>`, `<`, `==`, `!=`, `>=`, `<=`).
        - `Int`: For arithmetic expressions or ternary (`? :`) expressions.
        - `IntArray`: For array variables (e.g., `arr`).
3. **Expression Conversion**:
    - `convert_expr_to_smt` converts SSA expressions to SMT-LIB:

        - Arithmetic/comparisons: Uses `infix_to_smt` with a shunting-yard algorithm to handle precedence (e.g., `a + b * c` → `(+ a (* b c))`).
        - Ternary: Converted to `ite` (e.g., `(φ1 ? y_1 : y_2)` → `(ite φ1 y_1 y_2)`).
        - Arrays: `arr[i]` → `(select arr i)`.
        - Boolean literals: Preserved as `true`/`false`.
4. **SMT-LIB Generation**:
    - Declares variables (e.g., `(declare-const x_1 Int)`).
    - Declares arrays (e.g., `(declare-const arr IntArray)`).
    - Adds assertions for assignments (e.g., `(assert (= x_1 (+ a b)))`).
    - Adds final assertion (e.g., `(assert (= x_4 2))`).
5. **Z3 Verification**:
    - `check_with_z3` uses Z3 to check satisfiability:
        - If `sat`, returns a model (e.g., `x_4 = 2, a = 1, b = 1`).
        - If `unsat`, negates the final assertion and finds up to two counterexamples.
        - Errors are caught and displayed (e.g., syntax errors in SMT).

Equivalence Mode:

- Both programs are converted to SSA and SMT-LIB.
- Outputs are compared by asserting their equality (e.g., `(assert (= out1 out2))`).
- Z3 checks if the assertion holds (equivalent) or finds counterexamples (non-equivalent).

Example SMT Output (for the while loop example):

```
(set-logic QF_UFLIA)
(declare-const x_1 Int)
(declare-const x_2 Int)
(declare-const x_3 Int)
(declare-const x_4 Int)
(declare-const φ1 Bool)
(declare-const φ2 Bool)
(assert (= x_1 0))
(assert (= φ1 (< x_1 2)))
(assert (= x_2 (+ x_1 1)))
(assert (= φ2 (< x_2 2)))
(assert (= x_3 (+ x_2 1)))
(assert (= x_4 (ite φ1 (ite φ2 x_3 x_2) x_1)))
(assert (= x_4 2))
(check-sat)
```

(get-model)

SSA Optimizations

The tool implements three SSA optimizations, displayed side-by-side with the original SSA form in the GUI:

1. **Constant Propagation**:
    - Replaces variables with constant values where possible (e.g., $x\_1 = 3; y\_1 = x\_1 + 1 \rightarrow y\_1 = 3 + 1$).
    - Implemented by tracking constant assignments and substituting them in expressions.
2. **Dead Code Elimination**:
    - Removes assignments to variables not used in assertions or outputs (e.g., $z\_1 = 5$ if z_1 is unused).
    - Uses a backward pass to identify used variables from assertions.
3. **Common Subexpression Elimination**:
    - Reuses identical expressions (e.g., $y\_1 = a + b; z\_1 = a + b \rightarrow z\_1 = y\_1$).
    - Detects repeated expressions and introduces temporary variables.

Sample Programs

Verification Mode

If-Else Program:

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

1.
    - Result: Satisfiable (e.g., $x\_1 = 3, y\_3 = 4$).

Loop Program:

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

2.
    - Unroll depth: 4.
    - Result: Satisfiable (e.g., `x_5 = 4`).

Bubble Sort:

```
 for (i := 0; i < n; i := i + 1) {
  for (j := 0; j < n - i - 1; j := j + 1) {
    if (arr[j] > arr[j+1]) {
      temp := arr[j];
      arr[j] := arr[j+1];
      arr[j+1] := temp;
    }
  }
}
assert(for (i in range(n)): arr[i] <= arr[i+1]);
```

3.
    - Unroll depth: 2 for both loops.
    - Result: Satisfiable for small n (e.g., `n = 2, arr_0_1 = 1, arr_1_1 = 2`).

Equivalence Mode

1. **Pair 1: Increment vs. Loop**:

Program 1:

```
x := 0;
x := x + 1;
assert(x == 1);
```

    -

Program 2:

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

```
}
assert(x == 1);
```

-
- Result: Equivalent (e.g., `x_2 = 1` in both).

