Implementing a Verification Tool for a Simple Programming Language

Formal Verification 2024 Final Report

Mohamad Ali Atwi Salya Diallo Taufiq Mohammed

I. Introduction

In modern software systems, correctness and reliability are critical, particularly in safety-critical domains such as healthcare, finance and autonomous systems. Formal verification is a powerful technique that ensures program correctness by mathematically proving that a program meets its specifications. These tools help identify bugs, confirm correctness, and validate specific program properties before deployment, minimizing costly errors, and improving software quality. Unlike traditional testing, which only examines specific inputs, formal verification provides guarantees over all possible program behaviors.

In this project, we explored the fundamentals of verification by creating our own toy programming language and developing a custom verification tool for proving the partial correctness of a program. This tool was designed to analyze programs written in the toy language, enabling us to experiment with key verification techniques and their practical applications.

II. Theoretical Background

Our tool is based on the verification process that is described in chapter 5 of The Calculus of Computation [1], most of what we will discuss in this section can be found in the chapter.

First-order logic (FOL) is a formal system that uses quantified variables on non-logical objects and allows the construction of sentences containing these variables to express properties, relationships, and logical reasoning about the objects. For example, the proposition "all men are mortal" can be expressed in FOL as: "$\forall x$, x is a man $\implies$ x is mortal".

FOL formulas are introduced into our programming language in the form of annotations as described in Section III. Each annotation is a FOL formula whose free variables are a subset of the variables visible inside the function, the formula should evaluate to true when reached during the program execution.

These annotations are essential in the process of proving the **partial correctness** of a program, a function is partially correct when its postcondition is satisfied for all inputs that satisfy the precondition.

The verification process used to prove partial correctness is depicted in figure 1, this process is named the **inductive assertion method**, it transforms the program's functions into a number of FOL formulas that can be checked for validity; in case all formulas are valid, the program is partially correct.

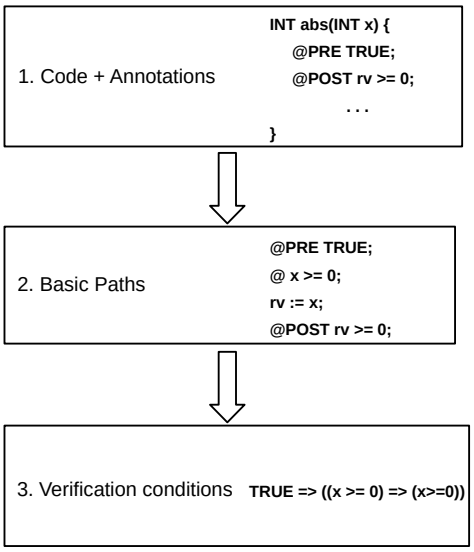

Fig. 1: General steps of the inductive assertion method.

In order to achieve this transformation, the inductive assertion method follows a two-step procedure:

1) The first step generates an intermediate form starting from the function code; in this form, the instructions inside the function are divided into a number of **basic paths** based on the control flow and the annotations used inside the function. Each basic path is a sequence of statements and assertions, starting and ending with an annotation.
2) We then use the intermediate form to simplify the process of generating the **verification conditions** (VCs), which are FOL formulas that need to be validated. This second step is achieved using the **weakest precondition predicate transformer** it is a function that computes the **weakest precondition** P', given a FOL formula and a statement.

$$wp : FOL \times Statement \rightarrow FOL$$

Given a statement S and a postcondition Q, the weakest precondition P is the least restrictive (weakest) FOL formula such that if P holds before executing S, then Q is guaranteed to hold afterward; this is formally expressed using the Hoare triple notation: {P}S{Q}. Because P is the weakest precondition, any other precondition such

that {P'}S{Q} must satisfy $P' \implies P$.

By combining these steps, the inductive assertion method systematically converts program functions into logical formulas, enabling their verification and ensuring partial correctness.

III. Toy Language

We introduce in this section the key properties of our **toy programming language (TPL)**, which is inspired by the language described in [1]. The simple features of TPL allow us to focus on implementing a basic verifying compiler from scratch, a task that would be significantly more challenging for an actual programming language. It is important to note that TPL is not a compilable language; it was specifically designed for this project and cannot be used to write actual programs or coding projects.

TPL is Turing-complete, imperative, and statically typed, with a syntax similar to C. Note that the language lacks function calls. The language supports four statement types: declaration, assignment, annotation, and control flow. Variables in TPL are restricted to two types: boolean and integer.

