# Technical Report: Program Equivalence Checker

May 2025

**Abstract**

This report provides a comprehensive analysis of the Program Equivalence Checker, a formal methods tool designed to verify the equivalence of programs using Static Single Assignment (SSA) form and Satisfiability Modulo Theories (SMT) solving. The tool supports a custom programming language, MiniLang, and includes a graphical user interface (GUI) for program analysis and equivalence checking. We discuss the language syntax, SSA translation logic, loop unrolling mechanism, SMT formulation strategy, GUI functionality, and test results. Additionally, we outline limitations and propose improvements for future development.

## 1 Introduction

The Program Equivalence Checker is a formal verification tool that determines whether two programs exhibit equivalent behavior under all possible inputs. It leverages a custom programming language, MiniLang, and employs formal methods techniques, including parsing, SSA transformation, loop unrolling, and SMT solving with the Z3 solver. The tool features a GUI for interactive program analysis and equivalence checking, making it accessible to users with varying levels of expertise in formal methods.

This report is structured as follows:

- Section 2 details the MiniLang syntax and parser assumptions.
- Section 3 explains the SSA translation logic.
- Section 4 describes the loop unrolling mechanism.
- Section 5 outlines the SMT formulation strategy.
- Section 6 presents the GUI and test results.
- Section 7 discusses limitations and potential improvements.

# 2 Language Syntax and Parser Assumptions

## 2.1 MiniLang Syntax

MiniLang is a simple imperative programming language designed for formal verification. Its syntax, defined in `mini_lang.lark`, supports the following constructs:

- **Assignments**: `VAR := expr;`, where `VAR` is an identifier and `expr` is an arithmetic expression.
- **If Statements**: `if (condition) { statements } [else { statements }]`, with conditions comparing expressions using `<`, `>`, `<=`, `>=`, `==`, `!=`.
- **While Loops**: `while (condition) { statements }`.
- **For Loops**: `for (assignment; condition; assignment) { statements }`.
- **Assertions**: `assert(condition);`, specifying postconditions.
- **Blocks**: `{ statements }`, grouping multiple statements.

Arithmetic expressions support addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). Variables are identifiers starting with a letter or underscore, followed by alphanumeric characters or underscores. Numbers are non-negative integers.

## 2.2 Array Support

The current implementation does not support arrays explicitly. The grammar in `mini_lang.lark` and the parser in `parser.py` assume scalar variables and expressions. Array operations, such as indexing or iteration, are not part of the syntax, limiting the tool's applicability to array-based programs. Assertions are restricted to conditions involving scalar variables and arithmetic expressions, e.g., `assert(x > 0);`.

## 2.3 Parser Assumptions

The parser, built using the Lark library, assumes:

- **Whitespace Ignorance**: Whitespace and single-line comments (`//`) are ignored.
- **Valid Input**: Programs must conform to the grammar; invalid syntax results in parse errors.
- **No Array Constructs**: The parser does not recognize array declarations or operations.
- **Postcondition Format**: Assertions must be boolean conditions, typically comparisons between expressions.

The `MiniLangTransformer` class in `parser.py` transforms the parse tree into an Abstract Syntax Tree (AST) with tuples representing statements and expressions, facilitating subsequent analysis.

# 3 SSA Translation Logic

The SSA conversion, implemented in `ssa_converter.py`, transforms the AST into Static Single Assignment form, where each variable is assigned exactly once. The `SSAConverter` class maintains:

- `counter`: Tracks variable versions (e.g., `{'x': 3}` for `x_3`).
- `env`: Maps variables to their current SSA versions (e.g., `{'x': 'x_3'}`).
- `ssa`: Stores the list of SSA statements.

## 3.1 Conversion Process

The `convert` method processes AST statements:

- **Assignments**: For `('assign', var, expr)`, a new version of `var` is created (e.g., `x_n`), and the expression is transformed to use current SSA variable versions. The statement `(new_var, '=', transformed_expr)` is added to `ssa`.
- **If Statements**: For `('if', cond, true_block, false_block)`, the condition is transformed, and a `('if', new_cond)` marker is added. True and false blocks are recursively processed for assignments.
- **Assertions**: For `('assert', cond)`, the condition is transformed, and `('assert', new_cond)` is added.

Expressions are transformed via `transform_expr`, which replaces variable references with their current SSA versions and preserves arithmetic and conditional operators.

## 3.2 Output Formatting

The `format_ssa_output` function converts SSA statements into readable text, e.g., `x_1 := 10` for assignments, `if x_1 > y_1` for conditions, and `assert(x_1 > 0)` for assertions.

## 3.3 Limitations

The SSA converter does not handle phi nodes for control flow merges, limiting its ability to model complex control flow accurately. It also lacks support for arrays, as variable versioning is applied only to scalars.

# 4 Loop Unrolling Mechanism

The loop unrolling mechanism, implemented in `loop_unroller.py`, transforms `for` and `while` loops into a sequence of conditional statements up to a specified bound.

