Individual Diploma Project

AN AI ASSISTED TOOL FOR CORRECTING SOFTWARE FAULTS USING HOARE LOGIC

Georgios Evangelou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2024

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

An AI Assisted Tool for Correcting Software Faults Using Hoare Logic

Georgios Evangelou

Supervisor

Professor Anna Philippou

The Individual Diploma Thesis was submitted for partial fulfillment of the requirements for obtaining the degree of Computer Science of the Department of Computer Science of the University of Cyprus

May 2024

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Dr. Anna Philippou, for her invaluable guidance throughout this project. Her insight and the contributions to the completion of this work were indispensable.

For providing feedback on the tool implemented in this thesis: Christoforos Seas, Loukia Christina Ioannou, Marios Flourentzou, Gregory Michael.

Finally, I would like to thank my family for their consistent support during my academic endeavors.

Abstract

Software engineering has continuously grown more complex and software applications are routinely used in high-stakes environments. Program correctness is for many applications of paramount importance, yet bugs can occur in every project. Static analysis tools and formal verification methods are widely used in many projects, ensuring correctness.

Due to the ongoing rapid advancement in artificial intelligence, particularly large language models (LLMs), AI-assisted software development techniques are on the rise and their utilization is universal. Numerous methods to improve the capabilities of LLMs and guarantee the validity of the generated code are emerging, as it is an active area of research.

This thesis presents the design and implementation of an interactive tool used to statically detect bugs in a procedural language. It employs a design by contract methodology, powered by a Hoare-like system. To accelerate the debugging process, a large language model is integrated as an assistant in identifying bugs and correcting programs. Characteristics regarding the tool's accuracy and usability, as well as possible extensions are analyzed in depth.

Contents

Table of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivation

Program correctness is crucial in software engineering as software faults can damage the revenue and reputation of an organization or even cost lives. However, software development is a complex process where software bugs occur regularly in every project, and debugging can take a considerable amount of time [1]. Various tools that detect such faults and guarantee the correctness of programs are routinely used in software development, and currently the rise of large language models (LLMs) through generative AI as programming assistants is prominent. Tools like GitHub Copilot [2], which is based on code completion and dcc –help [3], which augments compiler error messages are a handful in a plethora of many attempts to improve the software development process through LLMs.

1.2 Purpose

In this thesis, a tool is designed and implemented which approaches the debugging process from a verification perspective powered by a system which extends Hoare logic for a procedural programming language. Moreover, the tool incorporates an LLM as an assistant in finding bugs and correcting programs.

1.3 Methodology

A baseline target was set for the functional requirements of the tool, as well as possible extensions to this target. The formal background needed to design the tool was researched along with the necessary technologies to implement it. The tool was built incrementally by extending the language from an unstructured imperative language to a procedural language that supports recursion. In the final stages of the development the LLM assistant was integrated.

The tool's usability was evaluated through the feedback received from a limited conducted survey, where participants were asked to debug some programs using the tool. Some notes on the performance, where determined using a profiler.

1.4 Outline of Thesis

In the next chapter, the background behind the structure of the language and the verification system used in this thesis is described along with the necessary tools to implement them using automated proving techniques. Additionally. some information around the capabilities and usefulness of LLMs with respect to this thesis is given. Other similar work in static analysis is also discussed in this chapter.

Chapter 3 describes the design and operation of the tool. The language's syntax and semantics are specified along with a proof system used to reason for the correctness of a program. The basic algorithm that integrates the LLM is also discussed here.

In Chapter 4, the overall architecture of the tool is analysed and how each software component was implemented. Examples which present the basic functionalities of the tool and its operation are included.

In Chapter 5, qualitative characteristics of the tool are discussed regarding its usability, performance, and other limitations, as well as methods which may improve these shortcomings.

Chapter 6 concludes this thesis with some final thoughts and discusses some directions in which this work can be extended in the future and some challenges faced during this thesis.

Chapter 2 Background

2.1 Logic Errors

Many logic errors occur when programming, which are hard to detect and correct, since they do not violate the syntax or the semantics of the language. Common tools like compilers can detect some common logic errors (e.g. missing return, read before assignment), but for other errors no attempt is made to detect them, as this would penalise compilation times. Other tools (static analysers, debuggers, etc) exist which can assist the programmer in detecting such errors.

Alzahrani and Vahid [2] categorized logic errors found in thousands of student-written programs, many of which are caused by misinterpreting the syntax or the semantics, (e.g. confusing operator precedence). Others may be caused by typos (e.g. equality vs assignment operator in C) or by omitting a required step or misplacing code.

2.2 Formal Methods

Formal methods are techniques used to model the specification and verify the behaviour of programs. Such specifications are constructed using mathematical logic which are verified using inference rules in that logic, or state-space exploration techniques. Using formal methods, the whole state space of a program can be symbolically inspected, and certain properties like safety and liveness can be deduced [3].

2.3 Program Correctness

The notion of program correctness asserts that according to a specification if its behaviour satisfies that specification. Correctness can be distinguished as either "partial correctness" where a program is regarded as correct if its specification is satisfied when it terminates, or "total correctness" where a program is also required to terminate in addition to satisfying its specifications. Proving total correctness is an undecidable problem due to the halting problem.

2.4 Hoare Logic

Hoare logic, conceived by C.A.R Hoare, is a formal system for proving the correctness of programs. Its central concept are Hoare triples, notated as $\{\phi\}\ C\ \{\psi\}$ which state that when the precondition ϕ is satisfied, after executing the program C the postcondition ψ holds. These triples can be proved for correctness using an associated set of rules and axioms for each construct of the language. Hoare's initial work [4] included rules for a simple unstructured imperative language, and has since been extended to include recursive procedures [5], or mutable state through separation logic [6]. The formal verification system behind the tool designed in this thesis, is based on Hoare's revision of his original work, which includes recursive procedures [5].

Figure 2.1 presents the Hoare rules for partial and total correctness for a simple unstructured imperative language (WHILE language). Extensions of this system used in

the implementation of the tool designed in this thesis are discussed in detail in Section 3.4.1.

$$\frac{}{\{\phi[e/x]\}\ x{:=}e;\ \{\phi\}}\ Assignment \qquad \frac{\phi\rightarrow\phi_0 \quad \{\phi_0\}\ S\ \{\psi_0\} \quad \psi_0\rightarrow\psi}{\{\phi\}\ S\ \{\psi\}}\ Consequence$$

$$\frac{\{\phi\}\ S_0\ \{\eta\} \quad \{\eta\}\ S_1\ \{\psi\}}{\{\phi\}\ S_0\ S_1\ \{\psi\}}\ Composition \qquad \frac{\{\phi\wedge\beta\}\ \overline{S_0}\ \{\psi\} \quad \{\phi\wedge\neg\beta\}\ \overline{S_1}\ \{\psi\}}{\{\phi\}\ if\ \beta\ \overline{\{S_0\}}\ else\ \overline{\{S_1\}}\ \{\psi\}}\ If-Statement$$

$$\frac{\{\eta\wedge\beta\}\ \bar{S}\ \{\eta\}}{\{\eta\}\ while\ \beta\ \{\bar{S}\}\ \{\eta\wedge\neg\beta\}}\ Partial-while \qquad \frac{\{\eta\wedge\beta\wedge 0\leq e=e_0\}\ \bar{S}\ \{\eta\wedge 0\leq e<e_0\}}{\{\eta\wedge 0\leq e\}\ while\ \beta\ \{\bar{S}\}\ \{\eta\wedge\neg\beta\}}\ Total-while$$

Figure 2.1 Hoare Rules for Partial and Total Correctness

Hoare rules by themselves cannot be used in a semi-automated process (a fully automated process cannot exist since proving termination is undecidable). Instead, a reformulation by Edsger Dijkstra is used [7], the weakest preconditions, which can be computed using a recursive function. Programs still must be annotated with invariant conditions to prove the correctness of loop commands and variant expressions (strictly decreasing expressions through each iteration) to prove termination.

2.5 Satisfiability Modulo Theories and Z3 Theorem Prover

Satisfiability Modulo Theories (SMT) is a generalization of the Boolean SAT problem where the target is to decide the satisfiability of a set of logic formulas which may include elements and operations from theories beyond propositional logic, such as real numbers, integers, bit-vectors, strings and uninterpreted functions. The inclusion of many of theories causes instances of the problem to be semi-decidable or even undecidable.

SMT solvers are ubiquitous in program analysis, due to their expressiveness. Modern solvers are based on extensions of the DPLL and CDCL algorithms which are capable of reasoning about arbitrary theories. The Z3 Theorem Prover is one such solver developed by Microsoft Research which utilizes those algorithms (and many other techniques) [8] and is used by many program analysis tools, including the one presented in this thesis.

2.6 Design by Contract

Design by Contract, coined by Bertrand Meyer, is a methodology for designing software systems rooted in formal verification and Hoare Logic. Software components abide to a list of formally described specifications which include preconditions, postconditions and invariants, like a business contract, where the involved parties must meet the documented obligations to receive the agreed upon benefits [9]. A contract can be enforced at runtime in languages that support such functionality (like Eiffel), and by using a test-suite or simply documenting it in languages with no such support. Another approach would be to formally verify that contracts are enforced in the entire codebase at compile time.

2.7 Large Language Models

Large Language Models (LLMs) are machine learning models trained on vast amounts of corpora that aim to perform various natural language processing tasks such text generation and translation. [10] Accomplished models employ the Transformer which is a model architecture based on an attention mechanism which determines for each token the relevancy of other tokens with respect to that token [11]. In recent years, LLMs grew in popularity as they are publicly available, and attention is drawn to the models' capabilities of generating and analysing code. This feature is utilized by the tool described in this thesis.

2.8 Related Work

In this section are described a handful of tools which incorporate static analysis for verifying specifications and detecting bugs. The design of the tool described in this thesis is partially inspired by these tools.

2.8.1 Dafny

Dafny is a verification system and programming language developed by Microsoft Research, where program specifications can be formally expressed, and a verifier can statically check their correctness. It includes various imperative and functional features

such as generic classes and inductive datatypes. Program verification is based on extensions of Hoare logic such as implicit dynamic frames which are a variation of separation logic using the Z3 Automated Theorem Prover. Verified programs can be source-to-source translated to many general-purpose programming languages such as Go, C++, Java, and others [12]. Dafny strongly resembles the tool developed in this thesis due its design by contract methodology which is statically enforced albeit with greater focus to verification.

2.8.2 Infer

Infer is a static analysis tool developed by Meta Platforms, Inc., for Java C/C++ and Objective-C that can detect bugs such as null dereferences and memory leaks. This analyser is continuously processing modifications in the code of the Facebook apps to ensure certain program properties. It uses separation logic and includes a modular analysis engine which allows the extension of analyses for other domain such as security and concurrency [13]. Similar to the tool in this thesis, Infer approaches program analysis and bug detection using formal methods.