A. Functions

Function definitions are a fundamental feature since no statements can be written outside a function's body. A function definition has the following structure:

```
<Type> FUNCTION <function_name> (<Parameters>) {
    DECLARE (<Local_variables>);
    @PRE P;
    @POST Q;
    <body>
    RETURN R;
}
```

This syntax specifies the return type, function name, and parameter declarations in the header.

The function body begins with an optional DECLARE statement, where all local variables are declared. No further variable declarations can occur after this point. The two statements that follow are the function specifications: precondition (@PRE P) and postcondition (@POST Q), where P and Q are Boolean expressions representing FOL formulas. The function body then contains the rest of the code, concluding with one or more return statements, the returned expression R has to match the return type of the function.

B. Statements

1) Declaration: To declare a variable of Boolean or integer type, we use the keywords BOOL and INT. For instance, to declare a variable x we would write one of the following lines, for a Boolean or integer variable respectively:

```
BOOL x
INT x
```

The declaration of multiple variables as parameters or local variables must be separated by commas.

2) Assignment: To assign a certain value to a variable, we use the notation ":=" instead of the most commonly used "=". For example, to assign the value 3 to a variable x, we would write the following line inside the program:

$$x := 3$$

3) Annotation: Annotations assert that a formula F holds true at a specific program location L. This is a key feature of our language, as it facilitates the program validation process. There are four different types of annotation:

- *Precondition* (`@PRE`): Describes the input that the function expects;
- *Postcondition* (`@POST`): condition that should hold at the exit of the function;
- *Loop-invariant* (`@LOOP`): ensures that the condition holds when entering the while loop;
- *Assertion* (`@`): An annotation that is not a precondition, postcondition, or loop-invariant. It is used as a formal comment to assert that a certain condition holds at location L.

4) Control flow: Branching and looping are supported through if-then-else and while loop statements, respectively.

If-then-else syntax:

```
IF (<Condition>) {
    <Then-Body>
} ELSE {
    <Else-Body>
}
```

While loop syntax:

```
<Initialization>;
@LOOP F
WHILE (<Condition>) {
    <Body>
}
```

5) No operation statement: A statement with no effect; it can be used for an empty else body or while loop body.

```
NOP;
```

C. Expressions

Expressions in TPL fall into three categories:

- *Binary expression*: involving two operands and an operator;
- *Unary expression*: involving a single operand and an operator;

- *Parameter and variable reference*: Expressions that evaluate to the value of a variable. The special variable `rv` represents the return value of a function and can only be used in the postcondition.

The operators listed in Tables I, II and III are considered, each representing the different types of operators for the integer and boolean types. In each table, the first column shows the syntax of the operator in TPL, the second column explains its mathematical meaning, and the third column specifies the expression's arity and the type of its returned value upon evaluation.

Operator	Description	Expression Type
+	Addition	Binary Integer
−	Subtraction	
*	Multiplication	
−	Unary minus	Unary Integer

TABLE I: Arithmetic Operators for Integer type

Operator	Description	Expression Type
==	Equality check (==)	Binary Boolean
<	Less than	
<=	Less than or equals	
>	Greater than	
>=	Greater than or equals	

TABLE II: Comparison Operators for Integer type

Operator	Description	Expression Type
ˆ	Logical AND	Binary Boolean
v	Logical OR	
=>	Logical Implication	
NOT	Logical Negation	Unary Boolean

TABLE III: Logical Operators

IV. Implementation

A. Preliminaries

Our project is implemented in Python, it uses the PLY library for lexing and parsing TPL input programs, while the Z3 SMT solver is used for verifying the validity of the generated VCs. The project is available at Github and includes the following main files and testing folder:

- `lexer.py`: specifies tokenization rules for reserved keywords, operators, annotations, and other syntax elements allowing for the PLY library to tokenize TPL programs, enabling parsing and program analysis;
- `parser.py`: Defines grammar rules for statements, expressions, annotations, and sets the operators' precedence rules. The PLY-based parser constructs an Abstract Syntax Tree (AST) for our language.
- `statement.py`: Defines a hierarchy of statement classes for TPL, including assignments, loops, conditionals, annotations, declarations, and function definitions, Organizing the program structure, and simplifying the process of type checking;
- `expr.py`: Defines a framework for representing and serializing expressions, including binary and unary operations, literals and variables referencing, maps expressions from the TPL syntax to the Z3 syntax;
- `IR.py`: Implements a static analysis framework to validate and verify TPL programs. After ensuring that the program's code follows the rules of the language, the inductive assertion method described in Section II is applied to the code, Z3 is then used to check the validity of the generated FOL formulas.
- `main.py`: The entry point for executing our program verification pipeline;
- tests: Folder containing TPL's programs to test our code.