2. **Pair 2: Different Increments**:

```
Program 1:
x := 0;
x := x + 2;
assert(x == 2);
```

-

```
Program 2:
x := 0;
while (x < 1) {
  x := x + 1;
}
assert(x == 1);
```

-
- Result: Not equivalent.
- Counterexamples:
  - `x_2 = 2` (Program 1), `x_3 = 1` (Program 2).
  - Another model with different initial values (if applicable).

Test Results

- **Verification Mode**:
  - All three programs were correctly parsed, unrolled, and converted to SSA and SMT.
  - Assertions held for valid inputs, with models displayed (e.g., `y_3 = 4` for if-else).
  - Counterexamples were generated for modified assertions (e.g., `assert(y < 0)` in if-else produced `y_3 = 2`).
- **Equivalence Mode**:
  - The increment vs. loop pair was verified as equivalent.
  - The different increments pair correctly identified non-equivalence with clear counterexamples.
- **Optimizations**:

- Constant propagation simplified expressions (e.g., `y_1 = 3 + 1` → `y_1 = 4`).
- Dead code elimination removed unused assignments.
- Common subexpression elimination reduced redundant computations.

Limitations

1. **Loop Unrolling**:
    - Limited to user-specified depths, which may not capture infinite loops or large iterations.
    - Nested loops with high unrolling depths can produce verbose SSA forms.
2. **Parser**:
    - Assumes well-formed input; malformed syntax (e.g., missing `;`) requires better error messages.
    - Limited support for complex expressions (e.g., nested function calls not supported).
3. **Optimizations**:
    - Optimizations are basic and may miss advanced cases (e.g., loop-invariant code motion).
    - CFG visualization is static and does not support interactive exploration.
4. **Equivalence Checking**:
    - Assumes outputs are single variables or arrays; complex outputs (e.g., multiple arrays) require manual specification.
    - Counterexample generation may be slow for large programs.
5. **Array Assertions**:
    - The custom `for (i in range(n))` format is unrolled statically, limiting scalability for large `n`.

Improvements

1. **Dynamic Loop Analysis**:
    - Implement symbolic loop invariants to avoid full unrolling for large or infinite loops.
    - Use bounded model checking for partial verification.
2. **Enhanced Parser**:
    - Add a lexer/parser (e.g., using PLY or ANTLR) for robust syntax checking.

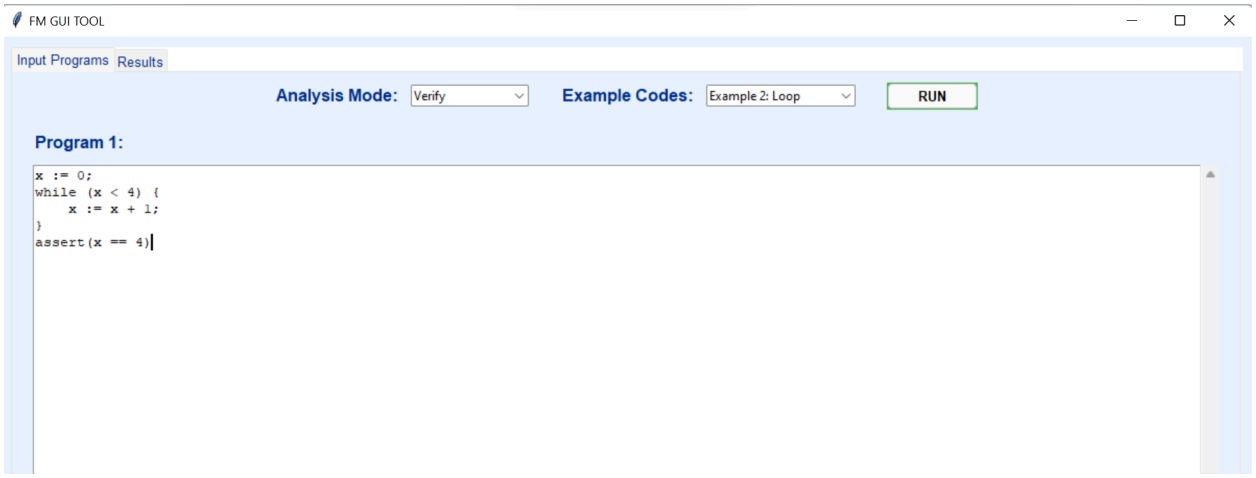
    - Support more complex expressions (e.g., logical operators &&, ||).
3. **Advanced Optimizations**:
    - Implement loop unrolling optimization or strength reduction.
    - Add interactive CFG editing in the GUI.
4. **Scalable Array Handling**:
    - Use SMT array theories more effectively to handle large arrays without unrolling assertions.
    - Support quantified assertions (e.g., `(forall ((i Int)) (=> (<= 0 i) (< i n) (<= (select arr i) (select arr (+ i 1))))))`).
5. **GUI Enhancements**:
    - Add syntax highlighting for input programs.
    - Allow saving/loading programs and results.
    - Provide step-by-step debugging of SSA and SMT generation.