2.8.3 Frama-C

Frama-C is a platform which uses several collaborative plugins that perform static and dynamic analysis to verify requirements in C programs. Specifications are described using the ANSI/ISO C Specification Language (ACSL) which is used to annotate the abstract syntax tree of a program and analysers ensure the validity of desired properties. More specifically, the WP (Weakest Precondition) plugin can perform deductive proofs of ACSL contracts [14]. The tool in this thesis takes after the deductive approach of Frama-C to guarantee annotated properties.

Chapter 3 Design

3.1 Abstract Syntax

Figure 3.1 describes the abstract syntax for the tool language. It is an amalgamation of an assertion language and a programming language where assertions are annotated before designated programming language constructs. Let f range over declaration names, x over variable names, n over integer literals and b over Boolean literals (true, false). Binary operators $\oplus$ are the arithmetic, division, relational and logical operators listed representing the usual operators found in C-family languages or propositional logic. Arithmetic and Boolean unary operators $!_*$ are shown below.

Notation $\bar{x}$ stands for $x_1, x_2, \ldots, x_n$ for expressions and $\bar{S}/\bar{D}$ stands for $S_1\ S_2\ \ldots\ S_n/D_1\ D_2\ \ldots D_n$ for statements and declarations respectively. Let e range over expressions which can be one of the following: the $result$ literal, a Boolean b, an integer n, a variable x, a call expression $f(\bar{e})$, a binary expression $e_0 \oplus e_1$, a unary expression $!_* e$, a conditional ternary expression $e_0 ? e_1 : e_2$ or a quantified expression $\forall \bar{x}.\, e$ or $\exists \bar{x}.\, e$.

Let S range over statements which can be one of the following: an assignment $x := e;$, an assertion $assert\ e;$, an if-else statement $if\ e\ \{\bar{S_0}\}\ else\ \{\bar{S_1}\}$, a while statement $\#invariant\ e_0\ \#variant\ e_1\ while\ e\ \{\bar{S}\}$ or a return statement $return\ e;$.

Let D range over declarations which have three forms: a function declaration $fn\ f(\bar{x}) := e;$, a predicate declaration $pred\ f(\bar{x}) := e;$ or a procedure declaration $\#pre\ e_0\ \#post\ e_1\ \#variant\ e_2\ f(\bar{x})\ \{\bar{S}\}$.

A program P is a sequence of declarations $\bar{D}$.

Declaration name	f	
Variable name	x	
Integer literal	n	
Binary operator	$\oplus ::=$	
	Arithmetic operator	$+ \mid - \mid *$
	Division operator	$/ \mid \%$
	Relational operator	$< \mid \leq \mid = \mid \neq \mid \geq \mid >$
	Logical operator	$\wedge \mid \vee \mid \rightarrow$
Unary operator	$!_* ::= + \mid - \mid \neg$	
Boolean literal	$b ::= true \mid false$	
Expression	$e ::=$	
	Result literal	$result$
	Boolean literal	b
	Integer literal	n
	Variable	x
	Call expression	$f(\bar{e})$
	Binary expression	$e_0 \oplus e_1$
	Unary expression	$!_* e$
	Ternary expression	$e_0 ? e_1 : e_2$
	Forall expression	$\forall \bar{x}. e$
	Exists expression	$\exists \bar{x}. e$
Statement	$S ::=$	

	$Assignment$	$x \coloneqq e;$
	$Assertion$	$assert\ e;$
	$If-else\ statement$	$if\ e\ \{\bar{S}_0\}\ else\ \{\bar{S}_1\}$
	$While\ loop$	$\#invariant\ e_0\ \#variant\ e_1\ while\ e\ \{\bar{S}\}$
	$Return\ statement$	$return\ e;$
$Declaration$	$D ::=$	
	$Function\ declaration$	$fn\ f(\bar{x}) \coloneqq e;$
	$Predicate\ declaration$	$pred\ f(\bar{x}) \coloneqq e;$
	$Procedure\ declaration$	$\#pre\ e_0\ \#post\ e_1\ \#variant\ e_2\ proc\ f(\bar{x})\{\bar{S}\}$
$Program$	$P ::= \bar{D}$	

Figure 3.1 Tool Language Abstract Syntax

3.2 Typing

3.2.1 Typing Context

Since the language includes constructs that describe specifications and program instructions, multiple judgement contexts are used to determine well-formedness.

Judgement $\Gamma \vdash_c e{:}\,t$ holds if in context c and environment Γ expression e has type t. There are five overlapping contexts used: $meta$ context which applies to invariants, variants, preconditions, functions and predicates, $post$ context which applies to postconditions, $call$ context which applies to procedure calls, $total$ context which applies to total expressions (containing no procedure calls or division operators), and $partial$ context which applies to all the previous contexts and call free program statements.

Let E_c denote the set of expressions that occur in $\bar{D}$ and are well-formed in context c given an environment Γ. Then $E_{total} \subset E_{partial} \subset E_{meta} \subset E_{post}$. Also, if $\bar{e} \subseteq E_{total}$ and $f \in proc(\bar{D})$ then $f(\bar{e}) \in E_{call}$.

3.2.2 Auxiliary Functions

Figure 3.2 describes auxiliary functions used to specify the typing and verification rules. Let t range over type names which can be Boolean $\mathbb{B}$, integral $\mathbb{Z}$ or mutable type $mut\ t$.

Function $fn(\overline{D})$ returns a set with the names of all function declarations in $\overline{D}$. Function $pred(\overline{D})$ returns a set with the names of all predicate declarations in $\overline{D}$. Function $proc(\overline{D})$ returns a set with the names of all procedure declarations in $\overline{D}$.

Predicate $procFree(e)$ holds if expression e contains no calls to a procedure. Similarly, predicate $divFree(e)$ holds if expression e contains no division or modulo operations. Predicate $total(e)$ holds if $procFree(e)$ and $divFree(e)$.

Predicate $assignable(\Gamma, e, t)$ holds if expression e can be assigned to a variable with type t in an environment Γ. Assignable expressions are total expressions, top-level division/modulo expressions, and top-level procedure calls. This restriction allows the inclusion of certain verification rules (see Section 3.4.1) on non-total expressions, since they require extra preconditions which are examined at per-statement level.

Predicate $endsReturn(\bar{S})$ holds if for every execution path the last statement in $\bar{S}$ is a return statement. Predicate $ifReturn(if\ e\ \{\overline{S_0}\}\ else\ \{\overline{S_1}\})$ holds if in both branches $\overline{S_0}, \overline{S_1}$ predicate $endsReturn$ holds.

Predicates not shown in Figure 3.2 $arith(\oplus), div(\oplus), rel(\oplus), log(\oplus)$ hold if $\oplus$ is an arithmetic, division, relational or logical operator respectively, as shown in the abstract syntax. Similarly, predicate $arith(!_*)$ holds when unary operator $!_*$ is $+$ or $-$. Predicate $distinct(\bar{x})$ holds if all elements in sequence $\bar{x}$ are distinct.

Judgements in $partial$ context are denoted without a subscript. For all other contexts their names are subscripted in judgements.

$Type\ name\ t ::= \mathbb{B} \mid \mathbb{Z} \mid mut\ t$

$$\frac{fn\ f(\bar{x}):=e;\in\bar{D}}{f\in fn(\bar{D})} \qquad \frac{pred\ f(\bar{x}):=e;\in\bar{D}}{f\in pred(\bar{D})} \qquad \frac{\#pre\ e_0\ \#post\ e_1\ \#variant\ e_2\ proc\ f(\bar{x})\{\bar{S}\}\in\bar{D}}{f\in proc(\bar{D})}$$

$$\frac{}{procFree(n)} \qquad \frac{}{procFree(b)} \qquad \frac{}{procFree(x)} \qquad \frac{procFree(e)}{procFree(!_* e)}$$

$$\frac{procFree(e_0)\quad procFree(e_1)}{procFree(e_0 \oplus e_1)} \qquad \frac{procFree(e_0)\quad procFree(e_1)\quad procFree(e_2)}{procFree(e_0 ? e_1 : e_2)}$$

$$\frac{}{divFree(n)} \qquad \frac{}{\text{divFree(b)}} \qquad \frac{}{\text{divFree(x)}} \qquad \frac{divFree(e)}{divFree(!_* e)}$$

$$\frac{divFree(e_0)\quad divFree(e_1)\ \neg div(\oplus)}{divFree(e_0 \oplus e_1)} \qquad \frac{divFree(e_0)\quad divFree(e_1)\quad divFree(e_2)}{divFree(e_0 ? e_1 : e_2)}$$

$$\frac{procFree(e)\quad divFree(e)}{total(e)} \qquad \frac{\Gamma\vdash_{total} e{:}t}{assignable(\Gamma,e,t)} \qquad \frac{\Gamma\vdash e_0\oplus e_1{:}t\quad total(e_0)\quad total(e_1)}{assignable(\Gamma,e_0\oplus e_1,t)} \qquad \frac{\Gamma\vdash_{call} f(\bar{e}){:}\mathbb{Z}}{assignable(\Gamma,f(\bar{e}),\mathbb{Z})}$$

$$\frac{\bar{S}\ return\ e;}{endReturn(\bar{S}\ return\ e;)}$$

$$\frac{\bar{S}\ if\ e\ \{\overline{S_0}\}\ else\ \{\overline{S_1}\}\quad ifReturn(if\ e\ \{\overline{S_0}\}\ else\ \{\overline{S_1}\})}{endReturn(\bar{S}\ if\ e\ \{\overline{S_0}\}\ else\ \{\overline{S_1}\})} \qquad \frac{if\ e\ \{\overline{S_0}\}\ else\ \{\overline{S_1}\}\quad endReturn(\overline{S_0})\quad endReturn(\overline{S_1})}{ifReturn(if\ e\ \{\overline{S_0}\}\ else\ \{\overline{S_1}\})}$$

Context Conversion

$$\frac{\Gamma\vdash e{:}t\quad \text{procFree(e)}}{\Gamma\vdash_{meta} e{:}t} \qquad \frac{\Gamma\vdash_{meta} e{:}t}{\Gamma\vdash_{post} e{:}t} \qquad \frac{\Gamma\vdash e{:}t\quad \text{total(e)}}{\Gamma\vdash_{total} e{:}t}$$

Figure 3.2 Typing Auxiliary Functions

3.2.3 Typing Rules

In this section, the typing rules and axioms for the language are presented. Judgements may be converted to different contexts according to the rules given in Figure 3.2. For each rule presented with conclusion $\Gamma \vdash_{meta} e{:}\,t$, there is a structurally identical rule with $post$ context $\Gamma \vdash_{post} e{:}\,t$ which is not presented for brevity. Whenever an expression is referred to have a type, it is implied that it is well-typed.