B. Verification Tool structure

The verification tool depicted in figure 2 is divided into three main components:

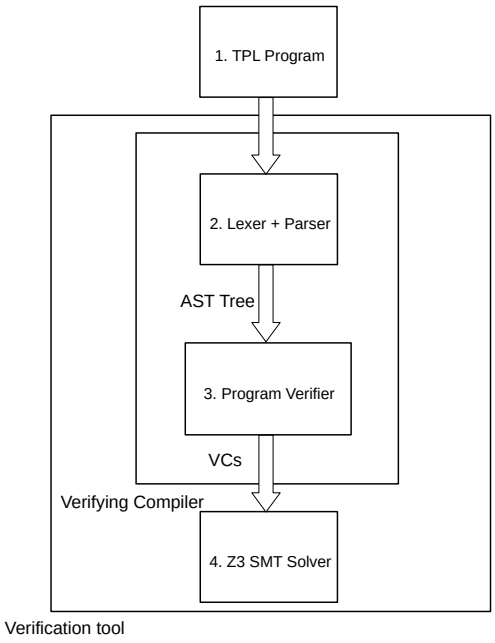

Fig. 2: Figure depicting the different components of the verification tool.

- Lexing and Parsing: This component tokenizes and parses the TPL program. At this stage, the program's syntax is validated, expressions are type checked, and variables are checked for correct declaration before use. The variables of each function are mapped to their corresponding type. As a result of the parsing process, an AST is constructed with the root node representing the entire program while its direct children correspond

to all the defined functions within the program, the leaf nodes represent literal expressions or variable references. The remaining intermediate nodes represent statements or expressions.

- Program Verifier: This component implements the inductive assertion method discussed in Section II. It starts by traversing the program's AST using depth-first search for each defined function, constructing basic paths along the way. These basic paths are then transformed into verification conditions (VCs), representing the logical correctness requirements of the program. Together, the first two components form the verifying compiler.
- Verification via Z3: The verification tool passes the generated VCs to the Z3 solver for validation, accessed via its Python API. Z3 is a Satisfiability Modulo Theories (SMT) solver developed by Microsoft [2].

Note: Z3 is designed to solve problems that combines propositional logic with domain-specific theories, making it highly effective for program verification. Let us highlight a few key features of Z3 that are important for our project:

1) The annotations in TPL are expressed as FOL formulas, Z3 supports FOL, enabling the translation of these annotations into logical constraints that can be validated. Plus, Z3 ensures that these formulas are satisfiable across all basic paths, verifying that the program adheres to its specified behavior.
2) TPL supports integer and boolean variables, and Z3 provides built-in support for integer arithmetic - handles basic operations - and boolean logic - includes logical operators like AND, OR, NOT and IMPLIES.

C. Basic Path Generation

A function can be decomposed into one or more basic paths. Each basic path:

- Starts with the function's precondition or a loop invariant;
- Ends with the postcondition or another loop invariant;
- Is deterministic and unconditional, representing a single, specific execution flow.

By generating a basic path for every branching or looping statement in the function, we ensure that all possible executions of the function are covered. Each path contains only sequential instructions, it starts and ends with annotations, while the rest of the statements are either assertions or assignment statements.

1) Examples - Generating Basic Paths: To illustrate how basic paths are generated, consider the function `abs` declared in Figure 3. The precondition and postcondition annotations of the function imply that the returned value (variable `rv`) must be greater than or equal to 0 for any integer parameter x (note that we are not checking if the returned value is the absolute value of x).

```
(1)  INT FUNCTION abs (INT x) {
         @Pre TRUE;
         @Post rv >= 0;
         IF (x > 0) {
             return x;
         } else {
             return -x;
         }
     }
```

Fig. 3: abs function with annotations.

Transforming the function into its intermediate form yields two basic paths, (2) and (3) presented in Figure 4, corresponding to the branches of the if-else statement in the function's body. Both basic paths start with the precondition and include all statements (in this case, none) before the if-else statement's guard `x > 0`. The first basic path asserts that `x > 0` and thus includes the statement **`return x;`** while the second basic path asserts the opposite and returns $-x$. Both basic paths conclude with the postcondition.

```
(2)  @Pre TRUE;          (3)  @Pre TRUE;
     @ x > 0;                 @ x <= 0;
     return x;                return -x;
     @Post rv >= 0;           @Post rv >= 0;
```

Fig. 4: Basic paths of the abs function.