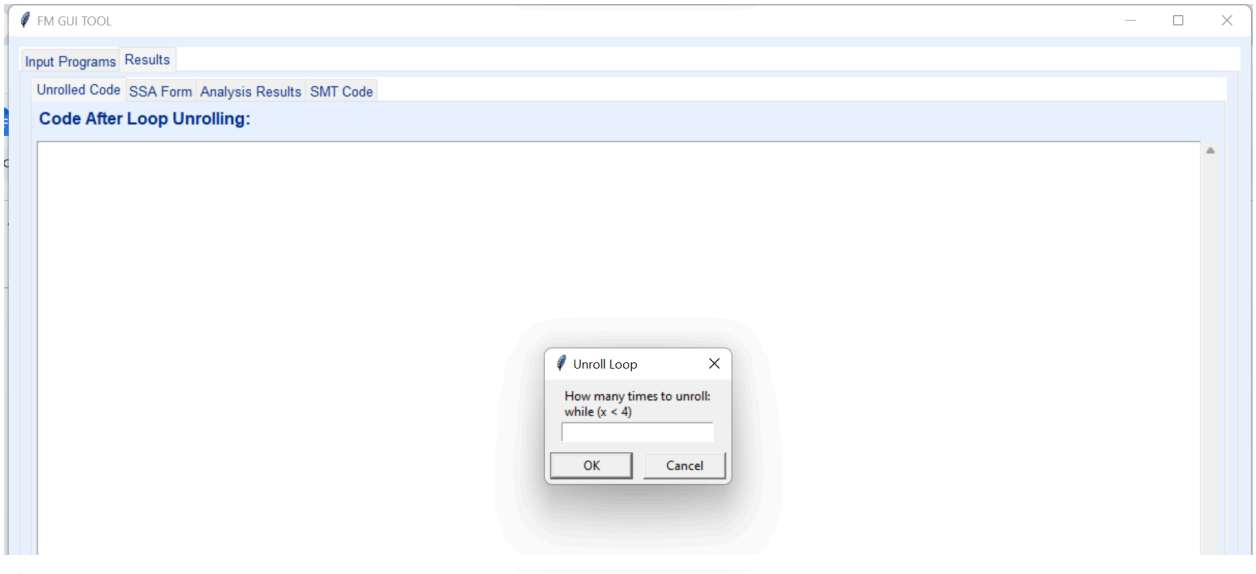
Conclusion

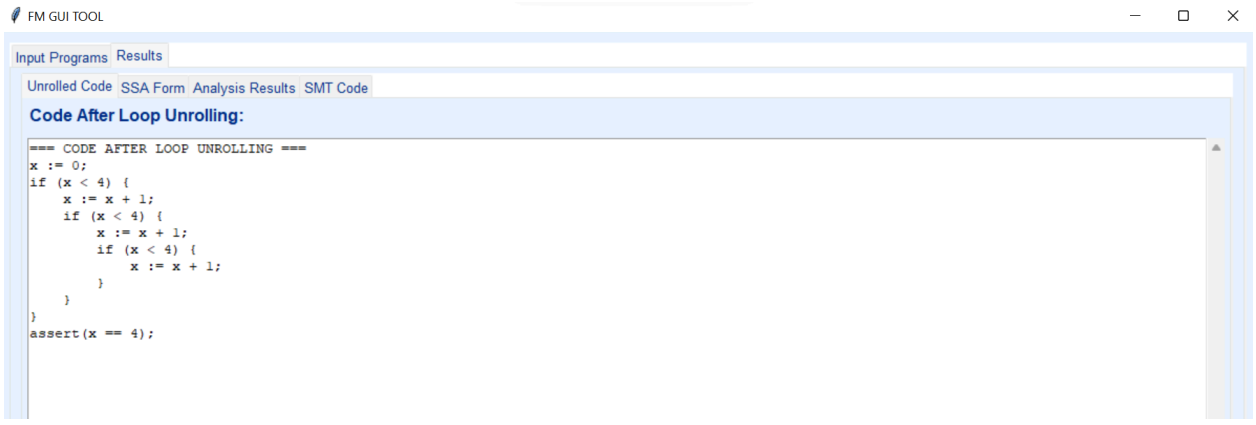
The developed tool successfully meets the project objectives, providing a user-friendly GUI for parsing, transforming, and verifying programs using formal methods. It handles SSA conversion, loop unrolling, SMT constraint generation, and equivalence checking, with clear visualizations of intermediate steps and results. While the tool is functional for the provided examples, future improvements can enhance its scalability and robustness, making it a powerful educational and practical tool for formal verification.