Figure 3.3 presents the typing rules for expressions. Let Γ range over typing environments, composed of pairs $x{:}\,t$, denoting that variable x has type t. Notation $\overline{x{:}\,\mathbb{Z}}$ stands for $x_1{:}\,\mathbb{Z}, x_2{:}\,\mathbb{Z}, \ldots, x_n{:}\,\mathbb{Z}$. Notation $\overline{\Gamma \vdash_c e{:}\,\mathbb{Z}}$ stands for $\Gamma \vdash_c e_1{:}\,\mathbb{Z}\ \Gamma \vdash_c e_2{:}\,\mathbb{Z} \ldots \Gamma \vdash_c e_n{:}\,\mathbb{Z}$. Judgement $\Gamma \vdash_c e{:}\,t$ holds if in context c and environment Γ expression e has type t. Integer literals have integral type and Boolean literals have Boolean type for any environment and context. The $result$ expression has

integral type in $post$ context. Variables have a type determined by their environment and mutable variables can be treated as immutable. Binary expressions $e_0 \oplus e_1$ where $\oplus$ is an arithmetic or division operator have integral type with environment Γ if expressions e_0, e_1 have integral type with environment Γ. Binary expressions $e_0 \oplus e_1$ where $\oplus$ is a relational operator have Boolean type with environment Γ if expressions e_0, e_1 have integral type with environment Γ. Binary expressions $e_0 \oplus e_1$ where $\oplus$ is a logical operator have Boolean type with environment Γ if expressions e_0, e_1 have Boolean type with environment Γ. A ternary expression $e_0 ? e_1 : e_2$ has type t in environment Γ if e_0 has Boolean type and e_1, e_2 have type t in environment Γ. A unary expression $!_* e$ where $!_*$ is an arithmetic operator has integral type with environment Γ if e has integral type with environment Γ. Unary expression $\neg e$ has Boolean type with environment Γ if e has Boolean type with environment Γ. A procedure call $f(\bar{e})$ has integral type with environment Γ and context $call$ if there exists a procedure declaration with name f in $\overline{D}$ and each argument e_i has integral type with environment Γ and context $total$. A quantified expression $\forall \bar{x}. e$ or $\exists \bar{x}. e$ has Boolean type with environment Γ and context $meta$, if variables $\bar{x}$ are distinct and expression e has Boolean type with environment $\Gamma, \overline{x:\mathbb{Z}}$ and context $meta$. A function call $f(\bar{e})$ has integral type with environment Γ and context $meta$ if there exists a function declaration f in $\overline{D}$ and arguments $\bar{e}$ have integral type with environment Γ. Similarly, a predicate call $f(\bar{e})$ has Boolean type with environment Γ and context $meta$ if there exists a function declaration f in $\overline{D}$ and arguments $\bar{e}$ have integral type with environment Γ.

$$\frac{}{\Gamma \vdash n:\mathbb{Z}} \quad \frac{}{\Gamma \vdash b:\mathbb{B}} \quad \frac{}{\Gamma \vdash_{post} result:\mathbb{Z}} \quad \frac{\Gamma(x)=t}{\Gamma \vdash x:t} \quad \frac{\Gamma \vdash x:mut\ t}{\Gamma \vdash x:t}$$

$$\frac{\Gamma \vdash e_0:\mathbb{Z} \quad \Gamma \vdash e_1:\mathbb{Z} \quad arith(\oplus)}{\Gamma \vdash e_0 \oplus e_1:\mathbb{Z}} \quad \frac{\Gamma \vdash e_0:\mathbb{Z} \quad \Gamma \vdash e_1:\mathbb{Z} \quad div(\oplus)}{\Gamma \vdash e_0 \oplus e_1:\mathbb{Z}} \quad \frac{\Gamma \vdash e_0:\mathbb{Z} \quad \Gamma \vdash e_1:\mathbb{Z} \quad rel(\oplus)}{\Gamma \vdash e_0 \oplus e_1:\mathbb{B}} \quad \frac{\Gamma \vdash e_0:\mathbb{B} \quad \Gamma \vdash e_1:\mathbb{B} \quad log(\oplus)}{\Gamma \vdash e_0 \oplus e_1:\mathbb{B}}$$

$$\frac{\Gamma \vdash e:\mathbb{Z} \quad arith(!_*)}{\Gamma \vdash !_* e:\mathbb{Z}} \quad \frac{\Gamma \vdash e:\mathbb{B}}{\Gamma \vdash \neg e:\mathbb{B}} \quad \frac{\Gamma \vdash e:\mathbb{B} \quad \Gamma \vdash e_1:t \quad \Gamma \vdash e_2:t}{\Gamma \vdash (e ? e_1 : e_2):t}$$

$$\frac{distinct(\bar{x}) \quad \Gamma, \overline{x:\mathbb{Z}} \vdash_{meta} e:\mathbb{B}}{\Gamma \vdash_{meta} \forall \bar{x}. e:\mathbb{B}} \qquad \frac{distinct(\bar{x}) \quad \Gamma, \overline{x:\mathbb{Z}} \vdash_{meta} e:\mathbb{B}}{\Gamma \vdash_{meta} \exists \bar{x}. e:\mathbb{B}}$$

$$\frac{\overline{\Gamma \vdash_{meta} e:\mathbb{Z}} \quad f \in fn(\overline{D})}{\Gamma \vdash_{meta} f(\bar{e}):\mathbb{Z}} \qquad \frac{\overline{\Gamma \vdash_{meta} e:\mathbb{Z}} \quad f \in pred(\overline{D})}{\Gamma \vdash_{meta} f(\bar{e}):\mathbb{B}} \qquad \frac{\overline{\Gamma \vdash_{total} e:\mathbb{Z}} \quad f \in proc(\overline{D})}{\Gamma \vdash_{call} f(\bar{e}):\mathbb{Z}}$$

Figure 3.3 Typing Rules for Expressions

Figure 3.4 presents the typing rules for statements. Judgement Γ ⊢ S holds if statement S is well-formed with environment Γ. An assertion $assert\ e;$ is well-formed with environment Γ if expression e has Boolean type in $total$ context. Similarly, a return statement $return\ e;$ is well-formed with environment Γ if expression e has integral type in $total$ context since procedures always produce an integer value. An assignment $x := e;$ is well-formed with environment Γ if expression e is assignable with type t with environment Γ and either variable x has type $mut\ t$ in Γ, or does not exist in Γ with any other type. A statement sequence $x := e; \bar{S}$ is well-formed with environment Γ if expression e has type t with environment Γ, statement $x :=$ e; is well-formed with environment Γ and statement sequence is well-formed with environment $\Gamma, x: mut\ t$. A statement sequence $S_0\ S_1$ is well-formed with environment Γ if each statement is well-formed with environment Γ. An if-else statement $if\ e\ \{\overline{S_0}\}\ else\ \{\overline{S_1}\}$ is well-formed with environment Γ if statement sequences $\overline{S_0}, \overline{S_1}$ are well-formed and expression e has Boolean type in $total$ context. A while statement $\#invariant\ e_0\ \#variant\ e_1\ while\ e\ \{\bar{S}\}$ is well-formed with environment Γ if with environment Γ expression e_0 has Boolean type in $meta$ context, expression e_1 has integral type in $meta$ context, expression e has Boolean type in $total$ context and statement sequence $\bar{S}$ is well-formed.

$$\frac{\Gamma \vdash_{total} e:\mathbb{B}}{\Gamma \vdash assert\ e;} \qquad \frac{\Gamma \vdash_{total} e:\mathbb{Z}}{\Gamma \vdash return\ e;} \qquad \frac{\Gamma \vdash_{total} e:\mathbb{B} \quad \Gamma \vdash \overline{S_0} \quad \Gamma \vdash \overline{S_1}}{\Gamma \vdash if\ e\ \{\overline{S_0}\}\ else\ \{\overline{S_1}\}} \qquad \frac{\Gamma \vdash S_0 \quad \Gamma \vdash S_1}{\Gamma \vdash S_0\ S_1}$$

$$\frac{\Gamma \vdash e:t \quad \Gamma \vdash x := e; \quad \Gamma, x: mut\ t \vdash \bar{S}}{\Gamma \vdash x := e;\ \bar{S}}$$

$$\frac{\Gamma \vdash x: mut\ t \quad assignable(\Gamma, e, t)}{\Gamma, x: mut\ t \vdash x := e;} \qquad \frac{\forall t \neg(\Gamma \vdash x:t) \quad assignable(\Gamma, e, t)}{\Gamma, x: mut\ t \vdash x := e;}$$

$$\frac{\Gamma \vdash_{meta} e_0:\mathbb{B} \quad \Gamma \vdash_{meta} e_1:\mathbb{B} \quad \Gamma \vdash_{total} e:\mathbb{B} \quad \Gamma \vdash \bar{S}}{\Gamma \vdash \#invariant\ e_0\ \#variant\ e_1\ while\ e\ \{\bar{S}\}}$$

Figure 3.4 Typing Rules for Statements

Figure 3.5 presents the typing rules for declarations and programs. Judgement ⊢ D holds if declaration D is well-formed. A function declaration $fn\ f(\bar{x}) := e;$ is well-formed if parameters $\bar{x}$ are distinct, and e has integral type with the environment consisting of the parameters typed as $\mathbb{Z}$ in $meta$ context. A predicate declaration $pred\ f(\bar{x}) := e;$ is well-formed if parameters $\bar{x}$ are distinct, and e has Boolean type with the environment

consisting of the parameters typed as $\mathbb{Z}$ in $meta$ context. A procedure declaration $\#pre\ e_0\ \#post\ e_1\ \#variant\ e_2\ proc\ f(\bar{x})\ \{\bar{S}\}$ is well-formed if parameters $\bar{x}$ are distinct and with the environment consisting of the parameters typed as $\mathbb{Z}$: e_0 has Boolean type in $meta$ context, e_1 has Boolean type in $post$ context, e_2 has integral type in $meta$ context, $\bar{S}$ is well-formed and $\bar{S}$ ends with a return statement in every execution path.

Judgement $\vdash P$ holds if all declarations $\overline{D}$ in P are distinct and well-formed.

Well-formed declarations

$$\frac{distinct(\bar{x}) \quad \overline{x{:}\mathbb{Z}} \vdash_{meta} e{:}\mathbb{Z}}{\vdash fn\ f(\bar{x}){:=}e;} \qquad \frac{distinct(\bar{x}) \quad \overline{x{:}\mathbb{Z}} \vdash_{meta} e{:}\mathbb{B}}{\vdash pred\ f(\bar{x}){:=}e;}$$

$$\frac{distinct(\bar{x}) \quad \overline{x{:}\mathbb{Z}} \vdash_{meta} e_0{:}\mathbb{B} \quad \overline{x{:}\mathbb{Z}} \vdash_{post} e_1{:}\mathbb{B} \quad \overline{x{:}\mathbb{Z}} \vdash_{meta} e_2{:}\mathbb{Z} \quad \text{endReturn}(\bar{S}) \quad \overline{x{:}\mathbb{Z}} \vdash \bar{S}}{\vdash \#pre\ e_0\ \#post\ e_1\ \#variant\ e_2\ proc\ f(\bar{x})\{\bar{S}\}}$$

Well-formed programs

$$\frac{distinct(\overline{D}) \quad \overline{\vdash D} \quad P{=}\overline{D}}{\vdash P}$$

Figure 3.5 Typing Rules for Declarations and Programs

3.3 Reduction Rules

Figure 3.6 presents the reduction rules and axioms for the language. Specification clauses (prefixed by '#') are omitted for brevity. Let σ denote a non-erroneous state of the program which maps variables to values. Assume an initial non-erroneous empty program state. Normal form values are Boolean or integer values. Operators $\bigoplus', !'_*$ denotes the concrete version of a symbolic operator $\bigoplus, !_*$ respectively. The symbol $\perp$ denotes an erroneous value or program state, where in both cases execution halts.