## 4.1 Unrolling Process

The `unroll_loops` function processes the AST:

- **For Loops**: A `('for', init, cond, update, body)` statement is unrolled into:
  - The initialization statement.
  - `unroll_bound` iterations, each comprising an `if` statement checking `cond`, executing `body`, and applying `update`.
- **While Loops**: A `('while', cond, body)` statement is unrolled into `unroll_bound` `if` statements, each checking `cond` and executing `body`.
- **If Statements**: Then and else blocks are recursively unrolled if they contain loops.
- **Other Statements**: Preserved unchanged.

The default `unroll_bound` is 3, adjustable via the GUI.

## 4.2 Formatting

The `format_unrolled_code` function converts the unrolled AST back into MiniLang syntax, preserving indentation and structure. For example, a `for` loop unrolled three times generates three `if` blocks with the loop body and update statements.

## 4.3 Limitations

Unrolling is finite, potentially missing behaviors beyond the bound. The mechanism does not handle nested loops optimally, as each loop is unrolled independently, leading to exponential growth in code size.

# 5 SMT Formulation Strategy

The SMT formulation, implemented in `smt_generator.py`, converts SSA code into Z3 constraints to verify assertions and check program equivalence.

## 5.1 SMT Generation

The `SMTGenerator` class:

- Maintains a Z3 `Solver` and a dictionary of Z3 variables (`vars`).
- Converts SSA statements via `to_smt`:
  - **Assignments**: `(var, '=', rhs)` becomes `var == expr_to_z3(rhs)`.
  - **If Statements**: `('if', cond)` adds `Implies(path_condition, expr_to_z3(co`
  - **Assertions**: `('assert', cond)` adds `Implies(path_condition, expr_to_z3(`
- Transforms expressions using `expr_to_z3`, mapping MiniLang operators to Z3 equivalents (e.g., `add` to `+`).

## 5.2 Assertion Checking

The `check_assertions` method uses the Z3 solver to check satisfiability. If satisfiable, it returns a model with variable assignments; otherwise, it indicates an assertion violation.

## 5.3 Equivalence Checking

The `check_program_equivalence` function compares two SSA programs:

- Creates two `SMTGenerator` instances with distinct variable prefixes (`p1_`, `p2_`).
- Adds constraints from both programs to a new Z3 solver.
- Checks if common variables can have different values under any input.
- Verifies if assertions behave differently under any conditions.
- Returns detailed results, including counterexamples if non-equivalent.

## 5.4 Limitations

The SMT formulation assumes integer arithmetic, lacking support for real numbers or arrays. Phi nodes are handled simplistically, potentially leading to incomplete equivalence checks. The solver may face scalability issues with large programs or high unroll bounds.

# 6 GUI and Test Results

The GUI, implemented in `gui.py`, provides two tabs:

- **Program Analysis**: Allows input of a single program, displaying parse tree, AST, SSA, SMT verification, and unrolled code.
- **Equivalence Check**: Compares two programs, showing equivalence results.

## 6.1 Test Results

We tested the tool with the example programs in `parser.py`:

**Program 1:**

```
x := 10;
y := 5;
if (x > y) {
    z := x + y;
    assert(z > x);
}
y := z - 2;
assert(y >= 0);
```

**Program 2:**

```
x := 1;
y := x + 2;
if (y > 5) {
    z := y - 2;
}
assert(z < 0);
```

The equivalence check correctly identifies that the programs are not equivalent, providing a counterexample with variable assignments.

## 6.2 Screenshots

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

- **Analysis Tab**: Shows input program text, buttons for analysis and loop unrolling, and tabs for parse tree, AST, SSA, SMT, and unrolled code.
- **Equivalence Tab**: Displays two program inputs side-by-side, a "Check Equivalence" button, and an output area with detailed results.

# 7 Limitations and Improvements

## 7.1 Limitations

- **Array Support**: Lack of array handling restricts the tool to scalar-based programs.
- **Phi Nodes**: The SSA converter does not implement phi nodes, limiting control flow analysis.
- **Loop Unrolling**: Finite unrolling may miss behaviors in loops with more iterations.
- **Scalability**: Large programs or high unroll bounds may cause performance issues in Z3.

- **Real Numbers**: The SMT solver assumes integers, excluding floating-point arithmetic.

### 7.2 Proposed Improvements

- **Array Support**: Extend the grammar and parser to include array declarations and indexing, updating SSA and SMT logic accordingly.
- **Phi Nodes**: Implement proper phi node handling in SSA conversion to support control flow merges.
- **Symbolic Unrolling**: Use symbolic execution to handle loops without fixed bounds.
- **Performance Optimization**: Optimize Z3 constraint generation and explore parallel solving.
- **Floating-Point Support**: Add support for real numbers in the SMT formulation.
- **Enhanced GUI**: Include syntax highlighting, error annotations, and visualization of counterexamples.

## 8 Conclusion

The Program Equivalence Checker is a robust tool for verifying program equivalence using formal methods. It effectively handles MiniLang programs, providing detailed analysis through parsing, SSA conversion, loop unrolling, and SMT solving. While limitations such as lack of array support and finite loop unrolling exist, proposed improvements can enhance its functionality and applicability. The GUI makes the tool accessible, and test results demonstrate its correctness in identifying non-equivalent programs.