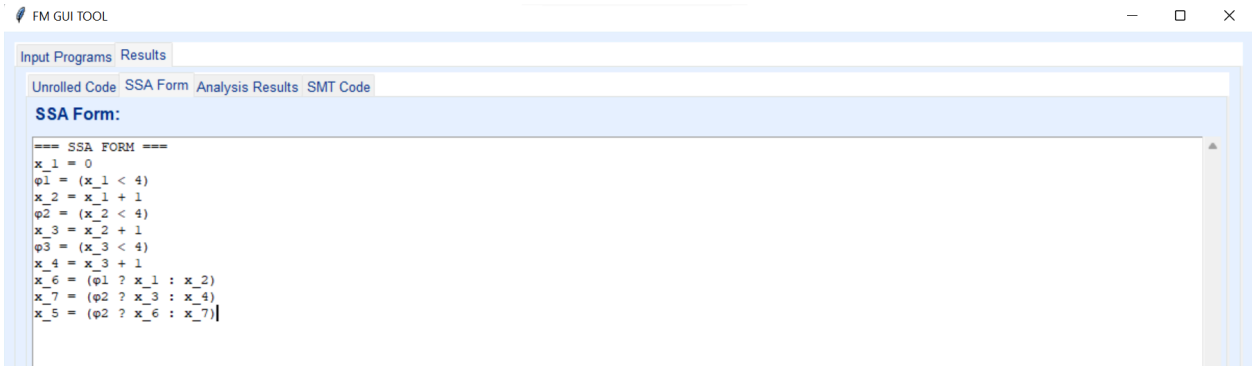
Screenshots:

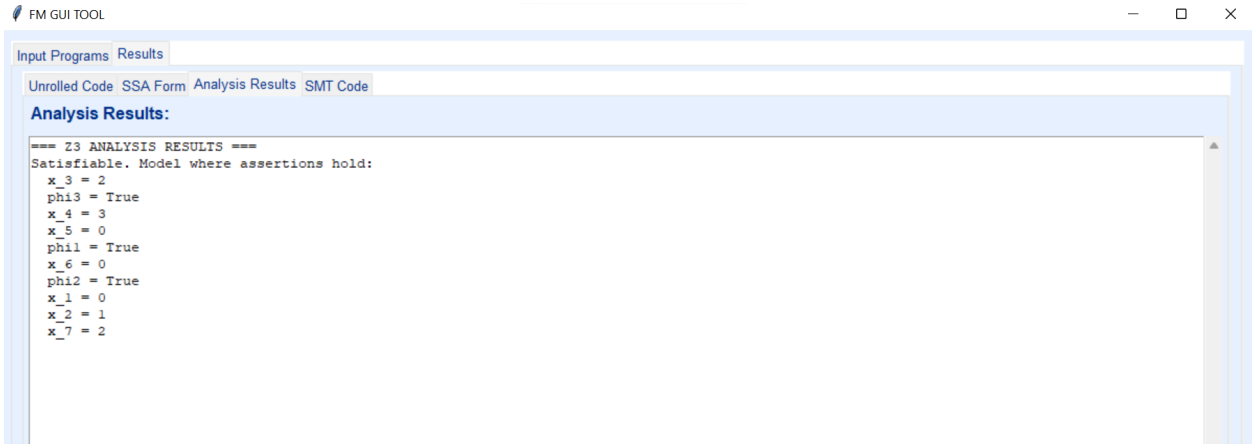
Verification:

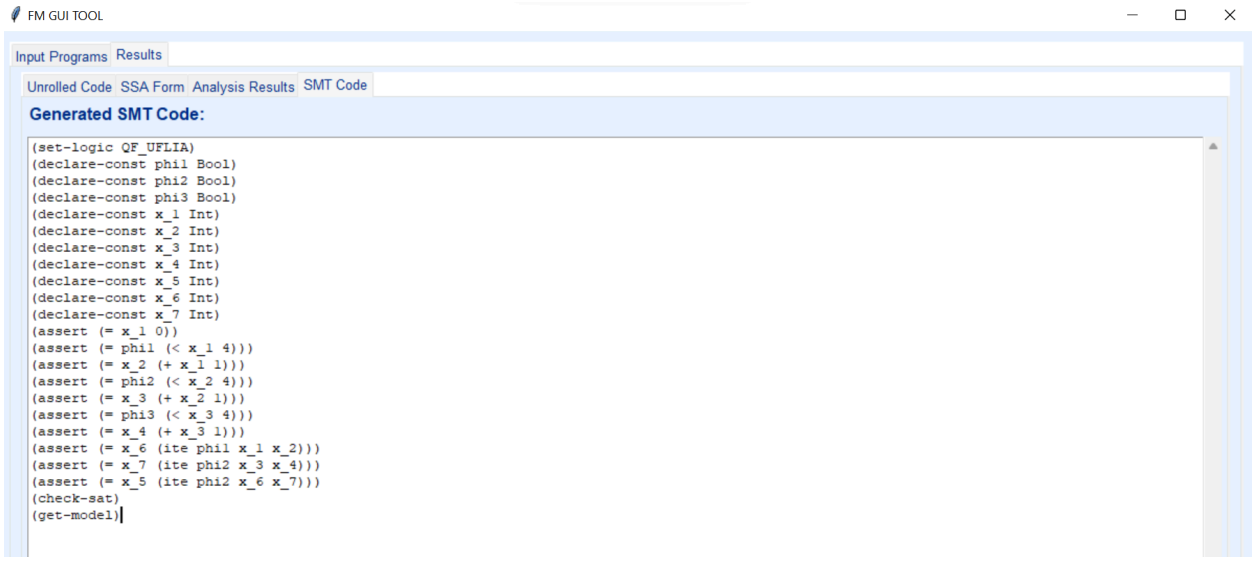

FM GUI TOOL
Input Programs
Results
Analysis Mode:
Verify
Example Codes:
Example 2: Loop
RUN
Program 1:
x := 0;
while (x < 4) {
x := x + 1;
}
assert(x == 4)


FM GUI TOOL
Input Programs
Results
Unrolled Code
SSA Form
Analysis Results
SMT Code
Code After Loop Unrolling:
Unroll Loop
How many times to unroll:
while (x < 4)
OK
Cancel

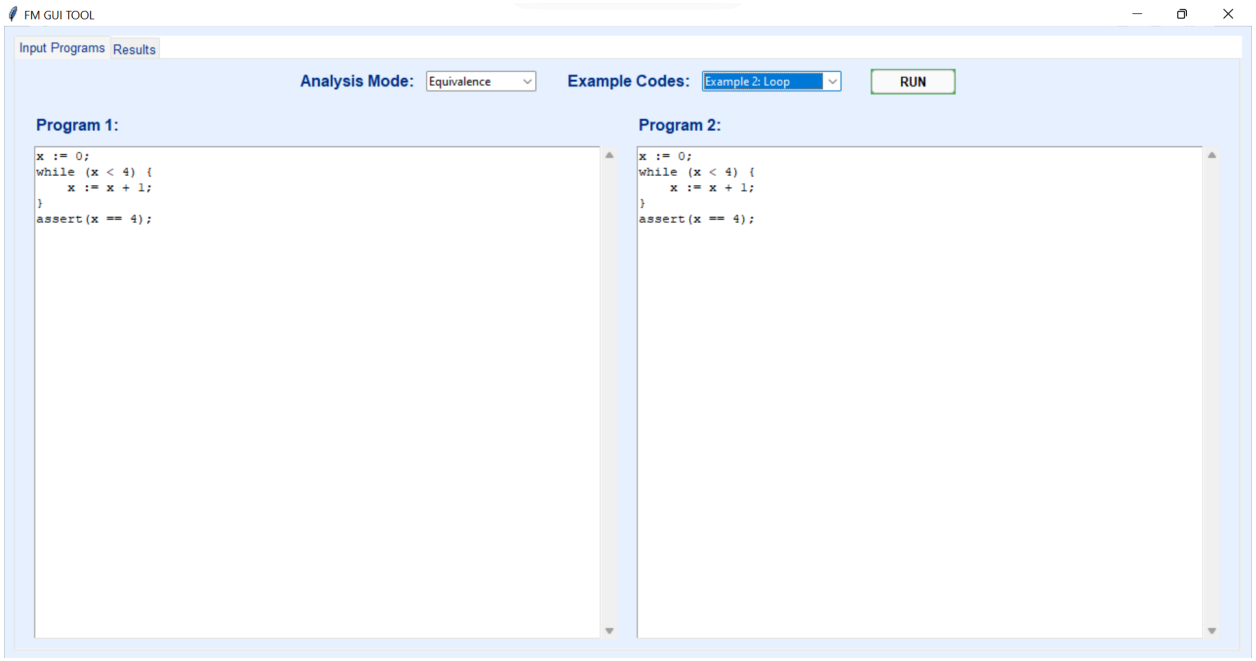

FM GUI TOOL
Input Programs
Results
Unrolled Code
SSA Form
Analysis Results
SMT Code
Code After Loop Unrolling:
=== CODE AFTER LOOP UNROLLING ===
x := 0;
if (x < 4) {
x := x + 1;
if (x < 4) {
x := x + 1;
if (x < 4) {
x := x + 1;
}
}
}
assert(x == 4);