Judgement $e, \sigma \Downarrow v$ holds if expression e with state σ evaluates to value v. Arithmetic and relational expressions evaluate to their normal form by evaluating their operands and applying their operator on the resulting values. Division expressions abort execution when the divisor is zero. Logical expressions and ternary expressions short-circuit their evaluation and implication expressions $e_0 \rightarrow e_1$ are evaluated as $\neg e_0 \vee e_1$. Procedure

calls are evaluated by substituting the parameters of the callee procedure in the program state with the evaluated arguments and evaluating the procedure body, where variable $result$ maps in the state to the evaluated value.

Judgement $S, \sigma \Downarrow \sigma'$ holds if statement S with state σ evaluates to a new state σ'. Two statements may be sequenced if the first statement is not a return statement, and it does not evaluate to a $\perp$ state. An assignment $x \coloneqq e$ with a state σ evaluates to a state $\sigma[x \coloneqq v]$ if e with state σ evaluates to v. A return statement $return\ e;$ with a state σ evaluates to a state $\sigma[result \coloneqq n]$ if expression e with state σ evaluates to n. An assertion evaluates to a $\perp$ state if its expression evaluates to false, otherwise it evaluates to the same state. If-else statements evaluate to the state determined by the branch which is to be executed, which is determined by the value evaluated by the statement's condition. A while statement evaluates to the current state if its guard condition evaluates to false, otherwise the body is evaluated for each successive state until the guard condition evaluates to false.

Program State σ $\qquad \sigma[x \coloneqq v](y) = \begin{cases} v & y = x \\ \sigma(y) & \end{cases}$

Value $v ::= b \mid n$

Expressions

$$\frac{}{n,\sigma \Downarrow n} \quad \frac{}{b,\sigma \Downarrow b} \quad \frac{}{x,\sigma \Downarrow \sigma(x)} \quad \frac{e_0,\sigma \Downarrow n_0 \quad e_1,\sigma \Downarrow n_1 \ arith(\oplus)}{e_0 \oplus e_1,\sigma \Downarrow n_0 \oplus' n_1}$$

$$\frac{e_1,\sigma \Downarrow 0 \quad div(\oplus)}{e_0 \oplus e_1,\sigma \Downarrow \perp} \quad \frac{e_0,\sigma \Downarrow n_0 \quad e_1,\sigma \Downarrow n_1 \quad n_1 \neq 0 \quad div(\oplus)}{e_0 \oplus e_1,\sigma \Downarrow n_0 \oplus' n_1} \quad \frac{e_0,\sigma \Downarrow n_0 \quad e_1,\sigma \Downarrow n_1 \ rel(\oplus)}{e_0 \oplus e_1,\sigma \Downarrow n_0 \oplus' n_1}$$

$$\frac{e_0,\sigma \Downarrow false}{e_0 \wedge e_1,\sigma \Downarrow false} \quad \frac{e_0,\sigma \Downarrow true \quad e_1,\sigma \Downarrow b_1}{e_0 \wedge e_1,\sigma \Downarrow b_1} \quad \frac{e_0,\sigma \Downarrow true}{e_0 \vee e_1,\sigma \Downarrow true} \quad \frac{e_0,\sigma \Downarrow false \quad e_1,\sigma \Downarrow b_1}{e_0 \vee e_1,\sigma \Downarrow b_1}$$

$$\frac{e_0,\sigma \Downarrow false}{e_0 \rightarrow e_1,\sigma \Downarrow true} \quad \frac{e_0,\sigma \Downarrow true \quad e_1,\sigma \Downarrow b_1}{e_0 \rightarrow e_1,\sigma \Downarrow b_1}$$

$$\frac{e,\sigma \Downarrow n \ arith(!_*)}{!_* e,\sigma \Downarrow !'_* n} \quad \frac{e,\sigma \Downarrow b}{\neg e,\sigma \Downarrow \neg' b} \quad \frac{e,\sigma \Downarrow true \quad e_1,\sigma \Downarrow v_1}{(e ? e_1 : e_2),\sigma \Downarrow v_1} \quad \frac{e,\sigma \Downarrow false \quad e_2,\sigma \Downarrow v_2}{(e ? e_1 : e_2),\sigma \Downarrow v_2}$$

$$\frac{\overline{e,\sigma \Downarrow n} \quad proc\ f(\bar{x})\{\bar{S}\} \quad \bar{S},\sigma[\overline{x \coloneqq e}] \Downarrow \sigma'}{f(\bar{e}),\sigma \Downarrow \sigma'(result)}$$

Statements

$$\frac{S_0 \neq return\ e; \quad S_0,\sigma_0 \Downarrow \sigma_1 \quad S_1,\sigma_1 \Downarrow \sigma_2}{S_0\ S_1,\sigma_0 \Downarrow \sigma_2} \quad \frac{e,\sigma \Downarrow v}{x \coloneqq e;,\sigma \Downarrow \sigma[x \coloneqq v]}$$

$$\frac{e,\sigma \Downarrow true \quad \overline{S_1},\sigma \Downarrow \sigma'}{if\ e\ \{\overline{S_1}\}\ else\ \{\overline{S_2}\},\sigma \Downarrow \sigma'} \quad \frac{e,\sigma \Downarrow false \quad \overline{S_2},\sigma \Downarrow \sigma'}{if\ e\ \{\overline{S_1}\}\ else\ \{\overline{S_2}\},\sigma \Downarrow \sigma'}$$

$$\frac{e,\sigma \Downarrow true \quad \bar{S}\ while\ e\ \{\bar{S}\},\sigma \Downarrow \sigma'}{while\ e\ \{\bar{S}\},\sigma \Downarrow \sigma'} \qquad \frac{e,\sigma \Downarrow false}{while\ e\ \{\bar{S}\},\sigma \Downarrow \sigma}$$

$$\frac{e,\sigma \Downarrow n}{return\ e;,\sigma \Downarrow \sigma[result := n]} \qquad \frac{e,\sigma \Downarrow true}{assert\ e;,\sigma \Downarrow \sigma} \qquad \frac{e,\sigma \Downarrow false}{assert\ e;,\sigma \Downarrow \bot}$$

Figure 3.6 Reduction Rules

3.4 Verification Semantics

The language implements a design by contract approach by enforcing contracts statically. Each procedure's body is proven correct according to its specifications. When a rule is violated, an implementation of the proof system described in Section 3.4.1, informs us that either an assertion or the program itself is incorrect.

Expressions defined in specifications are interpreted as symbolic logical expressions, where functions and predicates are not uninterpreted and their bodies correspond to the supplied expressions in their definition. This allows an SMT solver to perform instantiations on ground terms (e.g. given $f(x) = x + 3$, $f(4)$ can be simplified to 7).

3.4.1 Extended Hoare Logic for Recursive Procedures

Figure 3.7 lists the verification rules and axioms used for the language. They are an extension of Hoare logic that includes assertion statements and structured programming capabilities through callable procedures. The rules below are denoted using the usual Hoare triples notation $\{\phi\}\ C\ \{\psi\}$, instead of the directive-like notation that the language uses.

Rule Assignment states that if a specification ϕ holds after an assignment $x := e;$, then a specification $\phi[e/x]$ must hold before the assignment where we all free occurrences of x are substituted with e in the original specification and e must be total. Extending this rule to allow division expressions, rule Assignment Div requires that the divisor is non-zero. Rule Consequence allows the weakening of a precondition and the strengthening of a postcondition of a program segment. Rule Composition allows the derivation of a triple for the sequence of two statements if the postcondition of the first one is the precondition

of the second one. Rule If-Statement allows proving a triple for an if-else statement by considering a triple for each branch of the statement where the statements condition is assumed true on the first branch and false on the second and the triples share the same postcondition. Rule Partial-while allows proving the partial correctness of a triple for a while loop, where an invariant condition is needed to hold before and after the loop's body, where the loop guard holds on the precondition of the body and its negation at the postcondition of the loop. Rule Total-while allows proving the total correctness of a while loop triple, by augmenting the previous rule with a non-negative variant expression required to be strictly decreasing for each iteration.

$$\frac{total(e)}{\{\phi[e/x]\}\ x{:=}e;\ \{\phi\}}\ Assignment \qquad \frac{\phi{\to}\phi_0 \quad \{\phi_0\}\ S\ \{\psi_0\} \quad \psi_0{\to}\psi}{\{\phi\}\ S\ \{\psi\}}\ Consequence$$

$$\frac{\{\phi\}\ S_0\ \{\eta\} \quad \{\eta\}\ S_1\ \{\psi\}}{\{\phi\}\ S_0\ S_1\ \{\psi\}}\ Composition \qquad \frac{\{\phi{\wedge}\beta\}\ \overline{S_0}\ \{\psi\} \quad \{\phi{\wedge}\neg\beta\}\ \overline{S_1}\ \{\psi\}}{\{\phi\}\ if\ \beta\ \{\overline{S_0}\}\ else\ \{\overline{S_1}\}\ \{\psi\}}\ If-Statement$$

$$\frac{\{\eta{\wedge}\beta\ \}\ \bar{S}\ \{\eta\}}{\{\eta\}\ while\ \beta\ \{\bar{S}\}\ \{\eta{\wedge}\neg\beta\}}\ Partial-while \qquad \frac{\{\eta{\wedge}\beta{\wedge}0{\le}e{=}e_0\}\ \bar{S}\ \{\eta{\wedge}0{\le}e{<}e_0\}}{\{\eta{\wedge}0{\le}e\}\ while\ \beta\ \{\bar{S}\}\ \{\eta{\wedge}\neg\beta\}}\ Total-while$$

$$\frac{}{\{\phi{\wedge}\psi\}\ assert\ \phi;\ \{\psi\}}\ Assert \qquad \frac{\{\phi\}\ \bar{S}\ \{\psi\}}{\{\phi\}\ proc\ f(\bar{x})\ \{\bar{S}\}\ \{\psi\}}\ Proc$$

$$\frac{\{\phi\}\ proc\ f(\bar{x})\ \{\overline{S_0}\ return\ e;\ \overline{S_1}\}\ \{\psi\}}{\{\psi[e/result]\}\ return\ e;\ \{\psi\}}\ Return$$

$$\frac{div(\oplus)}{\{\phi[(e_0{\oplus}e_1)/x]{\wedge}e_1{\neq}0\}\ x{:=}e_0{\oplus}e_1;\ \{\phi\}}\ Assignment\ Div$$

$$\frac{\{\phi\}\ proc\ f(\bar{x})\ \{\bar{S}\}\ \{\psi\}}{\{\phi[\overline{e/x}]\}\ y{:=}f(\bar{e});\ \{\psi[\overline{e/x}]\}}\ Partial-call$$

$$\frac{\{\phi{\wedge}0{\le}a\}\ proc\ f(\bar{x})\ \{\bar{S}\}\ \{\psi\} \quad \{\phi'{\wedge}0{\le}a'\}\ proc\ f'(\overline{x'})\ \{\overline{S'}\}\ \{\psi'\} \quad y{:=}f(\bar{e});{\in}\overline{S'} \quad 0{\le}a[\overline{e/x}]{<}a'}{\{\phi[\overline{e/x}]\}\ y{:=}f(\bar{e});\ \{\psi[\overline{e/x}]\}}\ Total-call$$

Figure 3.7 Verification Rules

The rules described above cannot be used to prove the correctness of procedure calls and recursive procedures. The following rules extend Hoare logic for recursive procedures that return a value and use assertion statements. The simplest of these rules is the rule for assertions where the assertion condition should hold before the execution of the assert statement. A triple for a return statement $return\ e;$, has as postcondition ψ its enclosing procedure's postcondition and as precondition $\psi[e/result]$ where the symbolic result expression is substituted with the return expression e. The specification of a procedure is proven correct if the corresponding specification is proven correct for

its body. Procedure call triples can be deduced by substituting the parameters that occur in the specifications of the callee procedure with the arguments supplied to the call. To guarantee termination of directly or mutually recursive calls, the variant of the callee procedure where parameters are substituted by the supplied arguments, must be strictly less than the variant of the caller.