When encountering a function with a while loop as depicted in Figure 5, we generate a basic path that includes the precondition, the statements that follow, and the loop invariant (5). Upon reaching the loop invariant, we generate two additional basic paths that start from the loop invariant. The first of the two basic paths asserts that the loop guard holds and enters the loop body, it then terminates with the loop invariant after completing the iteration (6). The second basic path exits the while loop by asserting the negation of the guard and continues with the rest of the program (7). All these basic paths are depicted in Figure 6.

D. Verification condition Generation

Verification Condition Generation (VCG) is a crucial step in program verification that converts basic paths into logical formulas, known as verification conditions.

After generating the program's basic paths, the VCG process is started. In this step, the basic paths' annotations and assignment statements are translated into formal logical formulas.

```
(4)  INT FUNCTION pos_double (INT x) {
         DECLARE (INT double_x);
         @PRE x > 0;
         @POST rv == 2 * x;
         double_x := x;
         @LOOP double_x <= 2 * x;
         WHILE (NOT (double_x == 2 * x;)) {
             double_x := double_x
         }
         RETURN double_x;
         }
     }
```

Fig. 5: pos_double function with annotations.

```
(5)  @PRE  x > 0;
     double_x := x;
     RETURN x;
     @LOOP double_x < 2;

(6)  @LOOP double_x < 2;
     @ NOT (double_x == 2 * x);
     double_x := double_x;
     @ NOT (double_x == 2 * x);

(7)  @LOOP double_x < 2;
     @ NOT (NOT (double_x == 2 * x));
     RETURN double_x;
     @POST rv == 2 * x;
```

Fig. 6: Basic paths of the pos_double function.

1) Preconditions and loop invariants: Since these annotations represent the assumptions or requirements that must hold before executing a particular path, they are directly translated to FOL formulas with no additional modifications.

2) Postconditions and loop invariants: They define the outcome conditions after executing all the statements in the path. Since we are using the weakest precondition transformer wp, we need to apply the transformer starting from the postcondition Q and through the intermediate statements and assertions $S_n...S_1$ in reversed order until we reach the precondition P'; the initial state of the program, obtaining intermediate FOL formulas for each statement until the weakest precondition P is generated. The basic path is considered valid, meaning that $\{P'\}S\{Q\}$ holds iff the FOL formula $P' \implies P$ is satisfied for all values of the free variables.

3) Intermediate statements: For each statement S along the path, we apply the wp transformer, taking as input the statement S and the intermediate FOL R: $wp(R, S) = R'$, meaning that for any program state for which R' is valid, executing the statement S will lead to a state for which R is valid. In case S is an assertion for the FOL formula c, then the weakest precondition transformation should generate the FOL formula R' such that if R' holds and c is satisfied, then R is satisfied. Thus $R' = wp(R, S) = wp(R, @\ \text{c};) \equiv (c \implies R)$.

If S is an assignment that when executed, R[v] holds (where v is a variable that is assigned value e), then the weakest precondition transformation should generate R[e]. Thus $R[e] = wp(R[v], v := e;)$.

Below is the VC obtained for the first basic path (2) defined in Figure 4) of the absolute value function:

$$Precondition \implies wp(wp(Postcondition, S2), S1)$$

where Precondition and Postcondition are the FOL formulas of the precondition and postcondition annotations respectively, the VC is then simplified to:

$$\begin{aligned} True &\implies wp(wp(rv \geq 0, rv := x;)S1) \\ &= wp(x \geq 0, S1) \\ &= wp(x \geq 0, @\ \text{x} > 0;) \\ &= x > 0 \implies x \geq 0. \end{aligned}$$

V. Conclusion

This project aimed to implement a verifying compiler for a basic custom made programming Language, leveraging the inductive assertion method and the Z3 SMT solver. The verification tool successfully verifies TPL programs by constructing and checking verification conditions that establish partial correctness properties.

While TPL's simplicity allowed us to focus on core verification techniques, this limitation also highlights opportunities for future work, such as extending the language to support function calls, richer data types, or additional control flow constructs. Furthermore, integrating the verification tool with larger and more practical languages or environments could significantly enhance its applicability.

This project provided valuable insight into the challenges and methodologies of formal verification methods, laying the foundations for further exploration in the field of software verification.

References

[1] Aaron R. Bradley and Zohar Manna. "The Calculus of Computation: Decision Procedures with Applications to Verification". In: 1st ed. Heidelberg: Springer Berlin, 2007. Chap. 5.

[2] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.