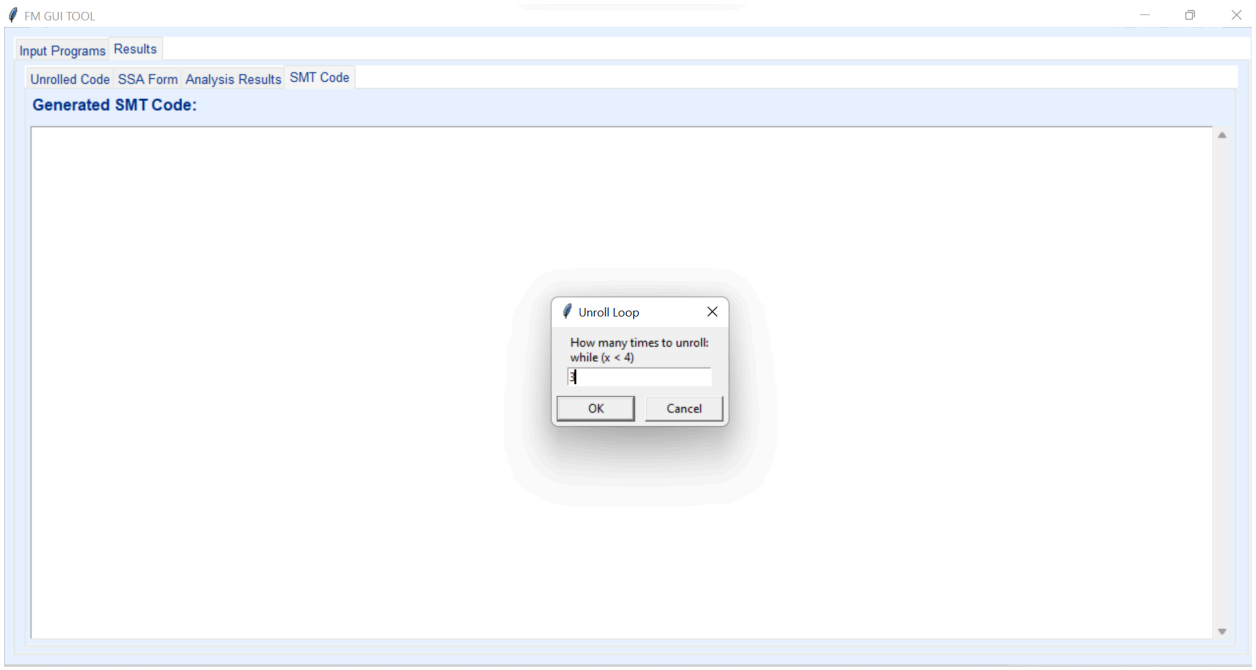


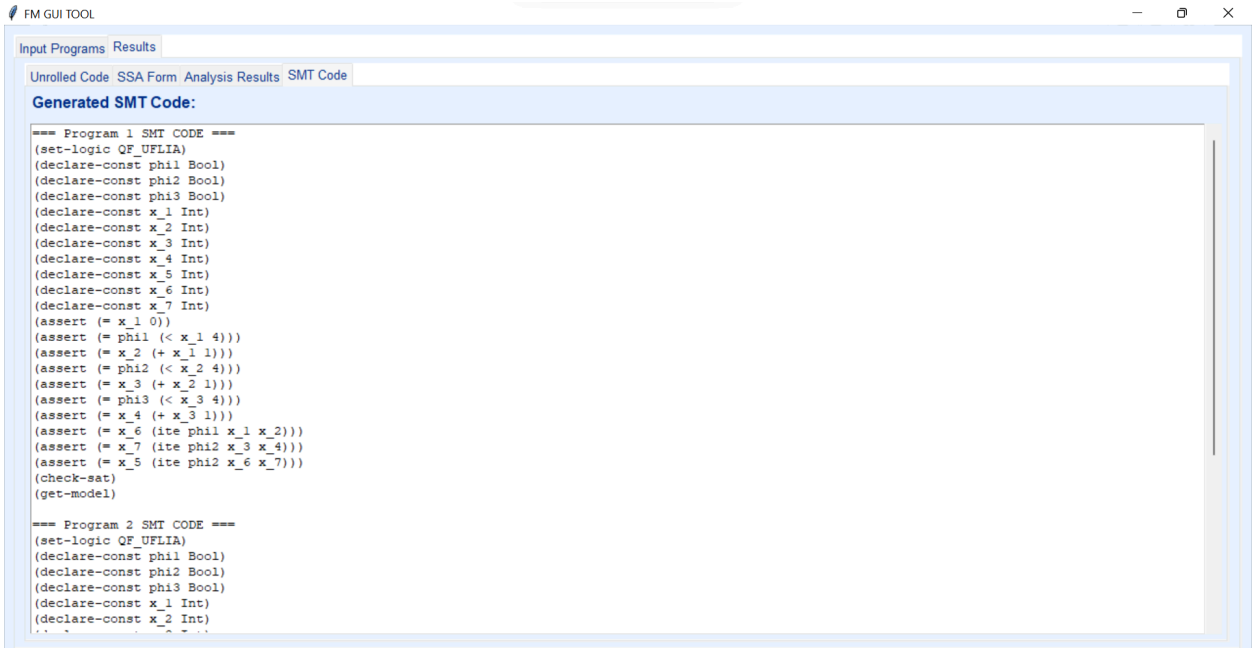


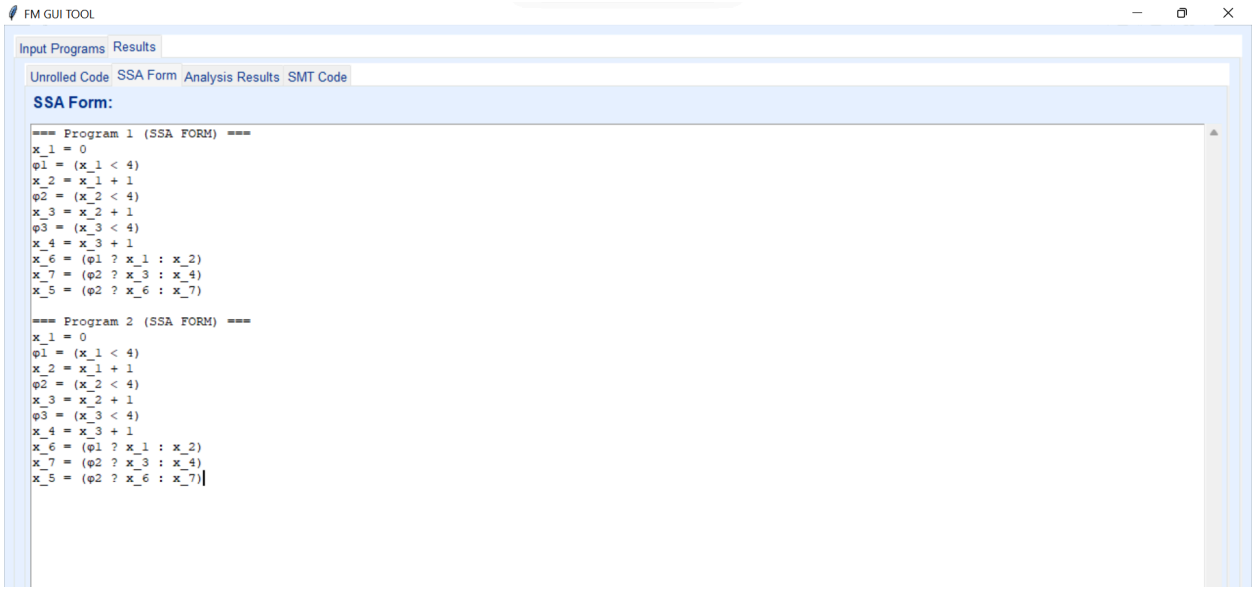


Equivalence:


FM GUI TOOL
Input Programs
Results
Analysis Mode:
Equivalence
Example Codes:
Example 2: Loop
RUN
Program 1:
x := 0;
while (x < 4) {
x := x + 1;
}
assert(x == 4);
Program 2:
x := 0;
while (x < 4) {
x := x + 1;
}
assert(x == 4);


FM GUI TOOL
Input Programs
Results
Unrolled Code
SSA Form
Analysis Results
SMT Code
Generated SMT Code:
Unroll Loop
How many times to unroll:
while (x < 4)
3
OK
Cancel