Clark showed that for languages which include recursion it is not possible to obtain a sound and complete Hoare-like system [15]. A similar system to the one described in this section was shown to be sound [16], therefore incomplete and these properties apply to this system using akin argumentation.

3.4.2 Weakest Preconditions

To automate the rules above, Figure 3.8 presents a recursive function to calculate the weakest precondition, given a statement and a postcondition.

$$wp(x \coloneqq e;, \psi) = \psi[e/x], if\ total(e)$$
$$wp(S_0\ S_1, \psi) = wp\big(S_0, wp(S_1, \psi)\big)$$
$$wp(if\ \beta\ \{S_0\}\ else\ \{S_1\}, \psi) = \big(\beta \rightarrow wp(S_0, \psi)\big) \wedge \big(\neg\beta \rightarrow wp(S_1, \psi)\big)$$
$$wp(\#invariant\ e\ \#variant\ v\ while\ \beta\ \{\bar{S}\}, \psi) = e \wedge 0 \leq v,$$
$$if\ \big(e \wedge \beta \wedge 0 \leq v = v_0 \rightarrow wp(\bar{S}, e \wedge 0 \leq v < v_0)\big) \wedge (e \wedge \neg\beta \rightarrow \psi)$$
$$wp(assert\ \phi;, \psi) = \phi \wedge \psi$$
$$wp(return\ e;, \psi) = \psi[e/result]$$
$$wp(y' \coloneqq f(\bar{e});, \psi) = \phi_f\big[\overline{e/x}\big] \wedge v_f\big[\overline{e/x}\big] < v_0 \wedge \forall y\big(\psi_f\big[\overline{e/x}, result/y\big] \rightarrow \psi\big)$$
$$wp(x \coloneqq e_0 \oplus e_1;, \psi) = \psi[(e_0 \oplus e_1)/x] \wedge e_1 \neq 0, if\ div(\oplus)$$
$$wp(\#pre\ \phi\ \#post\ \psi\ \#variant\ v\ proc\ f(\bar{x})\ \{\bar{S}\}, \psi) = wp(\bar{S}, \psi),$$
$$if\ \phi \wedge 0 \leq v \rightarrow wp(\bar{S}, \psi)$$

Figure 3.8 Weakest Precondition Function

When calculating the weakest precondition for a procedure call it is assumed that the relevant statement appears in a procedure declaration
$\#pre\ \phi_f\ \#post\ \psi_f\ \#variant\ v_f\ f(\bar{x})\ \{\bar{S}\}$.

A specification $\#pre\ \phi\ \#post\ \psi\ \#variant\ 0$ for a procedure $proc\ f(\bar{x})\ \{\bar{S}\}$ is valid for partial correctness if $\phi \rightarrow wp(\#pre\ \phi\ \#post\ \psi\ \#variant\ 0\ proc\ f(\bar{x})\ \{\bar{S}\}, \psi)$. A specification $\#pre\ \phi\ \#post\ \psi\ \#variant\ v$ for a procedure $proc\ f(\bar{x})\ \{\bar{S}\}$ is valid for total correctness if $\phi \wedge 0 \leq v \rightarrow wp(\#pre\ \phi\ \#post\ \psi\ \#variant\ v\ proc\ f(\bar{x})\ \{\bar{S}\}, \psi)$. The correctness of the whole program is determined by the correctness of every procedure.

3.5 LLM Assistant Integration

The tool utilizes the capabilities of current large language models to automate the debugging process of a faulty program or the semi-decidable aspects of the verification process through the assertion language. The LLM is treated as an unreliable oracle, and therefore cannot operate unsupervised, and its produced output has no guarantees regarding correctness (or anything else).

The LLM interferes when a program cannot be proven correct using the proof system described in the Section 3.4, by being prompted with the detected error and the original program. The model may respond by suggesting a different specification or altering the original program and explain the reasoning behind its actions. These responses are supervised by the user and can be used to evaluate the newly generated code for correctness or discarded.

Chapter 4 Implementation

A prototype for the language as described in the previous chapter along with integration of an LLM Assistant was implemented in python. Architecturally it follows the structure of a multi-pass compiler, being a hybrid of an interpreter and a static analysis tool with supplementary LLM assistance, shown diagrammatically below:

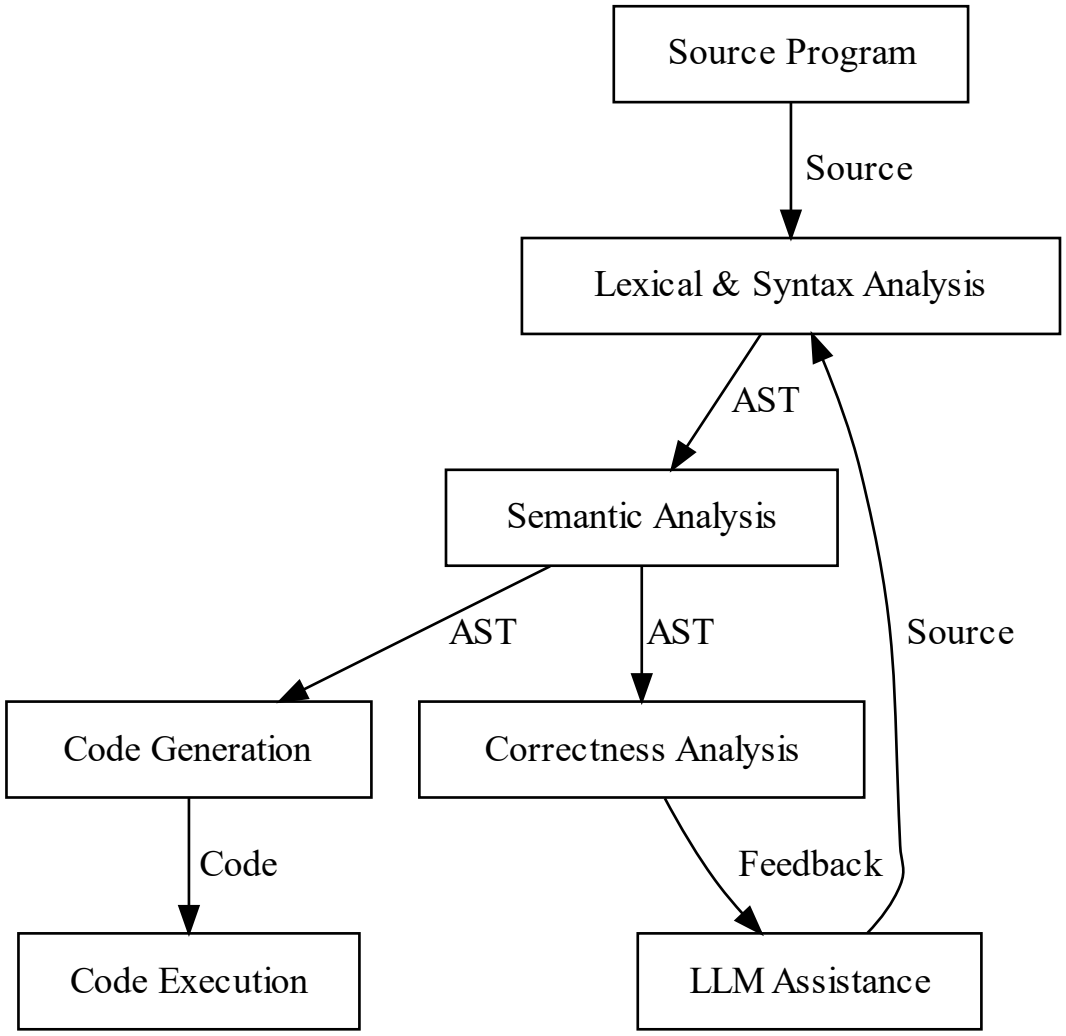

Figure 4.1 Tool Architecture

4.1 Parsing

The concrete syntax chosen for the tool closely resembles the abstract syntax given in Section 3.1. This syntax presented using EBNF notation can be found in Appendix B.

The syntax follows similar lexical rules as the C language. Operators which cannot be represented using ASCII characters are approximated using fairly conventional alternatives (e.g. using != for ≠). Integer literals are represented using signed base 10 notation. Identifier names are identical to standard C identifiers and do not overlap with any keyword. Lexemes may be separated by whitespace characters, which are insignificant. Single line comments prefixed by '//' are allowed and ignored by the parser.

The grammar includes precedence and associativity rules for all operators defined in the previous chapter, in a manner like most C-family languages. Expression grouping using parentheses is allowed. If-else statements may be chained without block nesting. Moreover, metaconditions (prefixed by #) are optional, which allows the user to omit variants if they do not wish to examine termination, or they can omit preconditions when there is no need to validate one, or even postconditions in case that a program should only be executed, not verified. Variants and invariants are also allowed to be omitted if it is desired to receive a proposed such expression by the LLM assistant.

The syntax was deliberately constructed to be easy to parse, as it can be shown that it can be specified by an LL(1) grammar and implement a simple recursive descent parser for it. Having said that, the pyparsing module [17] was used to implement the parser by specifying the respective parsing expression grammar. By executing the parser on an input string, an abstract syntax tree (AST) containing metadata about the nodes text location is created to be further processed, given that the input is syntactically valid.

4.2 Semantics and Typing

During semantic analysis the AST is processed to determine whether the respective program is well formed, construct a symbol table that associates variables with types and declarations with their kinds and produce the symbol table and the call graph between the program procedures.