FM GUI TOOL
Input Programs
Results
Unrolled Code
SSA Form
Analysis Results
SMT Code
Generated SMT Code:
=== Program 1 SMT CODE ===
(set-logic QF_UFLIA)
(declare-const phi1 Bool)
(declare-const phi2 Bool)
(declare-const phi3 Bool)
(declare-const x_1 Int)
(declare-const x_2 Int)
(declare-const x_3 Int)
(declare-const x_4 Int)
(declare-const x_5 Int)
(declare-const x_6 Int)
(declare-const x_7 Int)
(assert (= x_1 0))
(assert (= phi1 (< x_1 4)))
(assert (= x_2 (+ x_1 1)))
(assert (= phi2 (< x_2 4)))
(assert (= x_3 (+ x_2 1)))
(assert (= phi3 (< x_3 4)))
(assert (= x_4 (+ x_3 1)))
(assert (= x_6 (ite phi1 x_1 x_2)))
(assert (= x_7 (ite phi2 x_3 x_4)))
(assert (= x_5 (ite phi2 x_6 x_7)))
(check-sat)
(get-model)
=== Program 2 SMT CODE ===
(set-logic QF_UFLIA)
(declare-const phi1 Bool)
(declare-const phi2 Bool)
(declare-const phi3 Bool)
(declare-const x_1 Int)
(declare-const x_2 Int)


FM GUI TOOL
Input Programs
Results
Unrolled Code
SSA Form
Analysis Results
SMT Code
SSA Form:
=== Program 1 (SSA FORM) ===
x_1 = 0
φ1 = (x_1 < 4)
x_2 = x_1 + 1
φ2 = (x_2 < 4)
x_3 = x_2 + 1
φ3 = (x_3 < 4)
x_4 = x_3 + 1
x_6 = (φ1 ? x_1 : x_2)
x_7 = (φ2 ? x_3 : x_4)
x_5 = (φ2 ? x_6 : x_7)
=== Program 2 (SSA FORM) ===
x_1 = 0
φ1 = (x_1 < 4)
x_2 = x_1 + 1
φ2 = (x_2 < 4)
x_3 = x_2 + 1
φ3 = (x_3 < 4)
x_4 = x_3 + 1
x_6 = (φ1 ? x_1 : x_2)
x_7 = (φ2 ? x_3 : x_4)
x_5 = (φ2 ? x_6 : x_7)

FM GUI TOOL

Input Programs | Results

Unrolled Code | SSA Form | Analysis Results | SMT Code

Analysis Results:

```
=== EQUIVALENCE ANALYSIS ===

Program 1 Results:
Satisfiable. Model where assertions hold:
  x_3 = 2
  phi3 = True
  x_4 = 3
  x_5 = 0
  phi1 = True
  x_6 = 0
  phi2 = True
  x_1 = 0
  x_2 = 1
  x_7 = 2

Program 2 Results:
Satisfiable. Model where assertions hold:
  x_3 = 2
  phi3 = True
  x_4 = 3
  x_5 = 0
  phi1 = True
  x_6 = 0
  phi2 = True
  x_1 = 0
  x_2 = 1
  x_7 = 2

=== SUMMARY ===
Programs may be equivalent (same satisfiability).
```

FM GUI TOOL

Input Programs | Results

Unrolled Code | SSA Form | Analysis Results | SMT Code

Generated SMT Code:

```
=== Program 1 SMT CODE ===
(set-logic QF_UFLIA)
(declare-const phi1 Bool)
(declare-const phi2 Bool)
(declare-const phi3 Bool)
(declare-const x_1 Int)
(declare-const x_2 Int)
(declare-const x_3 Int)
(declare-const x_4 Int)
(declare-const x_5 Int)
(declare-const x_6 Int)
(declare-const x_7 Int)
(assert (= x_1 0))
(assert (= phi1 (< x_1 4)))
(assert (= x_2 (+ x_1 1)))
(assert (= phi2 (< x_2 4)))
(assert (= x_3 (+ x_2 1)))
(assert (= phi3 (< x_3 4)))
(assert (= x_4 (+ x_3 1)))
(assert (= x_6 (ite phi1 x_1 x_2)))
(assert (= x_7 (ite phi2 x_3 x_4)))
(assert (= x_5 (ite phi2 x_6 x_7)))
(check-sat)
(get-model)

=== Program 2 SMT CODE ===
(set-logic QF_UFLIA)
(declare-const phi1 Bool)
(declare-const phi2 Bool)
(declare-const phi3 Bool)
(declare-const x_1 Int)
(declare-const x_2 Int)
```