The well formedness of a program is evaluated using the typing rules described in the previous chapter in a top-down manner using pattern matching, based on the context of the specification. Type inference is also performed at this step on assignments by evaluating the type of the assigned expression and declaring a new variable with the inferred type or detecting ill-formed assignments to variables of different types or parameters which are immutable.

Whenever a rule is violated, the error is reported with a descriptive message and the whole semantic check phase is terminated, no recovery attempts are made.

4.3 Compilation and Execution

An interpreter for the language was also implemented to allow the empirical verification of the correctness or faultiness of the inspected program. The implementation uses a virtual stack-based execution context with a simple instruction set that is described below.

4.3.1 Virtual Machine

The virtual machine is composed of the following elements:

- Program memory, which is a sequence of instructions to be executed.
- Instruction pointer, which points the currently executing instruction.
- Value stack, which contains integer values used by the program.
- Call stack, where the values of the instruction pointer from previous calls are stored.
- Frame stack, which stores segments of the value stack according to calls already executed.
- String table, which contains messages when the program is abruptly terminated by a failed assertion.

There is no concept of word size since the native python integers are used in this implementation, which have unlimited precision. There is no explicit Boolean type either and the zero value is treated as false.

4.3.2 Instruction Set

Below are listed the instructions used by the virtual machine to execute a program. Each instruction may optionally include an integer operand that may be interpreted as a label or an immediate value. All non control flow instructions increment the instruction pointer by one. The program terminates when the instruction pointer reaches the end of program

memory. Division and modulo operations are not defined on zero divisor and this implementation raises a ZeroDivisionError which is part of the python runtime.

The following notation is used below:

- $[Before] \rightarrow [After]$: Stack modification, only the topmost values are shown.
- $base$: $[top\ values]$: Denotes the whole stack.
- $[a] * n$: A stack consisting of n repetitions of the value a.
- $|S|$: The length of the stack S.

Value operations are denoted using a C-like syntax. The assert instruction on failure reaches the end of program and outputs the message provided by its operand.

Mnemonic	Value Stack
NOP	
NEG	[V]→[-V]
NOT	[V]→[!V]
ADD	[V1, V2]→[V1+V2]
SUB	[V1, V2]→[V1-V2]
MUL	[V1, V2]→[V1*V2]
DIV	[V1, V2]→[V1/V2]
MOD	[V1, V2]→[V1%V2]
LT	[V1, V2]→[V1<V2]
LE	[V1, V2]→[V1<=V2]
EQ	[V1, V2]→[V1==V2]
NE	[V1, V2]→[V1!=V2]
GE	[V1, V2]→[V1>=V2]
GT	[V1, V2]→[V1>V2]

Table 4.1 Arithmetic Instructions

Mnemonic	Operand	Value Stack	Frame Stack	Side Effect
LOAD	I	[]→[S[I]]		
STORE	I	[V]→[]		S[I]=V

CONST	I	[]→[I]		
POP		[V]→[]		
ENTER	I	S:[]→S:[0]*max{0,|S|-I}		
FRAME	I	S:[V]*I→S	[]→[[V]*I]	

Table 4.2 Stack Manipulation Instructions

Mnemonic	**Operand**	**Value Stack**	**Call Stack**	**Frame Stack**	**Instruction Pointer**
JMP	I				I
JMP_IF	I	[V]→[V]			V!=0?I:IP+1
JMP_UNLESS	I	[V]→[V]			V==0?I:IP+1
ASSERT	I	[V]→[]			V==0?END:IP+1
CALL	I		[]→[IP]		I
RET		S0:[V]→S:[V]	[I]→[]	[S]→[]	I+1

Table 4.3 Control Flow Instructions

4.3.3 Code Generation

The code generation phase follows closely the reduction rules described in Chapter 3. Where a naive implementation of the interpreter would traverse the AST at runtime and apply the reduction rules, the AST is converted top-down in a sequence of instructions to be interpreted by the virtual machine. Given that the AST is well-formed, the code generation phase should not fail or produce ill-formed code which violates the virtual machine semantics (e.g. popping from an empty stack).

Base expressions are compiled using the CONST instruction. Composite expressions are compiled by recursively generating instructions for the respective subexpressions and emit the necessary instructions to combine them, for example $e_0 + e_1$ would be generated as: $e_0\ code; e_1\ code; ADD$.

Conditional expressions and control flow statements are compiled using backpatching to ‘fix’ jump instruction target that are not known when generating them, and POP instructions to remove unneeded values when branching is determined. Assignments are implemented using the STORE instruction. When compiling an assert statement, after generating the code for the assertion expression the instruction ASSERT is emitted with operand the location of an entry in the string table of the virtual machine with the message displayed in case of a failed assertion.

Procedure calls are generated by compiling their arguments in reverse order and allocating them on the stack frame (using the FRAME instruction), where the callee procedure is to be executed afterwards. Calls to procedures not yet compiled are backpatched after compiling all procedures. Procedures use the ENTER instruction in the same way that many calling conventions require a prologue to allocate space in the stack and a RET instruction as an epilogue. The amount of space to allocate is again, backpatched.

A symbol table with the addresses of the compiled procedures is also produced along with the generated code as there is no single specified entry point, and the user may select which procedure to execute and the supplied arguments using command-line options.

4.4 Verification

The tool detects software faults by applying the extensions of Hoare Logic for recursive procedural programming described in Chapter 3. All expressions that appear in the assertion language and all intermediate conditions are represented using Z3 expressions. Converting AST expressions to Z3 expressions is mostly trivial, since all AST expression nodes have equivalent Z3 nodes which can be constructed through the Z3 API. No function is left uninterpreted, where their Z3 definition is equivalent to their AST declaration expression.

Each program is verified for partial correctness or total correctness where variant expressions are employed. For each procedure and its specification, the weakest

precondition is computed using an implementation of the recursive function *wp*. Each intermediate inferred precondition is checked using the Z3 Automated Theorem Prover, where if the precondition is found unsatisfiable it is presented in a message with its location and verification fails. The provided precondition must imply the calculated precondition (by the consequence rule), and the needed implications of a while loop precondition are checked using the Z3 Automated Theorem Prover. If such implication is not proven to be a tautology, again a descriptive message is presented along with a counterexample, and verification fails. The total call rule is applied if a call is directly or mutually recursive which is determined through the call graph, otherwise the partial rule suffices since termination does not depend on non recursive calls.

Example:

As an example of fault detection using the implementation described, the following faulty program where it is intended that the integer square root of x is computed, is tried to be verified for partial correctness. By computing the weakest precondition of the procedure using backwards inferences traced in the figure below, the precondition computed before the if-else statement is unsatisfiable. By observing the implications that compose the precondition, their consequents contain the negation of their antecedents respectively. The program can be corrected by inverting the if condition, which causes the swapping of the antecedents of the implications of the precondition. The tool produces the following output when attempting to verify this program:

```
buggy/isqrt.hld:13:9: error: precondition `If(m*m <= x,
  And(And(l*l <= x, x < m*m), 0 <= x),
  And(And(m*m <= x, x < r*r), 0 <= x))` found is unsatisfiable
    if m * m <= x {
    ^
```

```
#pre x >= 0
#post result * result <= x && x < (result + 1) * (result + 1)
proc isqrt(x) {
  l := 0;
  r := x + 1;
```

```
#invariant l * l <= x && x < r * r
#variant r - l
while l != r - 1 {
  m := (l + r) / 2;
  {(m * m ≤ x → l * l ≤ x ∧ x < m * m) ∧ (m * m > x → m * m ≤ x ∧ x < r * r)}
  if m * m <= x {
    {l * l ≤ x ∧ x < m * m ∧ l = m − 1}
     r := m;
    {l * l ≤ x ∧ x < r * r ∧ l = r − 1}
  } else {
    {m * m ≤ x ∧ x < r * r ∧ m = r − 1}
     l := m;
    {l * l ≤ x ∧ x < r * r ∧ l = r − 1}
  }
  {l * l ≤ x ∧ x < r * r ∧ l = r − 1}
}
{l * l ≤ x ∧ x < r * r ∧ l = r − 1}
{l * l ≤ x ∧ x < (l + 1) * (l + 1)}
return l;
{result * result ≤ x ∧ x < (result + 1) * (result + 1)}
}
```

Figure 4.2 Faulty Integer Square Root Program Inference Trace

4.5 GPT Assistant

Using OpenAI's assistants and its API, an assistant was created using the gpt-3.5-turbo model by providing some instructions in natural language, where the language is described briefly, and example programs are provided to the model. The assistant is instructed to take in consideration comments starting with '///' as user supplied hints that may help in correcting the faulty program and if it was to generate a new program, to use a specified format that is easily parsed by the tool. The full prompt can be found in Appendix C.

When a program fails to be verified for correctness, the assistant is provided a prompt containing the program and the error through the asynchronous OpenAI API. The tool offers an interactive mode flag, where when a proposed modified program is detected in the assistant's response the user may choose to instantly update their previous program.

Successful Example:

As a successful example, the following figure presents the interaction between the user, the tool and the assistant through the messages displayed by the tool and the assistant's response for the faulty integer square root program shown previously. In this scenario the user chooses to apply the changes proposed by the tool and the program is subsequently verified and the calculated weakest precondition required by the procedure is presented.

```
$ ./hld/run.py buggy/isqrt.hld --ai --interactive
buggy/isqrt.hld:13:9: error: precondition `If(m*m <= x,
  And(And(l*l <= x, x < m*m), 0 <= x),
  And(And(m*m <= x, x < r*r), 0 <= x))` found is unsatisfiable
    if m * m <= x {
    ^
Asking Hoare Logic Debugging Assistant (gpt-3.5-turbo)...
```

The error message you have indicates that the precondition for the `if` statement inside the `isqrt` procedure is found to be unsatisfiable by the compiler. This implies that the condition being checked in the `if` statement contradicts the conditions specified in the preconditions and invariants of the program.

Looking closely at the `if` statement condition `m * m <= x`, it seems to be the cause of the issue. The precondition requires `m * m <= x` to be true, but this condition is directly conflicting with the invariant `l * l <= x`.

To correct this error, you should adjust the logic of the program to ensure consistency between the conditions in the `if` statement, the preconditions, and the invariants. One

possible correction would be to consider the relationship between `m`, `l`, and `r` in the binary search algorithm and update the conditions accordingly.

Here is the corrected version of the `isqrt` procedure:

```
#pre x >= 0
#post result * result <= x && x < (result + 1) * (result + 1)
proc isqrt(x) {
   l := 0;
   r := x + 1;
   #invariant l * l <= x && x < r * r
   #variant r - l
   while l != r - 1 {
      m := (l + r) / 2;
      if m * m > x {
         r := m;
      } else {
         l := m;
      }
   }
   return l;
}
```

In this corrected version, the condition in the `if` statement has been updated to `m * m > x` instead of `m * m <= x` to maintain consistency with the specified preconditions and invariants. This change should resolve the unsatisfiability error.

Apply proposed changes and retry? (Y/n) y

proc isqrt(...) {...} requires `And(Not((1 + x)*(1 + x) <= x), 0 <= x)`

Figure 4.3 Integer Square Root Program Correction Using a GPT Assistant

Failed Example:

As a failed example, the following figure presents the interaction between the user, the tool and the assistant through the messages displayed by the tool and the assistant's response for a faulty sum of consecutive even integers program (displayed using the cat command). In this case, the supplied loop invariant is incorrect and GPT identifies the error but fails to correct it properly, providing an invalid explanation.

```
$ cat examples/sum.hld
#post result == n * (n - 1)
proc sum(n) {
  i := 0;
  total := 0;
  #invariant total == i * (i + 1)
  #variant n - i
  while i != n {
    total := total + 2 * i;
    i := i + 1;
  }
  return total;
}
$ ./hld/run.py examples/sum.hld --ai --interactive
examples/sum.hld:7:16: error: invariant and guard negation do not imply post condition.
        post: total == n*(-1 + n)
counter-example: [i = -2, total = 2, n = -2]
  while i != n {
             ^
Asking Hoare Logic Debugging Assistant (gpt-3.5-turbo)...
```

The error message indicates that the postcondition specified in the program is not implied by the invariant and guard (loop condition) specified in the loop. This suggests an issue with the loop and the way the loop condition is interacting with the postcondition.

To resolve this issue, we need to ensure that the loop condition and the invariant are correctly related to eventually satisfy the postcondition. One way to correct the program is to adjust the loop condition and the invariant.

Here is the corrected program:

```
#post result == n * (n - 1)
proc sum(n) {
  i := 0;
  total := 0;
  #invariant total == i * (2 * i - 1)  /// Update the invariant to match the relationship between total and i
  #variant n - i
  while i != n {
    total := total + 2 * i;
    i := i + 1;
  }
  return total;
}
```

In this corrected version, the invariant has been updated to reflect the relationship between the total and the loop variable `i`. This adjustment ensures that the loop invariant is consistent with the changes happening inside the loop and will help in satisfying the specified postcondition.

Apply proposed changes and retry? (Y/n) n

Figure 4.4 Failed Attempt to Correct a Summation Program Using a GPT Assistant

Chapter 5 Evaluation

5.1 Usability and Accuracy

In this section, the usability of the tool will be discussed qualitatively. In general, the tool offers fine-grained, per-statement precision in detecting errors, assuming that specifications were accurate. The generated messages in case of a detected error, describe the inferred fault precisely according to the formal system used. This does not necessarily mean that the programmer will understand their program's fault, since the tool informs them about the validity of logical conditions, and it may not be immediately clear as to what exactly must corrected.

An early version of the tool which did not include the LLM assistant along with two simple faulty programs to debug were given to a handful of senior undergraduate students that had been previously taught Hoare logic. While they responded mostly positively to the tool and its usage, generally regarding it as helpful and partially succeeding in debugging the said programs, they felt uncomfortable with the amount of Hoare logic required to use and understand the tool.

The language poses some unusual restrictions on division expressions and procedure calls, as they must be directly assigned and not nested anywhere else in the program's code. This is required since these expressions require extra preconditions, and the verification rules are applied at the level of statements. These restrictions can be relaxed if a rewrite (or desugar) step is added after parsing where these expressions are de-nested and assigned to fresh variables.

The integration and usage of the LLM assistant is a double-edged sword. It generally succeeds in finding invariants and variant expressions for many programs. While in many cases it is capable of correcting faulty programs and is responsive to user hints, at times it proposes non-sensical solutions or even ignores the problem completely. Sometimes it can contradict itself between its explanations and proposed code, which can cause frustration and confusion to the programmer. It may also fail to comply to the syntactic rules of the language by prepending invariants before non-loop constructs.

The accuracy of the assistant can be increased by using more recent models such as GPT-4, Gemini, or Llama. Querying the LLM multiple times, merging and repairing the responses increases the precision of the model in finding invariants, [18] therefore similar techniques could be beneficial for this implementation in terms of accuracy. Querying can be done in parallel so that responsiveness will not be affected.

5.2 Performance

Even though performance was not a primary concern in this implementation, it is a detrimental factor for any static analysis tool. Since these tools attempt to solve (or approximate) NP-hard problems, their execution time must scale well with larger input programs. Additionally, this tool aims to be interactive, so that programmers may attempt to correct their programs as often as needed, therefore high response time is unwanted. This implementation written in python, whose runtime has many known caveats concerning its performance by itself, has some issues that although resolvable using various methods, shall be addressed.

By running a profiler, two major bottlenecks where identified: the parser and the OpenAI API. The parsing library used struggles with some inputs causing it to backtrack unnecessarily, and a simpler grammar cannot be easily expressed using the provided functionality. This is easily fixed by creating a handwritten recursive descent parser without backtracking, or by using an ‘industrial’ parser generator. The problems caused by OpenAI’s API were trickier. The needed modules’ import time, which occurs at

runtime, takes a great portion of the total execution time and this is caused by CPython's importing mechanisms. This was partially solved by using lazy imports, i.e. importing the needed modules when needed, in which case execution times are still penalized. A possible solution to this bottleneck, would be rewriting the tool in a compiled language where no such imports at runtime are necessary.

Another contributing factor to the performance of this tool, is the assistant's response time. Current LLMs need tens of seconds to respond to a few hundred tokens, which damages the tools latency even further.

Lastly, the SMT solver, needs a significant share of resources to determine the satisfiability of the computed properties. Some calls to the solver can be avoided, as this implementation checks the satisfiability of every inferred precondition at each statement. Instead, the solver can be called on fewer statements and then backtracking can be used to detect the offending statement.

5.3 Limitations

The language described in this thesis is computationally universal, as demonstrated by the program below which simulates rule 110, a known universal cellular automaton [19]. The integer n represents the state of the automaton as a binary vector. Since the language does not support bitwise operations, they are emulated using arithmetic and relational operators (addition of powers of two is equivalent to bitwise or, $n \,\%\, 2^{k+1} \geq 2^k$ is true if and only if the kth bit of n is set). Variables l, c, r represent the left, centre (self), and right cells respectively of the $\log_2(i)$ cell, used by the if condition to determine the $\log_2(i)$ cell of the next state m. The procedure terminates when no set cells with no set neighbouring cells remain, since their next state will always be zero.

```
proc rule110(n) {
  m := 0; i := 1;
  t := n % 4; l := t >= 2;
  t := n % 2; c := t >= 1;
```

```
    r := false;
    while i <= n * 2 {
        if (c || r) && (!l || !c || !r) {
            m := m + i;
        } else {}
        r := c; c := l;
        i := i * 2;
        t := n % (i * 4); l := t >= i * 2;
    }
    return m;
}
```

Figure 5.1 Procedure that Computes the Next State of Rule 110.

Even though any computable function can be expressed in this language, it poses some practical difficulties. Most general-purpose programming languages do not use arbitrary precision integers. It is possible to simulate bit-vectors with any overflow behavior (wrapping, undefined, saturated, etc.) using integers and a few user-defined functions and predicates. However, native support could be added to the tool without burdening the programmer, by extending the syntax to include type annotations, the type system to include bit-vectors and utilizing the Z3 bit-vector values. Support for memory references through arrays, pointers, and composite data types, would turn this tool closer to a typical programming language and the verification step would be powered by a suitable flavor of separation logic.

The tool does not restrict the programmer in any way in expressing specifications, therefore it can fail to terminate when verifying a program. Recursive functions and predicates are not checked for termination, but employing a paradigm such as total functional programming on these structures is possible and potentially beneficial.

Chapter 6 Conclusion

6.1 Summary

In this project a tool was devised to assist in identifying and resolving software faults, facilitating formal methods and the competences of LLMs. The tool ensures that programs comply to their specifications and leads the user to the necessary actions to correct their programs otherwise. LLMs can bridge the gap between the complexity of Hoare-like systems, accelerating debugging and making formal verification more accessible, by providing insight into the deductions constructed by the tool.

Through empirical evaluation and a small-scale survey, we conclude that this tool is deemed as useful by programmers with a background in Hoare-like proof systems. By utilizing the capabilities of LLMs, the barrier to entry for using formal methods is significantly lowered, even allowing the expression of specifications and bug causes in natural language.

This work opens new avenues for future research in combining artificial intelligence with formal methods, aiming to further refine the user experience and expand the capabilities of such tools. By exploring this synergy, we can move towards accessible, robust, and error-free software development.

6.2 Challenges

During this project, several challenges were encountered. One major hurdle was the design of the proof system, which required to carefully create and understand suitable extensions to Hoare logic, ensuring soundness. Moreover, the Z3 API presented difficulties due to its vastness and steep learning curve. Finally, the performance issues that emerged were a concern; the parsing library used to generate ASTs, python's runtime and OpenAI API's latency affected the tool's responsiveness. These challenges were addressed through iterative refinement and optimization of the implementation achieving the primary goals of this thesis.

6.3 Future Work

There are many possible extensions to the tool described in this thesis. The tool can be further developed to include more expressive language constructs such as memory references, composite data structures and dynamic methods, as well as to incorporate other approaches in formal methods such as symbolic execution or model checking. The communication between the tool and the LLMs can be implemented using more sophisticated approaches [18], to better the accuracy and reliability of the model, thereby improving user experience and accelerating software development.

There are countless static analysis tools with more features than the one developed during this thesis, that can be reused. By focusing on the synergy of LLMs and formal methods, different strategies can be implemented by providing information to the LLM using static analysis tools to debug or even synthesise programs, under the user's supervision or fully automated. Formal methods allow users to trust the responses generated by the LLMs as they can guarantee that the programs comply to certain specifications [20].

Bibliography

[1] T. Britton, L. Jeng, G. Carver, T. Katzenellenbogen and P. Cheak, "Reversible Debugging Software "Quantify the time and cost saved using reversible debuggers"," 2020. [Online]. Available: https://www.researchgate.net/publication/345843594_Reversible_Debugging_Software_Quantify_the_time_and_cost_saved_using_reversible_debuggers.

[2] "GitHub Copilot," GitHub, Inc., 2024. [Online]. Available: https://github.com/features/copilot.

[3] A. Taylor, A. Vassar, J. Renzella and H. Pearce, "Models, dcc --help: Transforming the Role of the Compiler by Generating Context-Aware Error Explanations with Large Language," in *SIGCSE 2024: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, Portland, 2024.

[4] N. Alzahrani and F. Vahid, "Common Logic Errors for Programming Learners: A Three-decade Literature Survey," in *2021 ASEE Virtual Annual Conference Content Access*, 2021.

[5] "Langley Formal Methods Program • What is Formal Methods," [Online]. Available: https://shemesh.larc.nasa.gov/fm/fm-what.html.

[6] C. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576-580, 1969.

[7] C. Hoare, "Procedures and Parameters: An Axiomatic Approach," in *Symposium on Semantics of Algorithmic Languages*, 1971.

[8] J. C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *LICS*, 2002.

[9] E. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communications for the ACM*, vol. 18, no. 8, pp. 453-457, 1975.

[10] N. Bjørner, C. Eisenhofer, A. Gurfinkel, N. P. Lopes, L. de Moura, L. Nachmanson and C. Wintersteiger, "Z3 Internals (Draft)," Microsoft Research, [Online]. Available: https://z3prover.github.io/papers/z3internals.html#sec-solver-cores.

[11] B. Meyer, "Applying "Design by Contract"," *Computer (IEEE)*, vol. 25, no. 10, pp. 40-51, 1992.

[12] "What are large language models (LLMs)?," [Online]. Available: https://www.ibm.com/topics/large-language-models.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, "Attention is All you Need," no. 30, pp. 5998-6008, 2017.

[14] R. L. F. D. R. C. K. Rustan M. Leino, "Dafny Reference Manua," 29 April 2022. [Online]. Available: https://dafny.org/dafny/DafnyRef/out/DafnyRef.pdf.

[15] "About Infer," Meta Platforms, Inc., [Online]. Available: https://fbinfer.com/docs/about-Infer.

[16] "Kernel & Plugins - Frama-C," [Online]. Available: https://www.frama-c.com/html/kernel.html.

[17] E. Clark, "Programming language constructs for which it is impossible to obtain good Hoare axiom systems," *Journal of the ACM*, vol. 26, no. 1, pp. 129-147, 1979.

[18] E.-R. Olderog, "On the notion of expressiveness and the rule of adaptation," *Theoretical Computer Science*, vol. 24, pp. 337-347, 1983.

[19] "pyparsing - PyPI," [Online]. Available: https://pypi.org/project/pyparsing/.

[20] K. Adharsh, S. Aditya, C. Saikat, D. Pantazis, K. L. Shuvendu, L. Akash, R. Aseem, R. Subhajit and S. Rahul, "Finding Inductive Loop Invariants using Large Language Models," 14 November 2023. [Online]. Available: https://arxiv.org/abs/2311.07948.

[21] M. Cook, "Universality in Elementary Cellular Automata," *Complex Systems*, no. 15, pp. 1-40, 2004.

[22] M. R. H. Missu, C. V. Lopes, M. Iris and J. Noble, "Towards AI-Assisted Synthesis of Verified Dafny Methods," 1 February 2024. [Online]. Available: https://arxiv.org/abs/2402.00247.

Appendix A

Implementation Code

The code for the implementation is publicly available at https://github.com/gevang03/epl401-hoare-logic-debugger.

User's Guide

Introduction

The purpose of this tool is to assist in finding bugs in programs, thus making programming easier. The tool uses formal methods to detect bugs and verify correctness and termination. It includes an integrated large language model that can assist in correcting found bugs.

Build

On Linux, run the following commands. Make sure that `python3 --version` is 3.9.18.

Run:

```
$ git clone --depth=1 https://github.com/gevang03/epl401-hoare-logic-debugger.git
$ cd epl401-hoare-logic-debugger
$ python3 -m venv .venv
$ . .venv/bin/activate
$ pip install --require-virtualenv -r requirements.txt
```

Usage

Synopsis: `$ ./hld/run.py [OPTIONS] FILE`

Run `./hld/run.py FILE` to check for partial correctness

Run `./hld/run.py --total FILE` to check for total correctness (termination)

Run `./hld/run.py --run 'f x y' FILE` to execute procedure f with arguments x and y.

Run `./hld/run.py --ai FILE` to ask GPT assistant about error.

Run `./hld/run.py --ai --interactive FILE` to interactively apply GPT assistant's changes.

Run `./hld/run.py -h` for help.

Language

The language that the tool supports is procedural and should be familiar to any C/Java programmer. There also exist some constructs to express program specifications.

Types and Basic Expressions

Only two value types exist in HLD: Booleans (`true`, `false`) signed integers (represented in base 10)

The following prefix/infix operators are defined with the same meaning, precedence, and associativity as in C or Java:

- Arithmetic: `+, -, *, /, %`
- Boolean: `!, &&, ||, ->`
- Relational: `<, <=, ==, !=, >=, >`
- Conditional: `?:`

Parentheses `()` can be used to group expressions. Relational operators are defined for integer typed expressions only.

Variables

Variables can be declared/assigned:

```
// c++ style comments
x := 11;
p := false;
y := x + 3;
```

The division and modulo operator cannot be nested in expressions, they must be directly assigned to a variable.

Example:

```
temp := x / y;
a := temp + 1;
// `a := (x / y) + 1;` is illegal
```

Control Flow

The following control flow structures are available:

if-else statements:

```
if x < 1 { // note that braces {} are mandatory.
    y := 3;
} else {
    y := x + 1;
}
```

while loops: Note: while loop conditions are referred as guards by the tool.

```
while x < 10 {
    s := s + x;
    x := x + 1;
}
```

return statements:

```
proc foo() {
    // ...
    return -1;
}
```

Assert statements: must be supplied a Boolean condition which should evaluate to true for every valid execution of a program:

```
assert x > 0;
```

Procedures

Procedures are the basic abstraction used to define specifications and discover bugs in a program.

- Procedures accept a number of integer parameters and return a single integer value. Parameters cannot be reassigned (they are immutable).
- Procedures must end with a return statement in a every reaching path of execution.
- Procedures may be preceded by a precondition (a condition which should hold before the execution of the procedure).
- Procedures may be preceded by a postcondition (a condition which should hold after the execution of the procedure). The result keyword is used to represent the value returned by the procedure.

Example:

```
#pre x >= 0           // precondition
#post result == x + 1 // postcondition
proc inc(x) {
    y := x + 1;
    return y;
}
```

Procedure Calls

Procedures may be called with the required number of arguments. However, they must be directly assigned, not nested in any other expressions.

Example:

```
#pre x >= 0
#post result == x + 2
proc inc2(x) {
    x1 := inc(x);
    x2 := inc(x1);
```

```
    // `return inc(inc(x));` is illegal
    return x2;
}
```

Variants and Invariants

Invariants are conditions which must hold before, during and after the execution of an iteration of a while loop/recursive call. They are used to prove the correctness of these iterative structures.

Variants are integer expressions which each iteration must be decreased and are always bound by zero. They are used to prove termination of while loops/recursive calls.

Example:

```
#pre n >= 0
#post result == n * (n - 1) / 2
proc sum(n) {
    i := 0;
    total := 0;
    #invariant total == i * (i - 1) / 2
    #variant n - i
    while i != n {
        total := total + i;
        i := i + 1;
    }
    return total;
}
```

Functions and Predicates

Functions and predicates are used to define other specifications, that may require recursion to do so, for example. Their body consists of a single integral or Boolean expression respectively. Functions and predicates cannot be called inside of a procedure.

Examples:

```
// example predicate for divides operator
pred divides(a, b) := b % a == 0;

// 'equivalent' to:
// function fct(n) {
//     if (n <= 0) {
//         return 1;
//     } else {
//         return n * fct(n - 1);
//     }
// }
fn fct(n) := n <= 0 ? 1 : n * fct(n - 1);

// note that variant here is used to prove the termination of a recurs
ive procedure
#pre x >= 0
```

```
#post result == fct(x)
#variant x
proc calc_fct(x) {
  if x == 0 {
    return 1;
  } else {
    y := calc_fct(x-1);
    return x * y;
  }
}
```

Quantified Expressions

Quantified expressions using the forall end exists keywords, can be used in specification if necessary. Quantified expressions can bind multiple variables (`forall x. forall y. p(x, y) == forall x y. p(x, y)`).

```
pred prime(n) := n > 1 && forall i. 2 <= i && i < n -> n % i != 0;
```

Appendix B

Concrete Syntax in EBNF Notation

(* Whitespace and single line comments prefixed by ‘//’ are not significant and omitted for brevity. *)

```
program = declaration, {declaration};

declaration = procedure | fn_or_pred;

procedure = [pre], [post], [variant], 'proc', identifier, paramlist, block;
fn_or_pred = ('fn' | 'pred'), ident, paramlist, ':=', expr, ';';

(* STATEMENTS *)
statement = ifelse | assert | assignment | while | return;
ifelse = 'if', expression, block, 'else', (ifelse | block);
assert = 'assert', expression, ';';
assignment = identifier, ':=', expression, ';';
while = [invariant], [variant], 'while', expression, block;
block = '{', {statement}, '}';
return = 'return', expression, ';';

(* EXPRESSIONS *)
expression = primary | ternary_expr;

(* Using regex: /[a-zA-Z_][a-zA-Z0-9_]*/ *)
(* Matches from keywords are excluded *)
identifier = ((alpha | '_'), {alphanum | '_'}) - keyword;
call = identifier, '(', [expression, {',', expression}], ')';
quantified = ('forall' | 'exists'), identifier, {identfier}, '.', expr;

literal = bool | int;
```

```
bool = 'true' | 'false';
int = digits, {digits};

(* ASSERTIONS *)
pre = '#pre', expression;
post = '#post', expression;
invariant = '#invariant', expression;
variant = '#variant', expression;

(* AUXILIARY *)
paramlist = '(', [identifier, {',', identifier}], ')';

alphanum = alpha | digits;
alpha = lower | upper;
lower = ? letters a to z ?;
upper = ? letters A to Z ?;
digits = ? digits 0 to 9 ?;

ternary_expr = impl_expr, '?', ternary_expr, ':', ternary_expr;
impl_expr = or_expr, '||', or_expr;
or_expr = and_expr, '||', and_expr;
and_expr = rel_expr, '&&', rel_expr;
rel_expr = add_expr, rel_op, add_expr;
add_expr = mul_expr, add_op, mul_expr;
mul_expr = unary_expr, mul_op, unary_expr;
unary_expr = un_op, primary;
primary = '(' expression ')' | literal | identifier | call | 'result' | quantified;

rel_op = '<' | '<=' | '==' | '!=' | '>=' | '>';
add_op = '+' | '-';
mul_op = '*' | '/' | '%';
un_op = '+' | '-' | '!';
```

```
keyword = 'assert' | 'if' | 'else' | 'proc' | 'fn' | 'while' | 'true' | 'false' | 'return'
  | 'result' | 'forall' | 'exists' | 'pred'
  | '#pre' | '#post' | '#invariant' | '#variant';
```

Appendix C

OpenAI assistant prompt

You are an assistant for a procedural programming language where each procedure is verified using Hoare logic and backwards inferences. Hoare Logic clauses are prefixed by '#'. If statements are always followed by an else statement. Invariants and variants do not affect program behavior. Invariants appear only before loops, variants appear before loops and procedures. fn functions are used to describe specifications. The result expression refers to the value returned by the procedure. Values prefixed by `e!` refer to variant bounds. Comments starting with `///` are user hints about what they think the error is. Users provide the program and an error message trying to figure out what is wrong with their program. Either some statement or expression in the program, or some precondition, postcondition, variant, invariant may be wrong. If a correction would be suggested, supply the whole program. Example correct programs are provided in file retrieval